

Grundlagen

Logik & Quantoren

Prädikate

- Aussagen über mathematische Objekte, können wahr oder falsch sein
- „Funktionen“ mit booleschen Rückgabewerten
- $P, Q(n), R(x, y, z)$

Verknüpfungen

- UND: $P \wedge Q$
- ODER: $P \vee Q$
- NICHT: $\neg P$
- Implikation: $P \Rightarrow Q = \neg P \vee Q$

Distributivgesetze

Verträglichkeit von \wedge und \vee

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$$

$$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$$

de Morgan

Verträglichkeit von \neg mit \wedge und \vee

$$\neg(P \wedge Q) = \neg P \vee \neg Q$$

$$\neg(P \vee Q) = \neg P \wedge \neg Q$$

Negation und Implikation:

$$P \Rightarrow Q = \neg Q \Rightarrow \neg P$$

Kontraposition

$$\neg \forall i \in \{1, \dots, n\} (P_i) \Leftrightarrow \neg (P_1 \wedge \dots \wedge P_n) \Leftrightarrow \neg P_1 \vee \dots \vee \neg P_n \Leftrightarrow \exists i \in \{1, \dots, n\} (\neg P_i)$$

Quantoren

Allgemeiner Quantor

$$\text{UND-Verknüpfung: } n \text{ Prädikate } P_i, \quad P_1 \wedge \dots \wedge P_n = \bigwedge_{i=1}^n P_i$$

Interpretation

$$P_i \text{ ist wahr für alle } i = 1, \dots, n$$

All-Quantor \forall

$\forall i \in \{1, \dots, n\} (P_i)$

„Für alle i ist P_i wahr.“

All-Quantor \forall

Existenz-Quantor \exists

$$\text{ODER-Verknüpfung: } n \text{ Prädikate } P_i, \quad P_1 \vee \dots \vee P_n = \bigvee_{i=0}^n P_i$$

Interpretation

$$P_i \text{ ist wahr für mindestens ein } i \text{ in } 0, \dots, n$$

Existenz-Quantor \exists

$\exists i \in \{1, \dots, n\} (P_i)$

„Es gibt ein i derart, dass P_i wahr ist.“

Negation = de Morgan 2.0

„Nicht für alle“ = „Es gibt einen Fall, für den nicht“

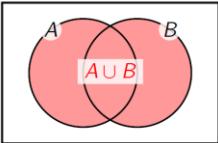
Normalformen: Konjunktiv: UND von ODER-Thermen, Disjunktiv: ODER von UND-Thermen

Mengenlehre

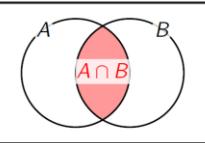
Mengen = «Primitive Datentypen der Mathematik», Zusammenfassung bestimmter Objekte.

Konstruktion: Grundmenge G , Prädikat $P(x)$ $A = \{x \in G \mid P(x)\}$

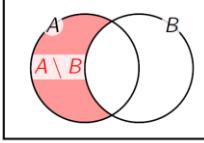
Vereinigung ODER



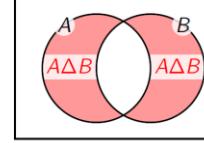
Schnittmenge UND



Differenzmenge



Symmetrische Differenz XOR



Paare und Tupel, Graphen

Paare und Tupel

Tupel = „Die zusammengesetzten Datentypen der Mathematik“

Paare

A, B Mengen, Menge aller Paare:

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$$

2-Tupel

Anwendungsbeispiele

- $\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$ = Koordinatenebene
- Funktion: $\{(x, f(x)) \mid x \in \mathbb{R}\}$

- Brüche sind Paare (a, b) , meistens geschrieben als $\frac{a}{b}$
- ganze Zahlen sind Paare (s, h) , $s=\text{Soll}$, $h=\text{Haben}$

n-Tupel

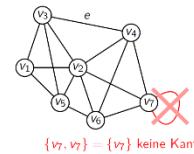
Menge A_0, \dots, A_n , Menge aller n -Tupel

$$\prod_{i=0}^n A_i = \{(a_0, a_1, \dots, a_n) \mid a_i \in A_i\}$$

Graphen

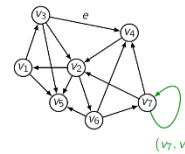
Ungerichteter Graph

- Vertizes $V = \{v_1, v_2, \dots, v_n\}$
- Kante $e = \{v_3, v_4\}$
- Kantenmenge $E = \{e \mid e \text{ Kante}\}$



Gerichteter Graph

- Vertizes $V = \{v_1, v_2, \dots, v_n\}$
- Kante $e = (v_3, v_4)$
- Kantenmenge $E = \{e \mid e \text{ Kante}\}$



Sprachen

Alphabet: Eine nichtleere endliche Menge $\Sigma = \{\dots\}$

Zeichen: Elemente der Menge Σ

Wort: Zeichenkette (Tupel aus Elementen des Alphabets der Länge $n \rightarrow w \in \Sigma^n$)

Leeres Wort: Zeichenkette $\varepsilon \in \Sigma^0 = \{\varepsilon\}$ der Länge 0

Menge aller Wörter: $\Sigma^* = \{\varepsilon\} \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots = \bigcup_{k=0}^{\infty} \Sigma^k$

Wortlänge: $w \in \Sigma^n \rightarrow |w| = n$

$w \in \Sigma^n \& a \in \Sigma \rightarrow |w|_a = \text{Anzahl Zeichen } a \text{ im Wort } w$

Bsp: $|01010|_1 = 2$, $|a^5b^3|_a = 5$ ($b^3 = bbb$)

Sprache: Teilmenge $L \subset \Sigma^*$ → Zu jeder Maschine M gibt es eine Sprache $L(M)$

Bsp: $\Sigma = \{0, 1\}$, Sprache aller Binärstrings: $L = \Sigma^*$

Reguläre Sprachen

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

Die regulären Sprachen sind kontextfrei.

Deterministischer Endlicher Automat (DEA)

Ein endlicher Automat hat keine Speicher, nur Zustände

Definition

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

► Zustände: $Q = \{q_0, q_1, \dots, q_n\}$

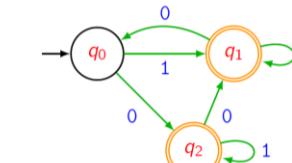
► Alphabet: Σ

► Übergangsfunktion: $\delta: Q \times \Sigma \rightarrow Q$

► Startzustand: $q_0 \in Q$

► Akzeptierzustände: $F \subset Q$

Zustandsdiagramm von A



Tabellenform des DEA A

	0	1	→ Σ
$q \rightarrow$	q_0	q_2	
q_1 / F	q_1	q_0	δ
q_2 / F	q_2	q_1	

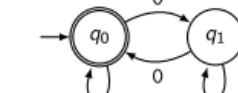
Myrell-Nerode Automat (Rekonstruiere DEA):

Mittels dem Satz von Myrell-Nerode kann erkannt werden, ob eine Sprache regulär ist.

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

w	$L(w)$	Q
Σ	$L(\Sigma) = L$	q_0
0	$L(0) = \{w \in \Sigma^* \mid w _0 \text{ ungerade}\}$	q_1
1	$L(1) = \{w \in \Sigma^* \mid w _0 \text{ gerade}\} = L$	q_0
10	$L(0) = \{w \in \Sigma^* \mid w _0 \text{ ungerade}\}$	q_1
...

⇒ Es gibt zwei Zustände.



Ergibt dieser Automat keine endliche Menge von Zuständen, dann gibt es kein DEA.

Unendlichkeit

Eine Unendliche Menge A heisst abzählbar Unendlich, wenn sie gleichmächtig ist wie die natürlichen Zahlen. A heisst überabzählbar unendlich, wenn es keine Bijektion zwischen N und A gibt.

Minimaler Automat

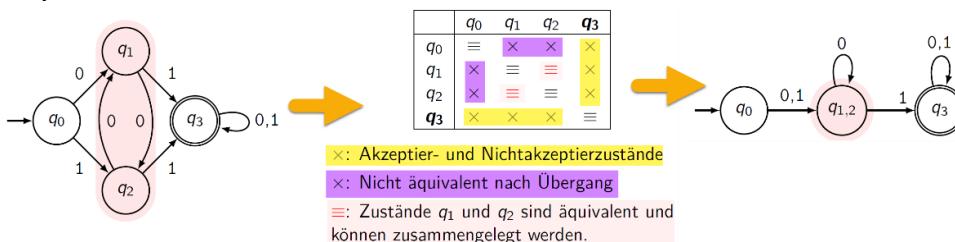
Dient als Beweis ob zwei Automaten die gleiche Sprache akzeptieren: $L(A) = L(B)$

Jeder DEA hat ein Minimalautomat und kann so mit einem andern verglichen werden.

Ablauf:

1. Tabelle aller Zustände erstellen
2. Alle Zustände zu sich selber äquivalent markieren $\rightarrow \equiv$
3. Paare aus einem Akzeptier- und einem Nichtakzeptierzustand $\rightarrow X$
4. Zweifingertechnik:
 - a. Von einem Paar alle Übergänge testen, führt zu einem Paar mit $X?$ Ja $\rightarrow X$
 - b. Wiederhole a) für alle Paare ohne Zuordnung
5. Wiederhole 4. Bis kein neues X gesetzt werden kann.
6. Restliche Zustände sind Äquivalent ($\rightarrow \equiv$) und können minimiert werden.

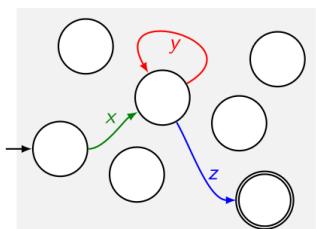
Beispiel:



Pumping Lemma

Idee: Wenn ein Wort länger sein kann als die Anzahl zustände, dann muss mindestens ein Zustand mehrmals vorkommen im endlichen Automat. Das heisst ein Wort ist aufpumpbar.

Man kann damit aber nur nachweisen, dass eine Sprache **nicht regulär** ist, wenn sie mindestens ein Wort enthält, das nicht aufpumpbar ist. \rightarrow Widerspruchsbeweis.



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. $xy^kz \in L \quad \forall k \in \mathbb{N}$

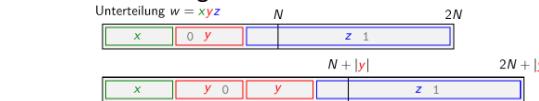
Beweis/Ablauf

1. Annahme: L ist regulär
2. Es gibt die Pumping Length N
3. Wähle ein Wort $w \in L$ mit $|w| \geq N$, Definition mit N schreiben
4. Aufteilung des Wortes gemäss Pumping Lemma $w = xyz$, $|xy| \leq N$, $|y| > 0$
5. Auswirkung des Pumpens mit Begründung
6. Widerspruch und Schlussfolgerung

Beispiele

$$L = \{0^n 1^n \mid n \geq 0\}$$

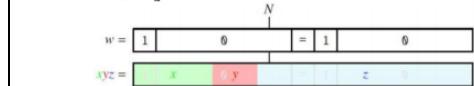
1. Annahme: L ist regulär
2. Pumping Length: $\exists N \in \mathbb{N}$
3. $w = 0^N 1^N$
4. Aufteilung:



5. Pumpen erhöht nur die Anzahl 0, jedoch nicht Anzahl 1
6. Widerspruch: $xy^kz \notin L$ für $k \neq 1$, im Widerspruch zum Pumping Lemma Wort ist nicht mehr Element der Sprache für $k \neq 1$, daher ist L nicht regulär

$$L = \{w = w \mid w \in \{0,1\}^*\}$$

1. Annahme: L ist regulär
2. Pumping Length: $\exists N \in \mathbb{N}$
3. $w = 10^N$
4. Aufteilung:



5. Pumpen: Auf der linken Seite des Gleichheitszeichens wird eine grössere Binärzahl, wie auf der rechten Seite, stehen.
6. Widerspruch: L ist nicht regulär

Nichtdeterministische endliche Automaten (NEA):

Automat kann sich aufhängen. Jede Verzweigung muss einzeln geprüft werden.

Nichtdeterministische endliche Automaten

Definition NEA

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

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

Definition Akzeptieren

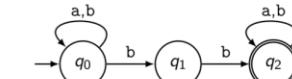
Ein NEA A akzeptiert das Wort $w \in \Sigma^*$, wenn es eine Wahl von Übergängen gibt, derart, dass das Wort w den Automaten in einen Akzeptierzustand überführt.

Faustregel

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

Beispiel

$$\Sigma = \{a, b\}, \\ L = \left\{ w \in \Sigma^* \mid \begin{array}{l} w \text{ enthält zwei } b \\ \text{nacheinander} \end{array} \right\}$$

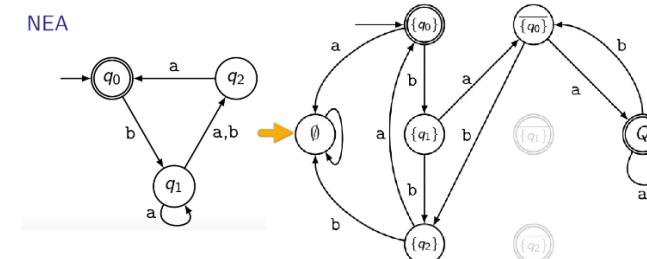


ϵ -Übergänge: Übergang ist Gratis, jeder NEA kann immer in NEA umgewandelt werden

Transformation NEA \rightarrow DEA

1. Alle Kombinationszustände einzeichnen
2. Alle Zustände, mit einem enthaltenen Akzeptierzustand \rightarrow Akzeptierzustände
3. Vom Startzustand ausgehend alle Übergänge einzeichnen.

