# Inhaltsverzeichnis



# 1 Logik

## 1.1 Prädikate

- Aussagen über mathematische Objekte, wahr oder falsch
- Funktionen mit booleschen Rückgabewerten:

$P, Q(n), R(x, y, z)$

## 1.2 Verknüpfungen

- und: $P \wedge Q$
- oder: $P \vee Q$
- nicht: $\neg P$
- Implikation $P \Rightarrow Q = \neg P \vee Q$

## 1.3 Distributivgesetze

- $P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$
- $P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$

## 1.4 De Morgan

- $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$
- $P \Rightarrow Q \Leftrightarrow (\neg Q \Rightarrow \neg P)$

## 1.5 Quantoren

- $\forall i \in \{1, \ldots, n\}(P_i)$ = (Für alle $i$ ist $P_i$ wahr)

- $\exists i \in \{1, ..., n\}(P_i)$ = (Es gibt ein $i$ derart, dass $P_i$ wahr ist)

### 1.5.1 Morgan 2.0

«Nicht für alle» = «Es gibt einen Fall, für den nicht»

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

# 2 Alphabet und Wort

- Alphabet: $\Sigma$
- Wort: Ein $n$-Tuplel in $\Sigma^n = \Sigma \times \cdots \times \Sigma$
- Menge aller Wörter: $\Sigma^* = \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \cdots = \bigcup_{k=0}^{\infty} \Sigma^k$

## 2.1 Wortlänge

- $|\varepsilon| = 0$
- $|01010|_0 = 3$
- $|(1201)^7| = 7 \cdot |1201| = 28, |w^n| = n \cdot |w|$

# 3 Reguläre Sprachen

## 3.1 Deterministische endliche Automaten (DEA)

**Definition 3.1.1**: Ein DEA ist ein Quintupel $(Q, \Sigma, \delta, q_0, F)$, wobei:

1. $Q$, Beliebige endliche Menge von **Zuständen**
2. $\Sigma$, **Alphabet** (Endliche Menge)
3. $\delta : Q \times \Sigma \to Q$, **Übergangsfunktion**
4. $q_0 \in Q$, **Startzustand**
5. $F \subset Q$, Menge der **Akzeptierzustände**

**Definition 3.1.2**: Sei $a$ ein endlicher Automat, dann ist $L(A)$ die Sprache

$$L(A) = \{w \in \Sigma^* \mid \text{w überführt A in einen Akzeptierzustand}\}$$

### 3.1.1 Beispiele für DEAs:

- Durch drei teilbare Zahlen
- Wörter mit einer geraden Anzahl $a$
- Bedingungen an einzelne Zeichen, wie: Wörter, die mit einem $a$ beginnen und genau ein $b$ enthalten.

### 3.1.2 Myhill-Nerode Automat

Wir müssen für beliebige Wörter herausfinden, mit welchem weiteren Input der Automat in einen Akzeptierzustand übergeht. Das leere Wort $\varepsilon$ ist speziell, wir benötigen die Sprache $L$ selbst, um zum Akzeptierzustand zu kommen. Beispiel:

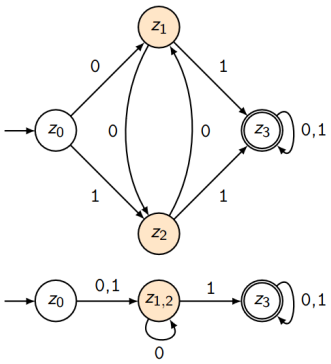
$$\Sigma = \{0, 1\}$$
$$L = \{w \in \Sigma^* \mid |w|_0 \text{ gerade}\}$$

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

### 3.1.3 Algorithmus für den Minimalautomaten

Wir füllen jeweils nur die untere Hälfte der Tabelle aus:

1. Tabelle erstellen mit allen Zuständen in den Zeilen und Spalten
2. Äquivalente Zustände (die Diagonale) mit $\equiv$ markieren
3. Akzeptierzustände von normalen Zuständen unterscheiden mit $\times$.
4. Falls man von einem Zustandspaar $(a, b)$ mit einem Übergang bei einem Paar mit $\times$ landet, kann man das Paar $(a, b)$ auch mit $\times$ markieren.