DEA



Mengenoperationen

Wenn A1 und A2 DEA's sind so kann man Mengenoperationen darauf anwenden und erhält so immer wieder einen DEA, also wieder eine reguläre Sprache.

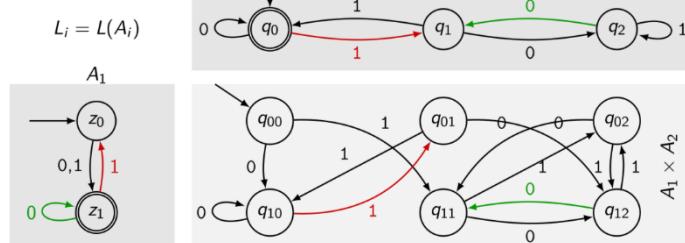
Operationen:

- Schnittmenge $L_1 \cap L_2: F = F_1 \chi F_2$
- Vereinigung $L_1 \cup L_2: F = F_1 \chi Q_2 \cup Q_1 \chi F_2$
- Differenz $L_1 \setminus L_2: F = F_1 \chi (Q_2 \setminus F_2)$

Produktautomat

1. Beide Automaten, wenn möglich, auf eine Linie strecken/dehnen
2. Beide Automaten als Tabelle darstellen, dass die Startzustände einander zugeneigt sind
3. Alle Übergänge einzeln übertragen. Wobei ein Zustandswechsel des einen Automaten sich auf die Horizontale auswirkt und bei der anderen auf die Vertikale.
4. Zustände des neuen Automaten einzeichnen
 - a. Jedes Paar aus A1 und A2 Zustände ergeben ein neues
 - b. Je nach Operation Akzeptierzustände einzeichnen
 - i. **Schnittmenge ($L_1 \cap L_2$):** Alle Zustände, welche bei beiden Automaten Akzeptierzustände sind.
 - ii. **Vereinigung ($L_1 \cup L_2$):** Alle Zustände, welche bei min einem Automaten ein Akzeptierzustand ist.
 - iii. **Differenz ($L_1 \setminus L_2$):** Alle Zustände, welche nur beim ersten Automaten ein Akzeptierzustand sind.

Produktautomat



Akzeptierzustände:

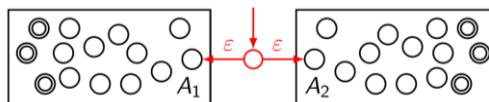
$$L_1 \cap L_2 : \{q_{10}\}$$

$$L_1 \cup L_2 : \{q_{00}, q_{10}, q_{11}, q_{12}\}$$

$$L_1 \setminus L_2 : \{q_{11}, q_{12}\}$$

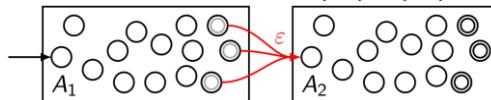
Reguläre Operationen

1. Vereinigung/Alternative (Oder) : $L = L_1 \cup L_2 = L(A_1) \cup L(A_2)$



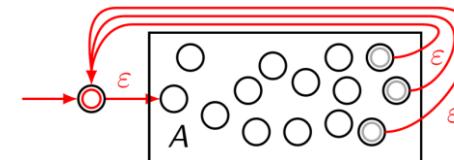
Neuer Startzustand mit ϵ -Übergängen zu den einzelnen Automaten

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



A_1 Akzeptierzuständen mit ϵ -Übergängen zu Startzugang A_2 , A_1 hat keine Akzeptierzustände mehr

3. *-Operation: $L^* = \{ \epsilon \} \cup L \cup L^2 \cup \dots = \bigcup_{k=0}^{\infty} L^k$



Neuer Startzustand mit ϵ -Übergang zum alten Startzustand, A-Akzeptierzustände werden mit ϵ -Übergang verknüpft mit neuem Startzugang.
Leeres Wort ist ebenso Akzeptiert

Reguläre Ausdrücke

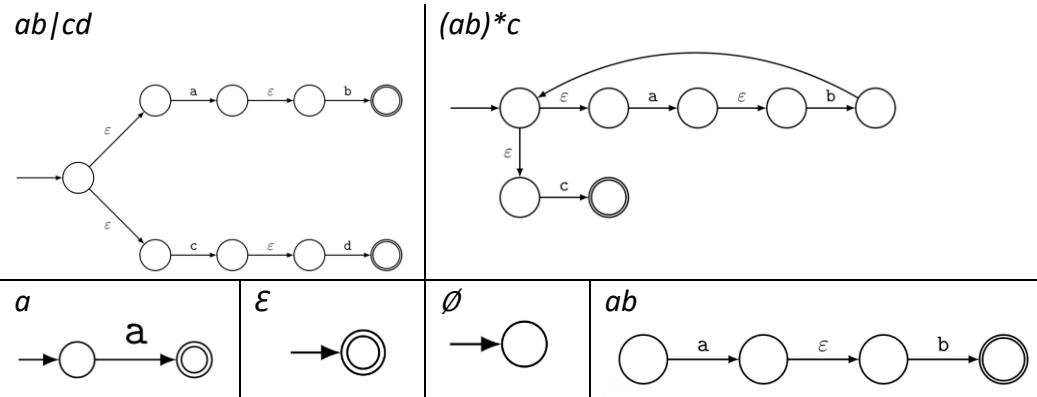
Ausdruck r	$L(r)$	Bedeutung
a	{a}	Steht für das Zeichen $a \in \Sigma$
.	Σ	Steht für ein beliebiges Zeichen aus Σ
[abc]	{a, b, c, d}	Steht für ein Zeichen aus {...} $\in \Sigma$
[1-9]	{1, 2, ..., 9}	Steht für eine positive Ziffer
	{ ϵ }	Steht fürs leere Wort
	{}	Steht für die leere Sprache
[^xyz]		Nicht x, y oder z
a^*	{ a^* }	0 oder beliebig viele a
a^+	{ aa^* }	Mindestens 1 a
a? oder a		0 oder 1 mal a
$a\{5\}$	{ a^5 }	a genau 5 Mal
$a\{2,\}$	{ aa^* }	a mindestens 2 mal
$a\{1,3\}$		a zwischen 1 oder 3 mal.
$a\{4\}$		a maximal 4 mal.
ab/cd		ab oder cd

Umwandlung Regex -> NEA

1. Von innen nach Aussen übersetzen (Klammern zuerst)
2. Einzelne Teile mittels ϵ -Übergänge verbinden

Beispiele:

$$ab/cd$$

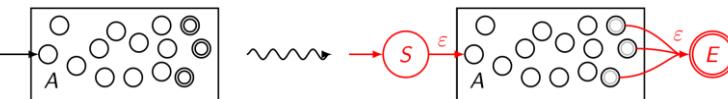


Umwandlung DEA -> VNEA mit einem Übergang

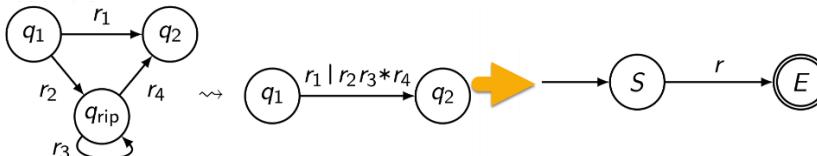
VNEA = NEA mit rexex-Übergängen

Vorgehen

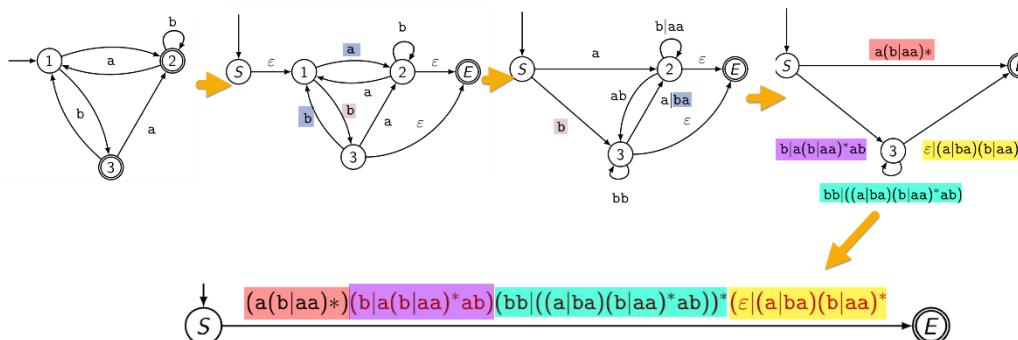
- Startzustand darf keine Übergänge haben und Akzeptierzustand muss minimiert werden:



- Reduktion: Alle Zwischenzustände entfernen. Übrig bleibt ein regulärer Ausdruck:



Beispiel



Kontextfreie Sprachen

Kontextfreie Sprachen können nur von nichtdeterministischen Stackautomaten erkannt werden. Existiert ein Stackautomat, Regex oder eine Grammatik, ist die Sprache kontextfrei.

Der Name «kontextfrei» basiert darauf, dass die Regeln nicht auf den Kontext ankommt in dem die Variable auf der linken Seite sich befindet.

Kontextfreie Grammatik

Eine kontextfreie Grammatik ist ein Quadrupel $G = (V, \Sigma, R, S)$. Dabei gelten folgende Rahmenbedingungen:

- V ist eine endliche Menge von Variablen
- Σ ist eine endliche Menge von Zeichen, disjunkt zu V , auch Terminalsymbole genannt
- R ist eine Menge von Regeln
- $S \in V$ ist die Startvariable

Regel: Besteht aus einer Variable und einer Kette von Variablen und Terminalsymbolen, geschrieben als: $A \rightarrow BCx$ (Rechts eine beliebige Folge von Variablen und Terminalsymbolen)

Beispiele

$$L = \{ w \in \{(), ()\} / w \text{ gültige Klammerung} \}$$

$$L = \{0^n 1^n / n \geq 0\}$$

$$V = \{K\}$$

$$\Sigma = \{(), ()\}$$

$$S = K$$

$$R = \{K \rightarrow \epsilon, K \rightarrow KK, K \rightarrow (K)\}$$

$$V = \{Q\},$$

$$\Sigma = \{0, 1\}$$

$$S = Q$$

$$R = \{Q \rightarrow \epsilon, Q \rightarrow 0Q1\}$$

Grammatik für reguläre Operationen

Neue Startvariable S_0 notwendig. Variablen $V = V_1 \cup V_2 \cup \{S_0\}$, Regeln müssen geeignet erweitert werden:

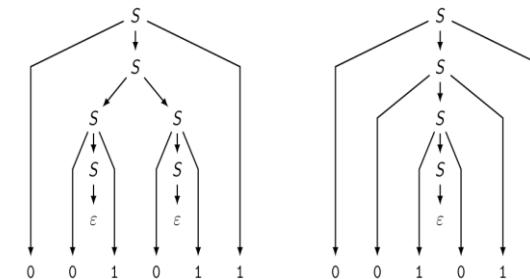
- **Alternative / OR:** Regeln für $L_1 L_2 \Rightarrow R = R_1 U R_2 U \{S_0 \rightarrow S_1, S_0 \rightarrow S_2\}$
- **Verkettung / AND:** Regeln für $L_1 L_2 \Rightarrow R = R_1 U R_2 U \{S_0 \rightarrow S_1 S_2\}$
- ***-Operation:** Regeln für $L_1^* \Rightarrow R = R_1 U \{S_0 \rightarrow S_0 S_1, S_0 \rightarrow \epsilon\}$

Parse-Tree

Zwei Ableitungen eines Wortes w einer kontextfreien Sprache L heißen äquivalent, wenn sie den gleichen Ableitungsbaum (Parse-Tree) haben. Hat eine Sprache, Wörter mit verschiedenen Parse-Trees heißt sie mehrdeutig.

Beispiel (Mehrdeutiger Parse Tree):

$$S \rightarrow 0S1 \mid 1S0 \mid SS \mid \epsilon$$



Chomsky-Normalform

Eine Regel ist in der Chomsky-Normalform, wenn S auf der rechten Seite nicht vorkommt und jede Regel von der Form $A \rightarrow BC$ oder $A \rightarrow a$ ist. Zusätzlich ist noch die Regel $S \rightarrow \epsilon$ erlaubt.

Umwandlung

1. Neue Startvariable $S_0 \rightarrow S$

$$2. \epsilon\text{-Regeln: } \begin{cases} A \rightarrow \epsilon \\ B \rightarrow AC \end{cases} \Rightarrow A \text{ kann weggelassen werden} \Rightarrow \begin{cases} B \rightarrow AC \\ \rightarrow AC \end{cases}$$

$$3. \text{Unit-Rules: } \begin{cases} A \rightarrow B \\ B \rightarrow CD \end{cases} \Rightarrow \text{aus } A \text{ kann man wie aus } B \text{ auch } CD \text{ machen} \Rightarrow \begin{cases} A \rightarrow CD \\ B \rightarrow CD \end{cases}$$

4. Verkettungen: $A \rightarrow u_1 u_2 \dots u_n$ ersetzen durch

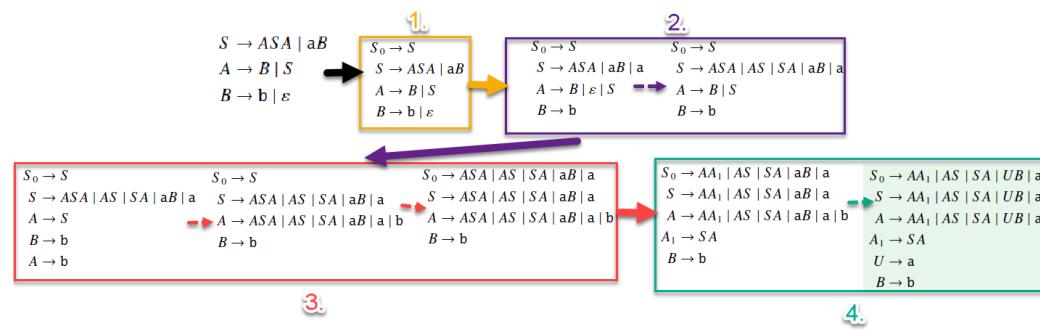
$$A \rightarrow u_1 A_1, A_1 \rightarrow u_2 A_2, \dots, A_{n-2} \rightarrow u_{n-1} u_n \text{ und falls } u_i \text{ ein Terminalsymbol ist: } A_{i-1} \rightarrow U_i A_i, U_i \rightarrow u_i.$$

Damit kann die Länge der Ableitung eines Wortes bestimmt werden.

Jedes Wort kann in $2^* |w| - 1$ Regelanwendungen gebildet werden.

⇒ Höchstens $|w| - 1$ Anwendungen von $A \rightarrow BC$, $|w|$ Anwendungen von $A \rightarrow a$

Beispiel



CYK-Algorithmus (Cocke-Younger-Kasami)

Mit dem CYK-Algorithmus kann ermittelt werden ob Wort zur kontextfreien Sprache gehört.

Dauer: $|w|^3 \rightarrow O(n^3)$

Gegeben:

- Grammatik $G = (V, \Sigma, R, S)$ in Chomsky-Normalform
- Variablen $A \in V$
- Wort $W \in \Sigma^*$

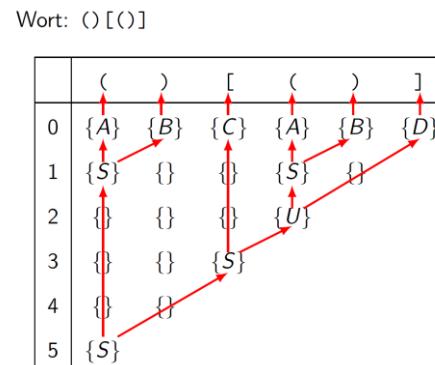
Frage: Ist w aus A ableitbar?

Spezialfälle: $w = \epsilon$ und $|w| = 1 \Rightarrow w$ ist ein Terminalsymbol

Falls $|w| > 1$:

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

Beispiel



Hinweis: Ergibt auch den Parse-Tree des Wortes.

Pumping Lemma für kontextfreie Sprachen

Es gibt auch Sprachen die nicht kontextfrei sind. Z.B. die Sprache: $L = \{a^n b^n c^n \mid n \geq 0\}$

Auch dafür kann man Pumping Lemma verwenden als Beweis, muss aber die Grammatik dazu nehmen.

Pumping Lemma für CFL

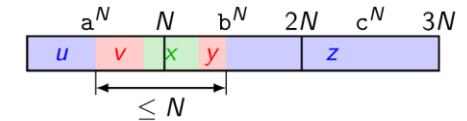
Ist L eine CFL, dann gibt es eine Zahl N , die Pumping Length, derart, dass jedes Wort $w \in L$ mit $|w| \geq N$ zerlegt werden kann in fünf Teile $w = uvxyz$ derart, dass

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

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 ⇒ $uv^kxy^kz \notin L \forall k \neq 1$
6. Widerspruch: L nicht kontextfrei

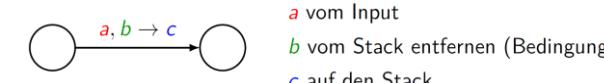
Stackautomat / Push Down Automat (PDA)

Stackautomat kann nur kontextfreie Grammatik erkennen. Er ist immer NICHT deterministisch.

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

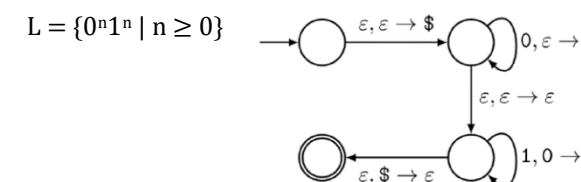
Zeichen	Bezeichnung	Beispiel
Q	Zustände	q0, q1, ...
Σ	Eingabe-Alphabet	{0,1}
Γ	Stack-Alphabet	{0,1,\$}
δ	$Q \times \Sigma \times \Gamma \rightarrow P(Q \times \Gamma)$	$\epsilon \rightarrow \$$
S	Startzustand	q0
F	Akzeptierzustände	q3

Übergänge:



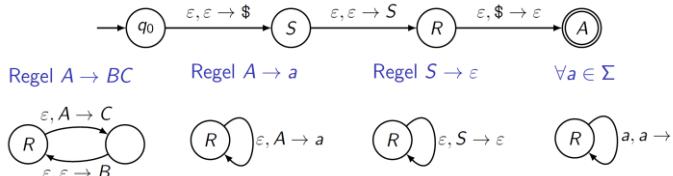
Verarbeite Input a und ersetze b Auf dem Stack durch c

Beispiel



Umwandlung Grammatik zu Stackautomat

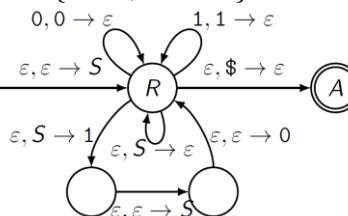
Grundgerüst



Beispiel

$L = \{0^n 1^n \mid n \geq 0\}$

$R = \{S \rightarrow \epsilon, S \rightarrow 0S1\}$



Umwandlung Stackautomat zu Grammatik

Ist P ein Stackautomat, dann gibt es eine Grammatik G, die die gleiche Sprache produziert.

Der Stackautomat muss zuerst normalisiert werden:

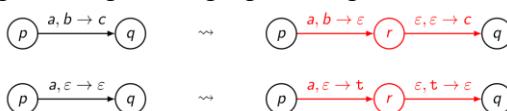
- Stackautomat darf nur ein Akzeptierzustand haben



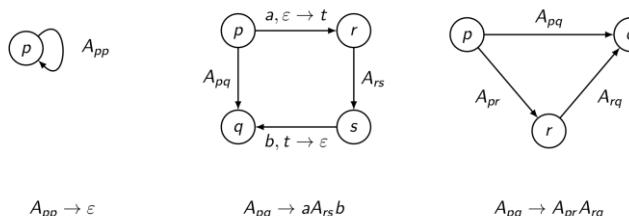
- Vor akzeptieren muss Stack geleert werden. Wird mit zusätzlichem Zeichen gelöst (Neuer Startzustand und nochmals neuer Akzeptierzustand)



- Jeder Übergang legt ein Zeichen auf dem Stack oder entfernt ein Zeichen, nicht gleichzeitig. ϵ -Übergänge benötigen weiteren Zwischenzustand mit beliebigen Zeichen.



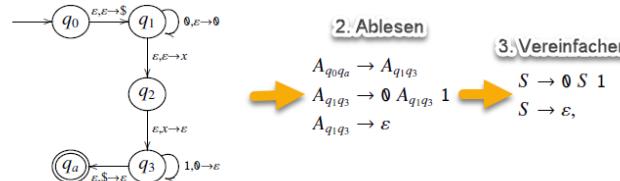
Anschliessend können die Regeln abgelesen werden:



Beispiel:

$L = \{0^n 1^n \mid n \geq 0\}$

1. Normalisierung Stackautomat



2. Ablesen

$A_{q_0 q_a} \rightarrow A_{q_1 q_3}$

$A_{q_1 q_3} \rightarrow \emptyset A_{q_1 q_3} 1$

$A_{q_1 q_3} \rightarrow \epsilon$

3. Vereinfachen

$S \rightarrow \emptyset S 1$

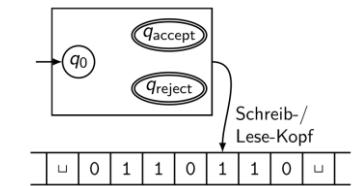
$S \rightarrow \epsilon$

Turing Maschine

Speicher ist ein unendliches Band mit mehreren Speicherzellen. Pro Zelle, ein Zeichen. Lesekopf liest/schreib Zeichen. Er muss sich immer entweder Links oder Rechts bewegen. Eine Sprache L heisst «Turing-Erkennbar», wenn es eine Turing-Maschine M gibt, die das Wort akzeptiert.

Turing Maschine $M = (Q, \Sigma, \Gamma, \delta, S, q_{accept}, q_{reject})$

Zeichen	Bezeichnung	Beispiel
Q	Zustände	q_0, q_1, \dots
Σ	Eingabe-Alphabet	$\{0, 1\}$
Γ	Stack-Alphabet	$\{0, 1, \sqcup\}$
δ	$Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$	$a \rightarrow b, R$
S	Startzustand	q_0
q_{accept}	Akzeptierzustand	
q_{reject}	Ablehnungszustand	



Übergang

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

Zustandsdiagramm

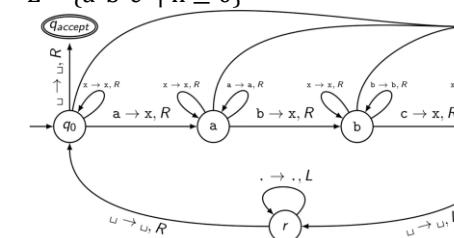
$$\delta(p, a) = (q, b, L): p \xrightarrow{a \rightarrow b, L} q$$

Ablauf Programm:

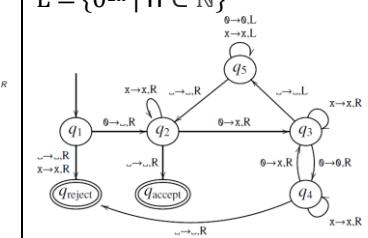
- Inputwort w auf Band schreiben
- Schreib-/Lesekopf auf erstes Zeichen positionieren (Meist \sqcup)
- Maschine starten, $t(w)$ Einzelschritte ausführen
- Maschine hält in q_{accept} oder q_{reject} an und akzeptiert oder verwirft Wort entsprechend

Beispiele

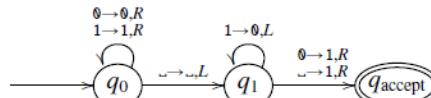
$L = \{a^n b^n c^n \mid n \geq 0\}$



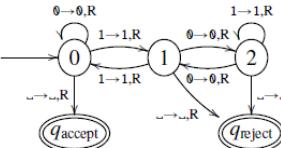
$L = \{0^{2n} \mid n \in \mathbb{N}\}$



Binärzahlen um 1 erhöhen



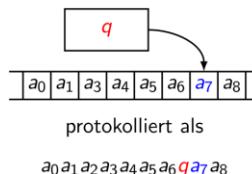
Binärzahl durch 3 teilbar (DEA in Turing umwandeln)



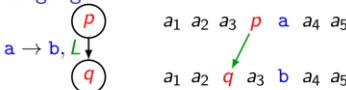
Berechnungsgeschichte

(Kann auch auf ein Ausfüllrätsel reduziert werden und somit beweisen das SAT NP-Vollständig ist.)

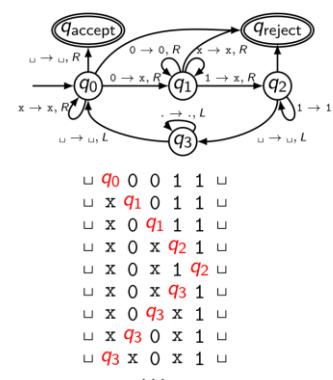
Notation für Maschinenzustand



Übergang



Protokoll einer Berechnung



Varianten von Turing-Maschinen

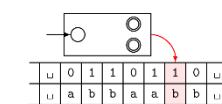
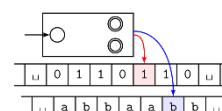
Jede Sprache, die von einer alternativen Turingmaschine erkannt werden kann, kann auch von einer einspurigen Touring-Maschine erkannt werden. Dauert einfach länger.