| | $z_0$ | $z_1$ | $z_2$ | $z_3$ |
|---|---|---|---|---|
| $z_0$ | $\equiv$ | $\times$ | $\times$ | $\times$ |
| $z_1$ | $\times$ | $\equiv$ | $\equiv$ | $\times$ |
| $z_2$ | $\times$ | $\equiv$ | $\equiv$ | $\times$ |
| $z_3$ | $\times$ | $\times$ | $\times$ | $\equiv$ |

Algorithmus

1. $q \in F, q' \notin F$
2. Unterscheidbar nach Übergang
3. Iteration bis keine Änderung mehr

$\Rightarrow$ $z_1$ und $z_2$ sind nicht unterscheidbar

### 3.1.4 Pumping Lemma

**Satz 3.1.4.1**: Ist $L$ eine reguläre Sprache, dann gibt es eine Zahl $N$, die Pumping Length, so dass jedes Wort $w \in L$ mit $|w| \geq N$ in drei Teile $w = xyz$ zerlegt werden kann, so dass:

$$|y| > 0$$
$$|xy| \leq N$$
$$xy^k z \in L, \forall k \geq 0$$

Zum beweisen, dass eine Sprache nicht regulär ist, nimmt man an, dass sie regulär ist und führt das Pumping Lemma durch. Gibt es einen Widerspruch, ist die Sprache nicht regulär (Widerspruchsbeweis).

*Beispiel* ($L = \{0^n 1^n \mid n \geq 0\}$):

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

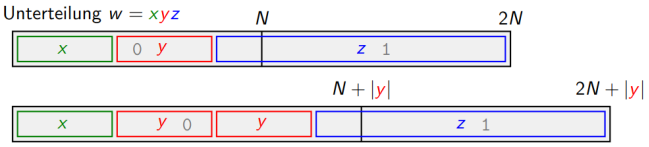



Abbildung 1: Pumping Lemma

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

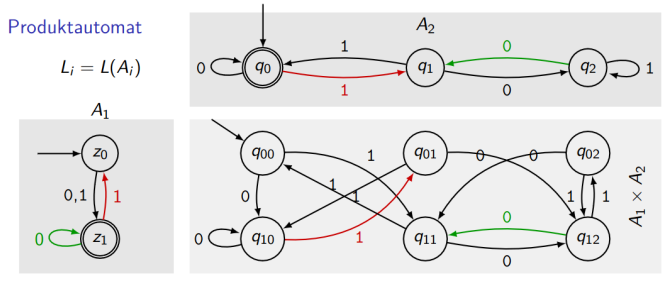
## 3.2 Nicht deterministische Automaten (NEA)

Ein DEA sieht immer nur ein Zeichen weit, kann sich nicht an ältere Zeichen erinnnern und kann Entscheidungen später nicht mehr revidieren. Zum Beispiel ist unklar, wie eine Bedingung wie «wenn ein Wort mit einer 0 aufhört, muss es auch mit einer 0 beginnen» implementiert werden müsste.

Jeder NEA lässt sich in einen DEA umwandeln, NEAs erkennen somit dieselbe Sprache wie DEAs.

NEAs erlauben auch verschiedene Zustandsänderungen durch dieselben Symbole oder gar keine Zustandsänderungen ($|\delta(q, a)| \neq 1$) und Zustandsänderungen durch das leere Wort ($\delta(q, \varepsilon) \neq \emptyset$)). Wenn ein NEA $\varepsilon$-Änderungen erlaubt, nennt man ihn auch NEA$_\varepsilon$.