Anderes Bandalphabet

- Simuliert z.B. durch binäre Codierung
- $O(t(n))$

Mehrere Bänder (Ein Pointer pro Band)

- Simulierbar mit $2n$ Mehrspurigem Band, jedes Band bekommt eine zusätzliche Spur für Position. Die Position wird über ein Zeichen gelöst, welches verschoben wird, und die Position anzeigt
- $O(t(n)^2)$



Mehrspurige TM (ein Pointer)

- Simuliert, indem man die Spuren alle seriell hinterlegt
- $O(t(n))$

Nicht deterministische TM

- Bei jedem Übergang maximal N verschiedene Möglichkeiten
- Mehrere Berechnungswege möglich
- Akzeptiert sobald ein Weg zu q_{accept} führt
- Simulation: Maximal $N^{t(n)}$

- 3 Bänder: Arbeitspband, Kopie von w und Liste aller Folgen von Wahlmöglichkeiten

Aufzähler

- Eine TM mit einem Drucker bei dem gefundene Wörter «ausgedruckt» werden können. Kann endlos laufen.
- Wort kann mit «Ausdruck» verglichen werden um so akzeptiert zu werden
- Aufzählbare Sprache \Leftrightarrow Turing-erkennbare Sprache

Entscheidbarkeit

- Eine deterministische TM M heisst Entscheider, wenn sie auf jedem Input w anhält.
 - Eine nichtdeterministische TM M heisst Entscheider, wenn jede Berechnungsgeschichte terminiert
- \Rightarrow Entscheider = immer einen Definitiven Endzustand

Turing-erkennbare Sprache

- Die Sprache L heisst Turing-erkennbar, wenn es eine TM M gibt
- Kann evtl. auch unendlich lange laufen

Turing-entscheidbare Sprache

- Die Sprache L heisst Turing-entscheidbare Sprache, wenn es ein Entscheider M gibt

Ist $w \in L(M)$?

- Turing-erkennbare Sprache:
 - Wann weiss man, ob M nicht anhält oder einfach noch mehr Zeit benötigt?
- Turing-entscheidbare Sprache:
 - M wird auf Input w garantiert irgendwann anhalten

Entscheidbare Sprache vs. erkennbare Sprachen

Entscheidbare Sprache hält für jeden beliebigen Input an.

Erkennbare Sprache hält nur für akzeptierten Input an.

Entscheidbare Probleme

E = Leerheitsproblem, EQ = Gleichheitsproblem, A = Akzeptanzproblem

Problem	Wort	Bedingung	Entscheidbar	Entscheidungsalgorithmus
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}	$\langle G, w \rangle$	$w \in L(A)$	ja	Cocke-Younger-Kasami deterministischer Parse-Algorithmus
A_{TM}	$\langle M, w \rangle$	$w \in L(A)$	nein	Halteproblem

Satz von Rice

Große Aussage:

Ist P eine nichtriviale Eigenschaft Turing-erkennbarer Sprachen, dann ist $P_{TM} = \{ \langle M \rangle / L(M) \text{ hat Eigenschaft } P \}$ nicht entscheidbar

Voraussetzung:

keine Einschränkungen machen, wie Zahlengröße Reduzieren

Definition nichtrivial:

Eine Eigenschaft P Turing-erkennbarer Sprachen heißt nichtrivial, wenn es zwei Turing-Maschinen M1 und M2 gibt mit: L(M1) hat die Eigenschaft, L(M2) nicht.

Folgerung:

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

Lösungsanleitung einer Prüfungsfrage:

1. Nichtriviale Eigenschaft P aufzuschreiben
2. Die beiden Sprachen L(M1) und L(M2) bilden
3. Gibt es ein Programm, welches beide Sprachen erkennen kann? Sind beide Sprachen Turing erkennbar?
4. Dann besagt der Satz von Rice, dass die Sprache nicht entscheidbar ist

Beispiel

P_{PRIMES} = "Sprache besteht nur aus den Primzahlen"

$L_0 = \{42\}$ $L_1 = \{\text{Primzahlen}\}$ \Rightarrow Turing erkennbar

Satz von Rice greift da, so P_{PRIMES} nicht entscheidbar

Alternativ:

1. Reduktion auf Problem mit Erklärung
 - a. Halteproblem
 - b. Leerheitsproblem
 - c. Regularitätsproblem
2. Problem ist nicht entscheidbar
3. Satz von Rice als zusätzliche Begründung

Reduktion

Mittels Reduktion kann bewiesen werden, dass eine TM nicht entscheidbar ist. Dazu reduziert man eine TM auf eine nicht entscheidbare TM (E_{TM}).

Idee: Neues Problem schaffen, welches allenfalls einfacher zu lösen ist.

Berechenbare Abbildung:

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

Ist B entscheidbar, $f: A \leq B \rightarrow A$ entscheidbar

Ist B nicht entscheidbar, $f: A \leq B \rightarrow A$ nicht entscheidbar

Beispiel

Halteproblem

Theorem (Alan Turing)
 A_{TM} ist nicht entscheidbar.

Beweis.
 Konstruiere aus einem Entscheider H für

$$\text{HALT}_{\Sigma^*} = \{ \langle M \rangle \mid M \text{ ist eine Turingmaschine und } M \text{ hält auf leerem Band} \}$$

1. Lasse H auf Input $\langle M, \langle M \rangle \rangle$ ist nicht entscheidbar

2. Falls H akzeptiert: q_{accept}

3. Falls H verwirkt: q_{reject}

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

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

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

Widerspruch!

Spezielles Halteproblem

Das spezielle Halteproblem

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

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

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

Entscheider für A_{TM}

1. Konstruiere das Programm S

2. Wende H auf $\langle S \rangle$ an

ist nicht entscheidbar.

Regularitätsproblem: $A_{TM} \leq \text{REGULAR}_{TM}$

A_{TM} ist nicht entscheidbar
 Akzeptiert die Maschine M das Wort w?
 Es gibt keinen Entscheider

$\langle M, w \rangle$

Ist REGULAR_{TM} entscheidbar?
 Ist die Sprache $L(M)$ regulär?

Programm S mit Input u

1. $u \notin \{0^n 1^n \mid n \geq 0\} \rightarrow q_{reject}$

2. M auf u 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_{TM}

Wäre H ein Entscheider für REGULAR_{TM} , könnte man daraus einen

Entscheider für A_{TM} konstruieren

Komplexität

Eine Turingmaschine mit mehreren Bändern und Laufzeit $t(n)$ kann in Laufzeit $O(t(n)^2)$ auf einer Standard-Turing-Maschine simuliert werden.

Sei N aber eine Nichtdeterministische Touring-Maschine, dann kann diese in Laufzeit $2^{O(t(n))}$ simuliert werden. (Gilt für NTM die auch ein Entscheider sind.)

Polynomielle und Exponentielle Laufzeit

Warum ist polynomiell/exponentiell wichtig?

Annahme

Zykluszeit 1 ns

Laufzeit = Anzahl Schritte

n n^5 2^n

1 1ns 2ns

2 32ns 4ns

4 1.0/μs 16ns

8 32.7μs 256ns

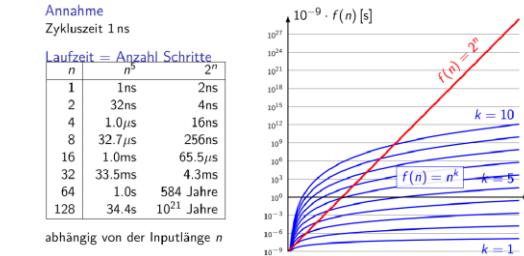
16 1.0ms 65.5μs

32 33.5ms 4.3μs

64 1.0s 584 Jahre

128 34.4s 10^{21} Jahre

abhängig von der Inputlänge n



Klassen P und NP

Die Klasse P besteht aus den Sprachen, die mit einem Entscheider (DTM) in polynomieller Laufzeit entschieden werden können. Ist eine Teilmenge von NP.

Die Klasse NP besteht aus Sprachen, die mit einer nichtdeterministischen Turingmaschine (NTM) in polynomieller Laufzeit entschieden werden können. Insbesondere enthält NP alle Sprachen mit einem polynominalen Verifizierer.

Verifizierer (Gleichbedeutend mit NP)

P, NP und polynomielle Verifizierer

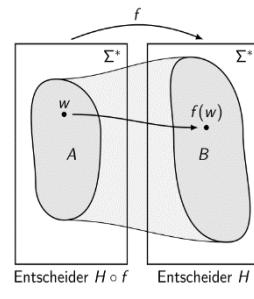
Verifizierer für: _____

Entscheidbar?

Zertifikat?

Verifikationsalgorithmus

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



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.

Beispiel Sudoku

Regeln:

Jede Zahl von 1-9 darf nur 1x vorkommen in jeder:

- Spalte
- Zeile
- Kleinem Quadrat

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

Verifizierer für: Sudoku

Entscheidbar?

Ja, alle $\leq (n^2)^{n^2 \times n^2}$ Möglichkeiten durchprobieren

Zertifikat? c = fehlende Ziffern

Verifikationsalgorithmus

Nr.	Was ist zu tun?	Aufwand
1	Für jedes Feld: Zeichen kommt auf der Zeile nicht mehr vor	$O(n^4 \cdot n^2)$
2	Für jedes Feld: Zeichen kommt in der Spalte nicht mehr vor	$O(n^4 \cdot n^2)$
3	Für jedes Feld: Zeichen kommt im Unterquadrat nicht mehr vor	$O(n^4 \cdot n^2)$
Total		$O(n^6)$

Reduktion

Dient dazu zwei Probleme zu vergleichen.

Polynomielle Reduktion

Polynomieller Laufzeit-Vergleich

Seien A und B entscheidbare Sprachen. Eine berechenbare Abbildung

$$f: \Sigma^* \rightarrow \Sigma^*: w \mapsto f(w)$$

mit

$$1. w \in A \Leftrightarrow f(w) \in B$$

$$2. f(w) \text{ ist berechenbar in polynomieller Zeit in } |w|$$

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

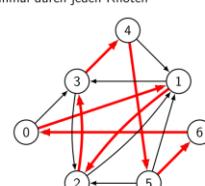
Satz
 $A \leq_p B$. Dann gilt
 $B \in P \Rightarrow A \in P, \quad A \notin P \Rightarrow B \notin P$
 $B \in NP \Rightarrow A \in NP, \quad A \notin NP \Rightarrow B \notin NP$

Beispiel
Stundenplan $\leq_p k$ -VERTEX-COLORING:
Zeitfenster \mapsto Farbe
Anzahl Zeitfenster $\mapsto k$
Fach \mapsto Vertext

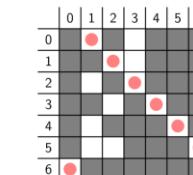
Anmeldung \mapsto Kante
Anmeldung eines Stunden auf zwei Fächer \mapsto Kante zwischen Vertext

Reduktion von HAMPATH

HAMPATH
Finde einen geschlossenen Pfad genau einmal durch jeden Knoten



Ausfüllrätsel
Platziere einen roten Stein genau einmal in jeder Zeile und Spalte



NP-Vollständig

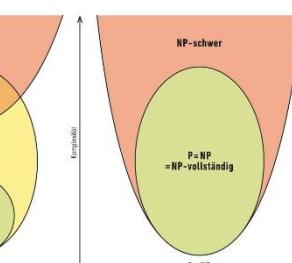
Eine Entscheidbare Sprache B heisst **NP-vollständig**, wenn sich jede Sprache A in NP polynomiell auf B reduzieren lässt. Sind die schwersten Probleme in NP (Katalog von Karp)

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
 $A \text{ NP-vollständig}$
 $B \in NP$
 $A \leq_p B$ $\Rightarrow B \text{ NP-vollständig}$

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



Beispiel NURIKABE

Nurikabe ist ein japanisches Logikrätsel, welches auf einem rechteckigen Gitter mit $u \times v$ Zellen gespielt wird. In einigen Feldern dieses Gitters sind Zahlen eingetragen, der Spieler muss einige der leeren Felder auswählen und schwarz färben. Die gefärbten Felder heissen "Fluss". Es entstehen so weisse Gebiete, bestehend aus weissen Feldern, die entlang einer Kante benachbart sind. Es müssen folgende Regeln beachtet werden:

- In jedem weissen Gebiet gibt es genau ein Zahlfeld, und die Zahl darin gibt an, wieviele weisse Felder das Gebiet umfasst.
- Es darf nur einen Fluss geben (das schwarze Gebiet muss zusammenhängend sein), und es darf keine schwarzen 2×2 -Gebiete geben (der Fluss hat keine Tümpel).

Die Abbildung zeigt ein korrekt gelöstes Nurikabe.



Verifizierer für: NURIKABE

Entscheidbar?

nr ur für die Anzahl Felder. Man kann alle 2^n möglichen Belegungen des Spielfelds mit schwarzen Feldern prüfen ob es die Regeln einhalten. Braucht exponentielle Zeit, aber ist Entscheidbar

Zertifikat?

Verifikationsalgorithmus

Verifikationsalgorithmus

Nr.	Was ist zu tun?	Aufwand
1	Für jedes Zahlfeld ($O(n)$) verwendet man einen Markieralgorimus, der alle zum Gebiet dieses Zahlfeldes gehörenden weissen Felder bestimmt ($O(n)$). Durchgänge mit Aufwand $O(n^3)$.	$O(n^4 \cdot n^2)$
2	Trifft dieser Algorimus auf ein weiteres Zahlfeld, ist der erste Teil von Regel 1 verletzt ($\rightarrow q_{reject}$).	$O(n)$
3	Weicht die Zahl der gefundenen weissen Felder vom Inhalt des Zahlfeldes ab, ist der zweite Teil von Regel 1 verletzt ($\rightarrow q_{reject}$).	$n \cdot O(n)$
4	Ebenfalls mit einem Markieralgorimus wird überprüft, ob das schwarze Gebiet zusammenhängend ist.	$O(n^2)$
5	Für jedes schwarze Feld wird überprüft, ob es Teil eines 2×2 -Tümpels ist.	$O(n)$
Total		$O(n^6)$

Katalog von Karp

Probleme mit K Zahlen:

- k-CLIQUE
- VERTEX-COLORING
- VERTEX-COVER
- FEEDBACK-*-SET
- SET-COVERING
- STEINER-TREE
- (MAX-CUT)
- SET-PACKING
- SEQUENCING

Aufteilung zwei Teilmengen:

- PARTITION
- MAX-CUT

Probleme mit Zahl 3:

- 3D-MATCHING
- 3SAT

Unterschiede HITTING-SET, EXACT-COVER:

- Hitting-Set: Menge von Punkten
- Exact-Cover: Menge von Teilmengen

Unterschiede FEEDBACK-*-SET:

- NODE \rightarrow Vertex entfernen
- ARC \rightarrow Kanten entfernen

Unterschiede SET-*:

- COVERING \rightarrow endliche Familie endlicher Mengen
- PACKING \rightarrow eine Familie von Menge

	SET-COVERING	SET-PACKING	EXACT-COVER
Anzahl Mengen	k	k	ganze Menge

Unterschiede *-COVER

Graph oder Mengen mit einer Zahl k

SAT (LOGIK)

3SAT (LOGIK)

Gibt es eine mögliche Lösung der Booleschen Gleichung, damit sie TRUE ergibt.

3SAT: Klausel besteht höchstens aus 3 Literalen pro Klausel und ist in KNF

$$F = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_1 \vee \bar{x}_2)$$

Klausel = Klammerausdruck, Literale = Variable

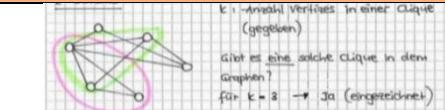
Bsp: Elektriker:

- n-Schalter \rightarrow n-Variablen
- m-Räume \rightarrow m-Klauseln
- Überall Licht \rightarrow Aussage = true
- Stromkreise \rightarrow Wertebereich (True, False)

Jedes Ausfüllrätsel ist mit SAT beschreibbar, Man kann Wahrheitstabellen bilden. NP-Vollständig ist jedes Problem, dass sich auf SAT reduzieren lässt.

k-CLIQUE

Gibt es k Knoten, die alle miteinander verbunden sind?



SET-PACKING

Geg: eine Familie $(S_i)_{i \in I}$ von Mengen und eine Zahl $k \in \mathbb{N}$

Gibt es eine k-Elementige Teilfamilie $(S_i)_{i \in J} \subset I$, d. $|J|=k$ derart, dass die Menge der Teilmengen paarweise disjunkt sind, also $S_i \cap S_j = \emptyset \quad \forall i, j \in J \text{ mit } i \neq j$

Gleich wie Exact-Cover nur Ausgangslage ist anders.

Einfach: Menge S und Menge T von Teilmengen. Gibt es k Teilmengen in T, welche disjunkt sind und die Menge S abdecken?

Beispiel 1 Küche: In einer Küche hat man eine Menge an Zutaten und ein Rezeptbuch voller Rezepte. Nun möchte man möglichst viele der Rezepte kochen, ohne eine Zutat mehrmals zu verwenden.

Beispiel 2 Medizinstudie: Für eine medizinische Studie ist eine grosse Zahl von Probanden rekrutiert worden. Sie sind bereits auf Allergien getestet worden, man weiss also von jedem Probanden, auf welche Allergene (Pollen, Katzenhaare, Hausstaub, Lactose, ...) er allergisch reagiert. Die Untersuchung soll sich auf eine Teilmenge von $k = 17$ oder noch mehr ausgewählten Allergenen beschränken, die so beschaffen ist, dass kein Proband auf mehr als eines der ausgewählten Allergene reagiert. Es stellt sich als schwierig heraus, eine solche Teilmenge zu finden. Warum?

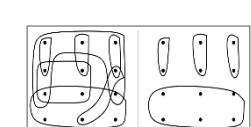
Allergene \leftrightarrow I

auf Allergen i allergische Probanden $\leftrightarrow S_i$

ausgewählte Allergene $\leftrightarrow J$

Ausschlussbedingung zwischen Allergenen i und j $\leftrightarrow S_i \cap S_j = \emptyset$

Es wird verlangt, k Allergene auszuwählen, also eine Teilmenge $J \subset I$ mit $|J| = k$ zu finden.



VERTEX-COVER

Geg: Ein Graph G und eine Zahl k

Gibt es eine Teilmenge von k Knoten so, dass jede Kante des Graphen ein Ende in dieser Teilmenge hat?

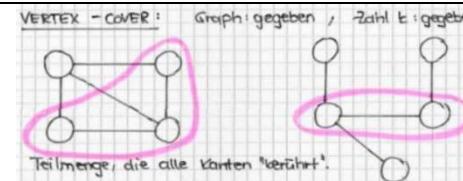
Bsp: Gibt es 4 Knoten, so dass jeder andere Knoten eine Kante zu diesem hat?

Bsp: Ein Verkehrsnetz soll regelmässig durch Mitarbeiter kontrolliert werden, die ihre Basis an einzelnen Knotenpunkten des Netzes haben. Kann man auf effiziente Art herausfinden, an welchen Knotenpunkten man Kontrolleure stationieren muss, damit jede Strecke in einem Konten mit Kontrolleur endet?

Knotenpunkte \leftrightarrow Knoten

Strecke \leftrightarrow Kante

Anzahl Kontrolleure \leftrightarrow k

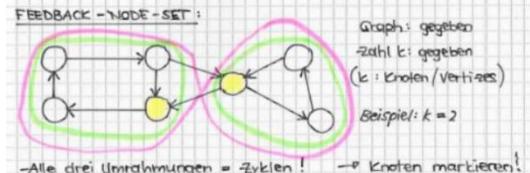


FEEDBACK-NODE-SET

Geg: Ein gerichteter Graph G und eine Zahl k

Gibt es eine endliche Teilmenge von k Vertizes/Knoten von G, so dass jeder Zyklus in G einen Vertex in der Teilmenge enthält.

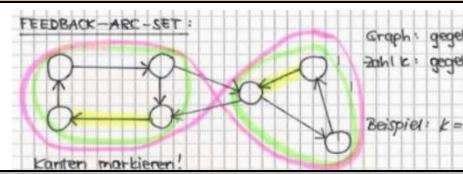
Es gibt mehrere Buslinien. Wo muss das Putzpersonal platziert werden, damit alle Linien geputzt werden können. Man möchte möglichst wenig Personal einsetzen.



FEEDBACK-ARC-SET

Geg: Ein gerichteter Graph G und eine Zahl k

Gibt es eine endliche Teilmenge von k Kanten von G, so dass jeder Zyklus in G eine Kante aus der Teilmenge enthält.



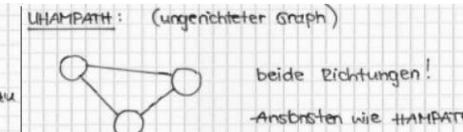
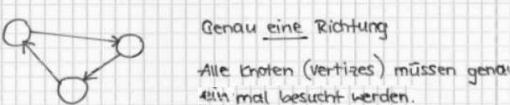
HAMPATH & UHAMPATH

Ein Hamilton-Pfad in einem gerichteten Graphen ist ein Pfad, der jeden Knoten genau einmal enthält. \rightarrow Haus des Nikolaus.

\rightarrow Komme zu jedem Knoten, jedoch nur einmal auf jeden Knoten.

Ungerichtet ist es der UHAMPATH. Bsp.: Ganze Schweiz bereisen ohne eine Stadt mehrmals besuchen

HAMPATH: (gerichteter Graph)



SET-COVERING

Geg: eine endliche Familie endlicher Mengen $(S_j)_{1 \leq j \leq n}$ und eine Zahl k

Gibt es eine Unterfamilie bestehend aus k Mengen, die die gleiche Vereinigung hat?

Einfach: Kann man k Teilmengen bilden, welche die Menge S komplett abdecken.

Bsp.: $k=3 \Rightarrow D \& E \& A$ oder B oder C

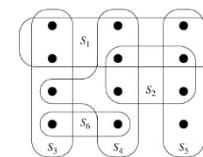
$A=\{1,3,5,11,19\}$

$B=\{1,3,7,9,17\}$

$C=\{1,3,21,29,27\}$

$D=\{1,5,21,27,29\}$

$E=\{1,9,11,21,31\}$



Beispiel Ältestenrat: Von einem uralten Planetensystem soll ein Ältestenrat gebildet werden. Jedoch kann nicht von jedem Stamm (wegen der Entfernung) ein Mitglied bestimmt werden. Darum wurde beschlossen das Mitglieder nur einen einzigen Tropfen Blut eines Stammes in sich haben muss um Mitglied zu werden (weit weit entfernter Vorfahre muss Mitglied eines Stammes gewesen sein).

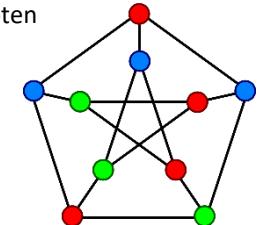
Zu jedem Bürger i ist die Menge S_i aller Stämme bekannt, von denen er Blut in sich trägt. Die Vereinigung aller S_i ist die Familie "U" aller Stämme. Gefragt wird jetzt nach k Bürgern sodass alle Stämme abgedeckt werden.

VERTEX-COLORING (Planungsprobleme)

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

Bsp: Job-Planung

- n Job \rightarrow Knoten,
- m gleiche Ressourcen \rightarrow Verbindungen
- parallele Jobs (bzw. Interval) \rightarrow gleiche Farbe
- N Intervalle \rightarrow n Farbe



EXACT-COVER

Geg: eine Familie $(S_j)_{1 \leq j \leq n}$ von Teilmengen einer Menge U

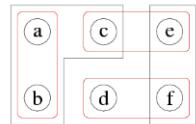
Gibt es eine Unterfamilie von Mengen, die disjunkt sind, aber die gleiche Vereinigung haben?

Die Unterfamilie $(S_{j_i})_{1 \leq i \leq m}$ muss also $S_{j_1} \cap S_{j_k} = \emptyset$ und $\cup_{j=1}^n S_j = \cup_{i=1}^m S_{j_i}$

Einfach: Jedes Element in U, soll genau in einer der Teilmengen in der Familie S vorkommen.

Die gesuchte Menge bildet eine exakte Überdeckung

$$U = \{a, b, c, d, e, f\}, S = \{\{a, b\}, \{a, b, c\}, \{c, e\}, \{d, f\}, \{e, f\}\}$$



Student Xaver Tecco soll im Rahmen einer Big-Data-Studienarbeit die Kunden einer grossen Shop-Website untersuchen und klassifizieren. Es steht eine grosse Zahl von binären Eigenschaften zur Verfügung, zum Beispiel ob Kunden ein bestimmtes Produkt gekauft haben, oder ob ein Kunde nur im Dezember einkauft. Herr Tecco soll herausfinden, ob es eine Teilmenge von Kriterien derart gibt, dass jeder Kunde genau eine der Eigenschaften hat. Die Abgabe der Arbeit steht in zwei Tagen bevor, und er hat noch keinen funktionierenden Algorithmus. Muss er sich Sorgen machen?

Eigenschaft \leftrightarrow Menge S_j

Teilmenge von Eigenschaften \leftrightarrow Unterfamilie S_j

Genau eine der Eigenschaften $\leftrightarrow S_{j_i} \cap S_{j_k} = \emptyset \forall i \neq k$

Alle Kunden erfasst $\leftrightarrow \cup_{j=1}^n S_j = \cup_{i=1}^m S_{j_i}$

3D-MATCHING

Geg: Endliche Menge T und eine Menge U von Tripeln TxTxT

Gibt es eine Teilmenge W von U so, dass $|W| = |T|$ und keine zwei Elemente von w stimmen in irgendeiner Koordinate überein?

Ein Koch hat je n Rezepte für Vorspeisen,

Hauptspeisen und Desserts. Nicht alle Vorspeisen lassen sich mit jeder Hauptspeise kombinieren, dasselbe gilt auch für Desserts. Damit jedes seiner Rezepte regelmäßig zum Einsatz kommt, möchte der Koch eine Folge von n Menüs zusammenstellen, so dass jedes Rezept in genau einem der Menüs vorkommt. Nach längerem

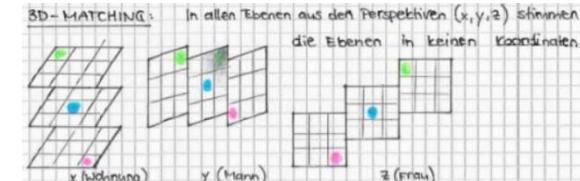
tüfteln gibt er jedoch frustriert auf. Können Sie erklären, warum ihm die Menugestaltung so schwer gefallen ist.

Nummern der Rezepte \leftrightarrow Menge T

Menuzusammenstellungen \leftrightarrow Tripel aus TxTxT

Menge der Möglichen Menuzusammenstellung \leftrightarrow Menge U

Gesuchte Menü's n \leftrightarrow Teilmenge W, so $n = |T| = |W|$



SUBSET-SUM (Rucksack-Problem)

Geg: Menge S von ganzen Zahlen

Kann man eine Teilmenge finden, die als Summe einen bestimmten Wert t hat?

Bsp. Rucksack-Problem

Grösse der Gegenstände → Zahl

Menge der Gegenstände → Menge S der Zahlen

Grösse des Rucksacks → Wert t (Summe)

Gegenstände in Rucksack → Teilmenge

SEQUENCING

Geg: ein Vektor $\in \mathbb{Z}^p$ von Laufzeiten von p Jobs, ein Vektor $\in \mathbb{Z}^p$ von spätesten Ausführzeiten, ein Strafenvektor $\in \mathbb{Z}^p$ und eine positive Zahl k

Gibt es eine Permutation der Zahlen 1,...,p, so dass die Gesamtstrafe für verspätete

Ausführung bei der Ausführung der Jobs nicht grösser ist als k?

Einfach: Geg.: eine Menge an Jobs, pro Job eine Ausführungszeit, Deadline und eine Strafe. Sowie eine maximale Strafe von k. Die Jobs müssen sequenziell abgearbeitet werden. Wird ein Job zu spät fertig, muss eine Strafe gezahlt werden.

Ges.: Eine Reihenfolge von Jobs so, dass die Strafe kleiner gleich k ist.

Eine Firma hat eine bestimmte Anzahl laufende Verträge. Der Firma ist es nicht möglich alle Verträge in einer bestimmten Zeit abzuarbeiten. Sie versucht also Schadensbegrenzung zu machen, indem sie möglichst viele Verträge in der verbleibenden Zeit abarbeitet die eine hohe Strafe zur Folge haben.

PARTITION

Geg: Eine Folge von S ganzen Zahlen c_1, c_2, \dots, c_s

Kann man die Indizes 1,2,...,S in zwei Teilmengen I und \bar{I} teilen, so dass die Summe der zugehörigen Zahlen c_i identisch ist: $\sum_{i \in I} c_i = \sum_{i \in \bar{I}} c_i$

Einfach: Gibt es zwei disjunkte Teilmengen mit der gleichen Summe?

Bsp: eine Reihe von Wassergläsern unterschiedlich gefüllt. Es sollen 2 Behälter gleich viel mit den Gläsern gefüllt werden. Welche Gläser müssen in welche Behälter geleert werden.

$$A: \{1, 2, 3, 5, 6, 8, 9\}$$

$$B = A - B = \{1, 3, 5, 8\} \Rightarrow 1+3+5+8 = 17$$

$$C = A - C = \{2, 6, 9\} \Rightarrow 2+6+9 = 17$$

$$B \cup C = A$$

MAX-CUT

Geg.: ein Graph G mit einer Gewichtsfunktion $w:E \rightarrow \mathbb{Z}$ und eine Zahl W

Gibt es eine Teilmenge S der Vertizes, so dass das Gesamtgewicht der Kanten, die S mit seinem Komplement verbinden, mindestens so gross ist wie W.

$$\sum_{\{(u,v) \in E \wedge u \in S \wedge v \in \bar{S}\}} (w(\{u, v\})) \geq W$$

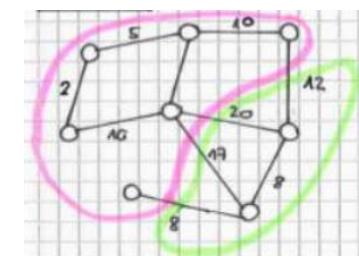
Einfach: Der Max-Cut eines Graphen ist eine Zerlegung seiner Knotenmenge in zwei Teilmengen, so dass das Gesamtgewicht der zwischen den beiden Teilen verlaufenden Kanten mindestens W wird.

Feindliche Übernahme einer Firma, mit resultierender Aufteilung der Abteilung, dass diese möglichst ineffizient miteinander kommunizieren können.

Abteilung ↔ Vertex

Kommunikationsbeziehung ↔ Kante

Kommunikationsvolumen ↔ Gewicht einer Kante



HITTING-SET

Geg: eine Menge von Teilmengen $S_i \subset S$

Gibt es eine Menge H, die jede Menge in genau einem Punkt trifft, also $|H \cap S_i| = 1$

$$\text{Geg.: } A = \{1, 2, 3\}, B = \{1, 2, 4\}, C = \{1, 2, 5\}$$

$$\text{Ges.: } H = \{3, 4, 5\}$$

STEINER-TREE

Geg: ein Graph G, eine Teilmenge R von Knoten und eine Gewichtsfunktion und eine positive Zahl k.

Gibt es einen Baum mit Gewicht $\leq k$, dessen Knoten in R enthalten sind? Das Gewicht des Baumes ist die Summe der Gewichte über alle Kanten im Baum.

Beispiel: Bau einer Zugstrecke oder Stromnetzes:

Stromnetz/Zugstrecke ↔ STEINER-TREE

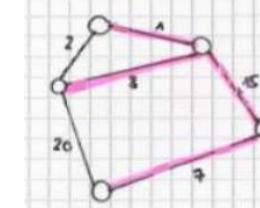
Ortschaften ↔ Knoten

zu erschliessende Ortschaften ↔ Knoten in R

Baukosten ↔ Gewicht w einer Kante

Budget ↔ maximales Gewicht k

Gewicht = 26 :

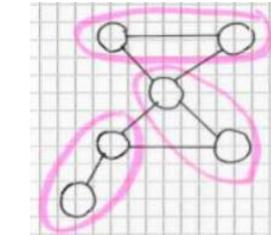


CLIQUE-COVER (EXOR mit bestimmter Zahl k)

Geg: ein Graph G und eine Positive Zahl k

Gibt es k Cliques so, dass jede Ecke in genau einer der Cliques ist?

Bsp.: Für eine Gruppenarbeit sollen k Gruppen gebildet werden. Um die Zeit für das Kennenlernen möglichst kurz zu halten, sollen sich die Leute einer Gruppe bereits kennen. Alle Leute sollen beschäftigt sein.



Teilnehmer ↔ Knoten

Kennen sich ↔ Kante

Anzahl Gruppe ↔ k

Gruppe ↔ Clique

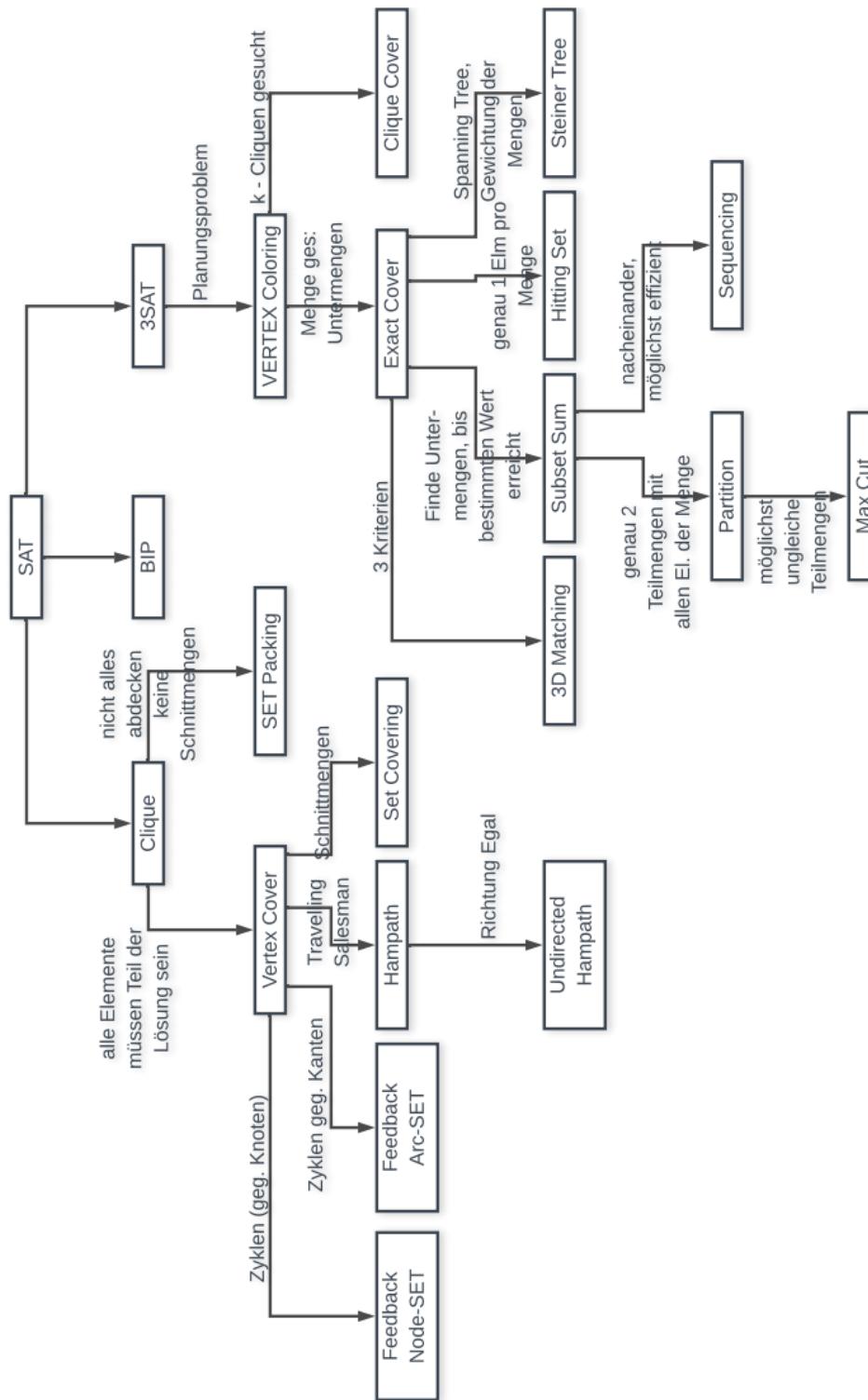
BIP (binary integer programming)

Zu einer ganzzahligen Matrix C und einem ganzzahligen Vektor d, ist ein binärer Vektor x zu finden mit $C^*x = d$

$$\begin{bmatrix} 1 & 3 & 0 \\ 0 & 2 & 5 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \end{bmatrix}$$

Vorgehen bei Prüfungen:

1. Problem aus Liste suchen (Dabei Punkte beachten, wie: Ist es ein Mengenproblem, Planungsproblem, kommen k-Zahlen vor, Teilmengen?)
2. Reduktion des Problems auf Karp mit Pfeil-Liste darstellen (Eigenschaften)
3. Zusammenfassung Schreiben: «Problem x ist äquivalent mit Karp-Katalog-Problem x und deshalb gehört es zu den NP-Vollständigen Problemen und daher ist es nicht einfach entscheidbar (Es gibt keinen polynominalen Algorithmus derzeit).»



Turing-Vollstndigkeit

Turing-Maschine und moderne Computer haben einige Gemeinsamkeiten. Er erfüllt alle Eigenschaften einer Turing-Maschine (Zustände, Band, Schreib-/Lesekopf/Anhalten). Einzig kann er mehrere Programme ausführen und eine Turing-Maschine ist Problemspezifisch.

Ist sie das wirklich? NEIN, es gibt die Universelle Turing-Maschine mit.

- ...eigenes Band für Codierung der Übergansfunktionen
 - ...eigenes Band für aktuellen Zustand
 - ...Arbeitsband

Eine solche Mehrbandmaschine kann auch auf einer Standard-Turingmaschine simuliert werden.

Es gibt aber Komponenten die ein PC hat, aber eine Touring-Maschine nicht. Doch sind diese wesentlich (Verändern Fähigkeiten einer TM nicht-polynomiell)? NEIN sind sie nicht:

Persistenter Speicher: Files nicht unterscheidbar von Daten, für TM egal

Interaktion: Lösung eines spezifizierten Problem braucht keine Interaktion

Input/Output: Output ist nicht wesentlich. Input «existiert nicht». (Wenn TM angehalten wird, kann Kernel Band «manipulieren» und TM sieht nun geänderter Input, weiss aber nicht ob es nun selbst gemacht hat oder von aussen kommen. Also nicht wesentlich.

Vergleich von Maschinen

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

$M_2 \leq M_1$ (M_2 ist simulierbar auf M_1)

Programmiersprachen

Frage: Ist eine Programmiersprache Turing-Vollständig?

⇒ Turing-Vollständig bedeutet das Programme ebenso mit Halteproblem, Rice und NP konfrontiert sind. So kann man nicht sagen ob ein Programm anhält, akzeptabler Input eine bestimmte Eigenschaft hat, ob Spezifikationen erfüllt sind und NP-Vollständige Probleme haben exponentielle Laufzeit.

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?

Beispiele für Turing-Vollständige Sprachen:

- Maschinennahe Programmiersprache C
 - *Javascript* ist ebenso Turing-Vollständig weil es ein Programm gibt in JS, dass Linux und auch den C-Compiler simulieren kann und dann ausführen.
 - *L^AT_EX* und *XSLT* ebenso Turing-Vollständige Sprachen.

LOOP

Ist nicht Turing-Vollständig, da Programme immer terminieren. (Lässt sich beweisen mit vollständiger Induktion)

Grundelemente für Sprachen

- Konstanten c mit natürlichen Werten $0, 1, \dots, 1291, \dots$
- Variablen x_0, x_1, \dots 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

Kontrollstruktur

Führe ein Programm P genau x_i mal aus:

```
LOOP xi DO  
    P  
END
```

Bedingte Anweisung

```
IF xi THEN  
    P  
END  
implementieren als  
y := 0  
LOOP xi DO y := 1 END  
LOOP y DO P END
```

Addition/Subtraktion

$x_i := x_j \pm x_k$
kann implementiert werden als

```
xi := xj  
LOOP xk DO  
    xi := xi ± 1  
END
```

bzw.
 $x_i := 0$
LOOP x_k DO
 LOOP x_j DO
 x_i := x_i + 1
 END
END

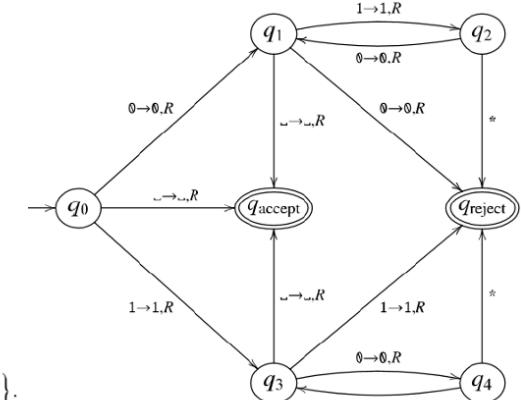
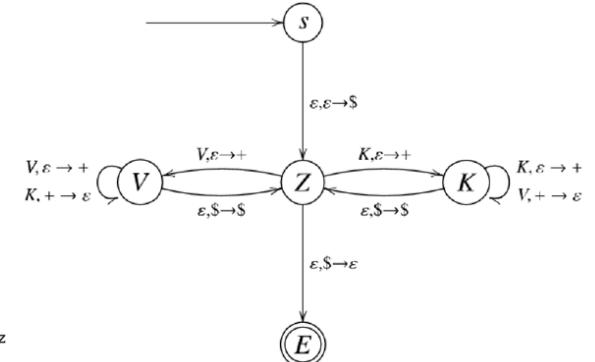
Multiplikation

$x_i := x_j * x_k$
kann implementiert werden als

```
xi := 0  
LOOP xk DO  
    xi := xi + xj  
END
```

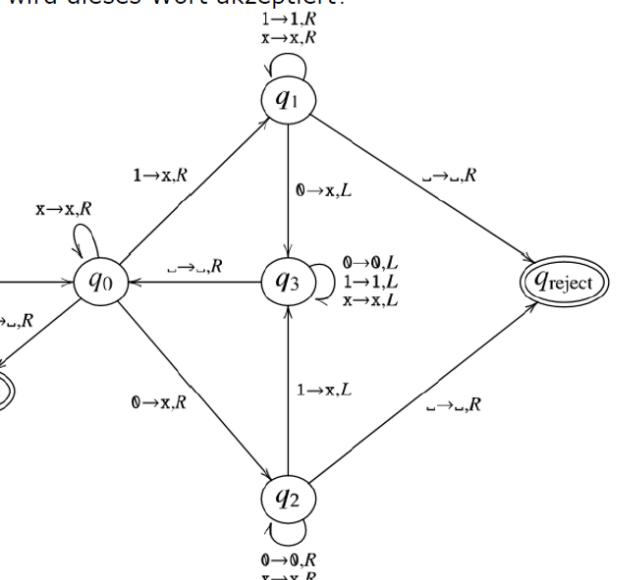
Diverse alte Aufgaben

Stackautomaten und Turing-Maschine



$$L = \left\{ w \in \Sigma^* \mid w \text{ besteht abwechselnd aus } 0 \text{ und } 1 \text{ und hört mit } \right. \\ \left. \text{dem gleichen Zeichen auf wie es beginnt} \right\}.$$

Verarbeite das Wort 101, bzw wird dieses Wort akzeptiert?



WHILE und GOTO

Turing-Vollständig und Äquivalenz

WHILE

Definition der Sprache WHILE

- Grundelemente wie in LOOP
- "Bedingte" Schleifenkonstruktion:
LOOP x_i DO P END
- WHILE $x_i > 0$ DO P END
- für 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
 $y := y - 1$

LOOP in WHILE

- Die Schleifenkonstruktion von LOOP
- LOOP x_i DO P END
- kann implementiert werden in WHILE

GOTO

Definition der Sprache GOTO

- Grundelemente wie in LOOP
- Markierte Folge von Anweisungen

$M_1 : A_1$
 $M_2 : A_2$
...
 $M_k : A_k$

Unbedingte Anweisung

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

abgekürzt:

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

Implementation von GOTO in WHILE

Übersetzung

Ein WHILE-Konstrukt

WHILE $x_i > 0$ DO P END

wird übersetzt in ein GOTO-Code-Segment

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

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

Implementation von GOTO in WHILE

GOTO-Programm:	Modifizierte Anweisung A'_i
$M_1 : A_1$	► A_i eine bedingte Sprunganweisung
$M_2 : A_2$	$M_i : \text{IF } x_i = c \text{ THEN GOTO } M_j$
... $M_k : A_k$	wird übersetzt in A'_i :
implementiert in WHILE:	IF $x_i = c$ THEN $z := j$ ELSE $z := z + 1$ END
$z := 1$	► A_i eine gewöhnliche Anweisung
WHILE $z > 0$ DO	
IF $z = 1$ THEN A'_1 END	$M_l : A_l$
IF $z = 2$ THEN A'_2 END	wird übersetzt in A'_i :
: IF $z = k$ THEN A'_k END	$A_i ; z = z + 1;$
IF $z = k + 1$ THEN $z := 0$ END	

Brainfuck und Ook Ook

BrainFuck

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

Ook Ook

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

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

} ⇒ Turing-vollständig

Eine sehr primitive Sprache reicht aus für Turing-Vollständigkeit.(Auch PowerPoint ist Turing-Vollständig)

BrainFuck

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

Ook Ook

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

Pumping Lemma Kontextfrei

In der Vorlesung wurde gezeigt, dass die Sprache $\{a^n b^n c^n \mid n \geq 0\}$ über dem Alphabet $\Sigma = \{a, b, c\}$ nicht kontextfrei ist. Wenn man die Bedingungen etwas lockert, und nur noch verlangt, dass die Anzahl der verschiedenen Zeichen übereinstimmt, die Reihenfolge aber beliebig sein darf, erhält man die Sprache

$$L = \{w \in \Sigma^* \mid |w|_a = |w|_b = |w|_c\}.$$

Ist L kontextfrei?

Lösung. Nein, auch L ist nicht kontextfrei. Man kann dies mit dem Pumping-Lemma für kontextfreie Sprachen nachweisen, wobei man sogar das gleiche Beispielwort verwenden kann wie im Falle der Sprache $\{a^n b^n c^n \mid n \geq 0\}$.

- ❶ Man nimmt dazu an, dass L kontextfrei sei. ❷ Gemäß Pumping-Lemma gibt es daher die Pumping-Length N . ❸ Wir konstruieren jetzt ein Wort $w = a^N b^N c^N$, welches die Voraussetzungen des Pumping-Lemma sicher erfüllt. ❹ Es muss also eine Unterteilung $w = uvxyz$ geben, so dass $|vxy| \leq N$ ist. ❺ Diese letzte Bedingung hat zur Folge, dass v und y höchstens in zwei der drei Blöcke a^N, b^N oder c^N liegen können. Insbesondere ändert sich beim Pumpen nur die Anzahl von zwei der drei Buchstaben, nach auf- oder abpumpen erhält man also ein Wort, bei dem die Anzahl jedes der drei Buchstaben nicht mehr gleich ist, also kein Wort mehr aus L , im Widerspruch zur Aussage des Pumping-Lemma. ❻ Daher kann L nicht kontextfrei sein. \circ

Turing-Maschine für binäres Additionsprogramm

Konstruieren Sie ein binäres Additionsprogramm für eine geeignet ausgebauten Turing-Maschine, welches Laufzeit $O(n)$ hat, wobei n die Anzahl Stellen der Summanden ist. Warum ist Ihre Maschine so viel schneller als die in der Vorlesung im Video gezeigte Maschine?

Lösung. Wir verwenden eine Maschine mit drei Bändern. Zunächst schreiben wir den zweiten Summanden auf das zweite Band, was in $O(n)$ Schritten durchgeführt werden kann. Dann beginnt die eigentliche Addition, wir lesen den ersten Summanden vom ersten Band, beginnend beim niederwertigsten Bit, und den zweiten Summanden vom zweiten Band, ebenfalls beginnend beim niederwertigsten Bit. Die Summe wird jeweils auf Band 3 geschrieben. Nach $O(n)$ solchen Operationen steht das Resultat auf Band 3.

Nach einem in der Vorlesung bewiesenen Satz benötigt die Simulation dieser Maschine mit drei Bändern auf einer Standardmaschine die Laufzeit $O(n^2)$, was sich mit der Laufzeit der Maschine aus der Vorlesung deckt. \circ

Logik-Rätsel Fillomino

Das Rätselspiel *Fillomino* wird auf einem $n \times m$ Spielfeld gespielt. In einzelnen Zellen des Spielfeldes sind Zahlen eingetragen. Der Spieler muss die leeren Felder mit Zahlen füllen, so dass zusammenhängende Gebiete soviele Felder enthalten wie die Zahl angibt. Ein zusammenhängendes Gebiet besteht aus Feldern, die sich entlang einer Kante berühren und alle die gleiche Zahl enthalten. Die Abbildung zeigt ein Fillomino-Rätsel (links) mit Lösung (rechts).

3	1		4	1	2
				6	
4	4	1			
		4		1	
			2		
6		1	3		1

3	1	4	4	1	2
3	3	4	4	6	2
4	4	1	6	6	6
6	4	4	2	1	6
6	6	6	2	3	6
6	6	1	3	3	1

Verifizierer

1. Überprüfe, ob jede Zelle mit einer Zahl gefüllt ist.
2. Für jede Zelle des Spielfeldes prüfe wie folgt, ob die Zahl der Zellen des zusammenhängenden Gebietes der Zelle die richtige Anzahl Zellen enthält
 - (a) Markiere die Startzelle
 - (b) Gehe durch das Spielfeld und markiere alle Zellen, die zu bereits markierten Zellen benachbart sind, und die die gleiche Zahl enthalten wie die Startzelle.
 - (c) Wiederhole 2b bis sich nichts mehr ändert
 - (d) Zähle die markierten Zellen, brich mit q_{reject} ab, wenn die Zahl der markierten Zellen nicht mit der Zahl in der Startzelle übereinstimmt.

Es muss jetzt nur noch gezeigt werden, dass der Verifizierer tatsächlich in polynomieller Zeit fertig wird. Die einzelnen Schritte des Verifizierers brauchen Zeit wie folgt

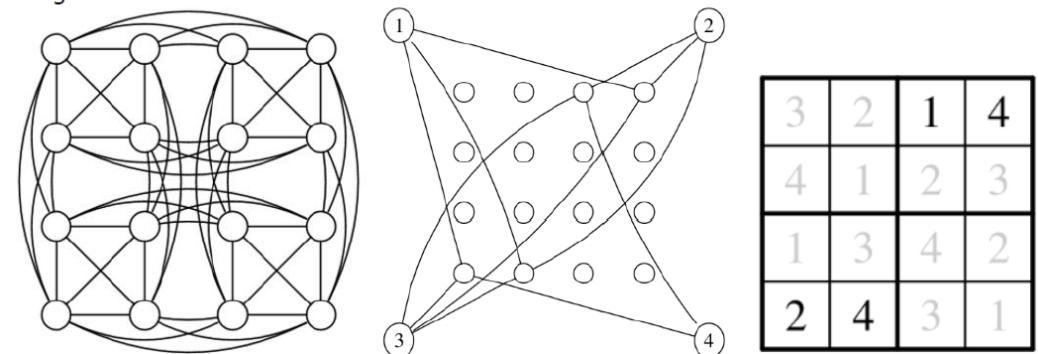
1. Zellen prüfen: $O(mn)$
2. In diesem Schritt werden mn mal die folgenden Schritte ausgeführt:
 - (a) Startzelle markieren: $O(1)$
 - (b) Nachbarzellen markieren: $O(mn)$
 - (c) Abbruch falls keine Änderung: $O(1)$
 - (d) Markierte Zellen zählen: $O(mn)$

Insgesamt ist also die Verifikation in maximal $mnO(mn) = O(m^2n^2)$ möglich, insbesondere ist die Arbeit polynomiell in der Größe des Spielfeldes. Damit ist gezeigt, dass es einen polynomiellen Verifizierer gibt, also kann eine nichtdeterministische Maschine das Problem in polynomieller Zeit lösen. \circ

Reduktion Sudoku-Vertex-Coloring

Jedes Feld \leftrightarrow ein Vertex im Graphen
nicht gleiche Zahlen \leftrightarrow Kante

Vorgabefelder \leftrightarrow zusätzliche Vertices



Entscheidbarkeit (Reduktion Halteproblem, RICE)

6.21. Ist es möglich, ein Programm zu schreiben, welches von einem anderen Programm herausfinden kann, ob es eine bestimmte Speicherzelle verändern wird, ohne es laufen zu lassen?

Lösung. Es geht darum, ein Programm zu schreiben, welches durch Inspektion einer Turing-Maschine herausfindet, ob ein bestimmtes Feld des Bandes während der Laufzeit irgendwann verändert wird.

Gäbe es nämlich so einen Entscheider E , könnte man damit das Halteproblem entscheiden. Dazu sei M irgend eine Turing-Maschine, von der wir wissen wollen, ob sie anhalten wird oder nicht. Wir konstruieren jetzt eine neue Turing-Maschine mit zwei Bändern. Auf dem ersten Band operiert die Turing-Maschine wie gewohnt. Die Zustände q_{accept} und q_{reject} werden aber leicht modifiziert: wenn die ursprüngliche Turingmaschine in einen dieser Zustände gehen will, lassen wir die modifizierte Turingmaschine erst noch ein Zeichen in das erste Feld des zweiten Bandes schreiben. Dies modifiziert Turingmaschine übersetzen wir jetzt noch in eine Turingmaschine mit einem Band (Standardalgorithmus). Wir haben jetzt eine neue Turingmaschine, die genau dann ein ganz bestimmtes Feld verändern wird, wenn die ursprüngliche Turingmaschine anhält. Wir haben also eine Reduktion des Halteproblems auf das Problem gefunden zu entscheiden, ob ein bestimmtes Feld überschrieben wird. Da E ein Entscheider für dieses Problem ist, wäre auch das Halteproblem entscheidbar. Dieser Widerspruch zeigt, dass es einen solchen Entscheider nicht geben kann.

Diese Frage lässt sich allerdings nicht mit dem Satz von Rice behandeln. Dieser ist nur anwendbar auf die Sprache, die von einer TM akzeptiert wird. Im vorliegenden Fall interessiert die Sprache, die die TM akzeptiert, aber gar nicht. Es interessiert nur die Implementationseigenschaft, ob eine bestimmte Speicherzelle verändert würde. Dabei ist es durchaus denkbar, dass es für die gleiche Sprache L zwei verschiedene TMs M_1 und M_2 gibt, die beide die Sprache L akzeptieren, aber M_1 modifiziert die besagte Speicherstelle, M_2 jedoch nicht. \circlearrowright

6.22. Ein e-Learning-System soll Schülern arithmetische Ausdrücke zur Auswertung geben, und die Antworten der Schüler überprüfen. Die Qualitätssicherung verlangt vom Programmierer dieses Moduls, dass er einen unabhängigen Test schreibt, welcher aus dem Source-Code des Moduls ableiten kann, ob je ein inkorrekt arithmetischer Ausdruck als Aufgabe gestellt werden könnte. Kann der Programmierer dieses Problem lösen?

Lösung. Das Modul soll nur Wörter einer Sprache von korrekten arithmetischen Ausdrücken produzieren. Die vom Modul akzeptierte Sprache soll also die Eigenschaft

$$P = \text{"enthält nur korrekte arithmetische Ausdrücke"}$$

haben. Diese Eigenschaft ist nicht trivial. Die Sprache

$$L_1 = \{w \mid w \text{ ist ein korrekter arithmetischer Ausdruck}\}$$

ist Turing erkennbar (sogar Turing-entscheidbar, dank des CYK-Algorithmus), und hat die Eigenschaft P . Die Sprache

$$L_2 = \{"7-\text{"}\}$$

ist als endliche und damit reguläre Sprache ebenfalls Turing-erkennbar, hat aber die Eigenschaft P nicht. Nach dem Satz von Rice folgt daher, dass nicht entscheidbar ist, ob die von dem Modul akzeptierte Sprache von Wörtern nur aus korrekten arithmetischen Ausdrücken besteht. So einen Test kann der Programmierer also nicht schreiben. \circlearrowright

Sprachproblem

6.24. Formulieren Sie die folgenden Probleme als Sprachprobleme

- Welche natürlichen Zahlen sind Quadrate einer natürlichen Zahl?
- Falls $n \in \mathbb{N}$ eine Quadratzahl ist, finde man die Wurzel.
- Hat die quadratische Gleichung $ax^2 + bx + c = 0$ mit $a, b, c \in \mathbb{N}$ reelle Lösungen?

Lösung. a) Sei L die Sprache über dem Alphabet $\{\emptyset, 1\}$ gegeben durch

$$L = \{w \in \Sigma^* \mid w \text{ ist die Binärdarstellung einer Quadratzahl}\}$$

Das Problem wird entscheiden von einer Turing-Maschine, die Binärzahlen auf dem Band analysiert, ob sie Quadratzahlen sind, und im Zustand q_{accept} stehen bleibt, falls dies zutrifft.

- b) Sei $\Sigma = \{\emptyset, 1, :\}$ und

$$L = \left\{ w \in \Sigma^* \mid \begin{array}{l} w \text{ ist von der Form } w_1 : w_2, \text{ wobei } w_i \text{ Binärdarstellungen von} \\ \text{Zahlen } n_i \text{ sind mit } n_1 = n_2^2. \end{array} \right\}.$$

Diese Sprache kann von einer TM entschieden werden, welche n_2^2 berechnet und genau dann im Zustand q_{accept} anhält, wenn das Resultat mit n_1 übereinstimmt.

- c) Sei $\Sigma = \{\emptyset, 1, :\}$ und

$$L = \left\{ w \in \Sigma^* \mid \begin{array}{l} w \text{ ist von der Form } a:b:c, \text{ wobei } a, b \text{ und } c \text{ Binärdarstellun-} \\ \text{gen der Koeffizienten einer quadratischen Gleichung sind,} \\ \text{die reelle Lösungen hat.} \end{array} \right\}$$

Diese Sprache kann mit einer TM entschieden werden, die die Diskriminante $b^2 - 4ac$ berechnet und im Zustand q_{accept} stehen bleibt genau dann, wenn die Diskriminante ≥ 0 ist.

Abzählbar oder nicht?

5.13. Welche der folgenden Mengen sind abzählbar unendlich, welche sind überabzählbar?

- Die Menge aller kontextfreien Sprachen.
- Die Menge aller Entscheider.
- Die Menge aller Folgen a_1, a_2, \dots von rationalen Zahlen, die nach endlich vielen Gliedern konstant sind.
- Die Menge aller konvergenten Folgen a_1, a_2, a_3, \dots von

Lösung. a) Die kontextfreien Grammatiken kann man zum Beispiel nach Anzahl Regeln ihrer Chomsky-Normalform aufzählen, also ist die Menge der kontextfreien Grammatiken abzählbar.

- Die Menge der Folgen, die nach endlich vielen Gliedern konstant werden, kann man zerlegen in die Mengen F_k , der Folgen, die ab dem k -ten Glied konstant sind. Die Menge F_k besteht aus folgen, die k verschiedene rationale Glieder haben, also gleich mächtig wie \mathbb{Q}^{k+1} . Als abzählbare Vereinigung von abzählbaren Mengen ist die Menge der Folgen, die nach endlich vielen Gliedern konstant sind, also abzählbar.
- Jede reelle Zahl lässt sich durch eine Folge rationaler Zahlen approximieren. Daher ist die Menge der konvergenten Folgen mindestens so gross wie die Menge der reellen Zahlen, also überabzählbar. \circlearrowright

Kontextfreie Grammatik

4.5. Zeigen Sie, dass die Sprache $L = \{x\#y \mid x, y \in \{0, 1\}^*\}$ über dem Alphabet $\Sigma = \{\emptyset, 1, \#\}$ kontextfrei ist.

Lösung. Die folgende Grammatik erzeugt L :

$$\begin{aligned} S &\rightarrow W\#W \\ W &\rightarrow W\emptyset \\ &\rightarrow W1 \\ &\rightarrow \epsilon \end{aligned}$$

Noch einfacher hätte man argumentieren können, dass L vom regulären Ausdruck $.^*\#.^*$ akzeptiert wird, also regulär und damit erst recht kontextfrei sein muss. \circlearrowright

4.10. Sei $\Sigma = \{0, 1\}$. Wir sagen, ein Wort in Σ^* sei ‘‘wachsend’’, wenn auf jede Folge von Nullen eine mindestens so lange Folge von Einsen folgt. Betrachten Sie die Sprache

$$L = \{w \in \Sigma^* \mid w \text{ ist wachsend}\}$$

- a) Ist L regulär?
- b) Ist L kontextfrei?

Lösung. a) L ist nicht regulär, wie man mit dem Pumping-Lemma beweisen kann. Dazu nimmt man an, L sei regulär. Das Pumping-Lemma garantiert, dass es eine Zahl N gibt, die Pumping-Length, so dass Wörter mit gröserer Länge aufgepumpt werden können. Das Wort $0^N 1^N$ ist wachsend, denn auf die Folge von N Nullen folgen $N \geq N$ Einsen. Nach dem Pumping-Lemma kann $w = xyz$ geschrieben werden, mit $|xy| \leq N$ und $|y| > 0$. Insbesondere bestehen x und y ausschliesslich aus Nullen. Das aufgepumpte Wort xy^kz besteht also aus $N + |y|(k-1)$ Nullen, gefolgt von N Einsen. Da $N + |y|(k-1) > N$ für $k > 1$, ist das aufgepumpte Wort nicht mehr wachsend, im Widerspruch zur Behauptung des Pumping-Lemmas. Der Widerspruch zeigt, dass die Voraussetzung nicht zutreffen kann, L ist also nicht regulär.

b) Wachsende Wörter bestehen aus einzelnen Wörtern der Form $w = 0^k 1^{k+s}$. Das Wort w kann man erzeugen, indem man zuerst das leere Wort in \emptyset und 1 ‘‘einschachtelt’’, und dann noch eine Anzahl von Einsen anhängt. Damit haben wir aber in informeller Form alle Produktionsregeln für Wörter der Sprache L zusammengefasst.

Das Symbol W stehe für Wörter wie w , dann kann man beliebige Wörter der Sprache zusammensetzen, in dem man solchen Wörter weitere anhängt:

$$\begin{aligned} S_0 &\rightarrow \epsilon \\ &\rightarrow S \\ S &\rightarrow A \\ &\rightarrow SA \end{aligned}$$

Die Teile A entstehen wir folgt:

$$\begin{aligned} A &\rightarrow W \\ &\rightarrow A1 \\ W &\rightarrow \emptyset 1 \\ &\rightarrow \emptyset W1 \end{aligned}$$



Grammatik in Chomsky-Normalform

4.15. Die Grammatik

$$\begin{aligned} S &\rightarrow A \\ &\rightarrow Bc \\ A &\rightarrow AB \\ &\rightarrow ABC \\ &\rightarrow a \\ B &\rightarrow b \\ C &\rightarrow cc \end{aligned}$$

hat nicht Chomsky-Normalform. Finden Sie eine äquivalente Grammatik in Chomsky-Normalform.

Lösung. Die Ausgangsgrammatik ist

$$\begin{aligned} S &\rightarrow A \mid Bc \\ A &\rightarrow AB \mid ABC \mid a \\ B &\rightarrow b \\ C &\rightarrow cc \end{aligned}$$

Elimination der Unit rule $S \rightarrow A$:

$$\begin{aligned} S &\rightarrow Bc \mid AB \mid ABC \mid a \\ A &\rightarrow AB \mid ABC \mid a \\ B &\rightarrow b \\ C &\rightarrow cc \end{aligned}$$

Ersetzung der ABC -Regeln mit Hilfe einer zusätzlichen Variablen U :

$$\begin{aligned} S &\rightarrow Bc \mid AB \mid AU \mid a \\ A &\rightarrow AB \mid AU \mid a \\ U &\rightarrow BC \\ B &\rightarrow b \\ C &\rightarrow cc \end{aligned}$$

Elimination der Terminal-Symbole

$$\begin{aligned} S &\rightarrow BC' \mid AB \mid AU \mid a \\ A &\rightarrow AB \mid AU \mid a \\ U &\rightarrow BC \\ B &\rightarrow b \\ C &\rightarrow C'C' \\ C' &\rightarrow c \end{aligned}$$

Diese Grammatik hat offenbar Chomsky-Normalform. \circlearrowright