## 3.3 Mengenoperationen




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

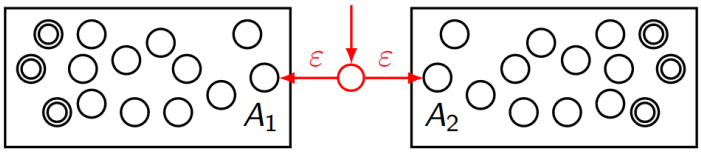
Vereinigung $L_1 \cup L_2$
$F = F_1 \times Q_2 \cup Q_1 \times F_2$

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

# 4 Reguläre Operationen und Ausdrücke

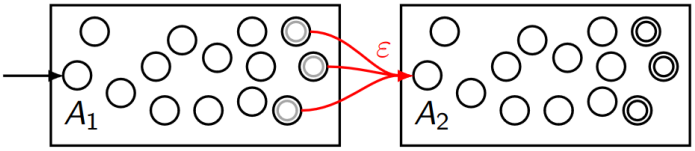
## 4.1 Alternative

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



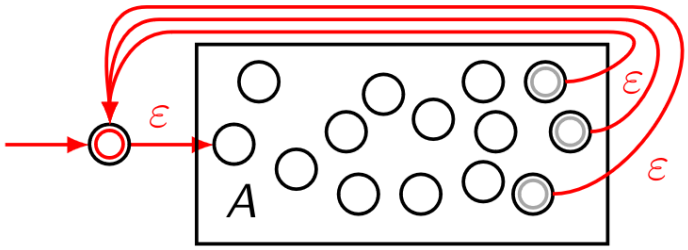

## 4.2 Verkettung

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




## 4.3 *-Operation

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




## 4.4 Reguläre Ausdrücke

- Metazeichen `() [] * ? | . \`
- `[abc]` = `a|b|c`, `[^abc]` = nicht `[abc]` (das ^-Symbol bedeutet innerhalb `[]` «nicht»)

### 4.4.1 Erweiterungen

- `{n, m}`: zwischen $n$ und $m$
- `+`: mindestens eines (`{1,}`)
- `?`: optional (`{0, 1}`)
- `\d` Ziffern, `\s` whitespace, `\w` Wortzeichen (inkl. Ziffern), `[:space:]`, `[:lower:]`, `[:xdigit:]`

# 5 Kontextfreie Sprachen

## 5.1 Grammatik für Klammerausdrücke

$$\begin{aligned} A &\to \varepsilon \\ &\to AA \\ &\to (A) \end{aligned}$$

Oder (Kurzschreibweise):

$$A \to \varepsilon \mid AA \mid (A)$$

### 5.1.1 Beispiel für Ableitung von Klammerausdruck

Wort $(()())$ soll abgeleitet werden:

$$A \to (A) \to (AA) \to ((A)A) \to ((A)(A)) \to ((\varepsilon)(A)) \to ((\varepsilon)(\varepsilon)) \to (()())$$

## 5.2 Parse Tree

**Definition 5.2.1**: Zwei Ableitungen eines Wortes $w$ einer kontextfreien Sprache $L(G)$ heissen äquivalent, wenn sie den gleichen Ableitungsbaum haben. Hat eine Sprache Wörter mit verschiedenen Ableitungen, heisst sie mehrdeutig (engl. ambiguous)

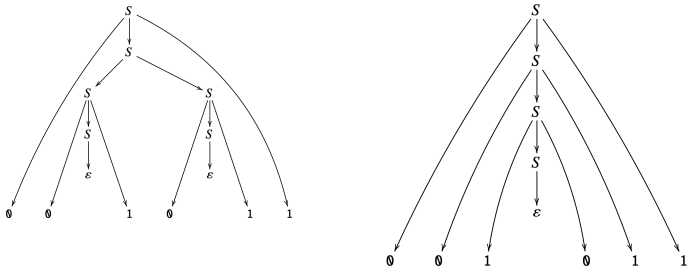


Abbildung 2: Nicht eindeutig ableitbare Grammatik

Ein Beispiel einer Grammatik, die nicht eindeutige Ableitungen hat, ist:

$$G = (\{S\}, \{0,1\}, \{S \to 0S1 \mid 1S0 \mid SS \mid \varepsilon\}, S)$$

Ein Ausruck 001011 produziert verschiedene Parsetrees (siehe Abbildung 2).

Von besonderer praktischer Bedeutung sind jedoch Grammatiken, die immer eindeutig ableitbar sind. Ein Beispiel solch einer Grammatik ist die `expression-term-factor`-Grammatik für einfache arithmetische Ausdrücke.

## 5.3 Reguläre Operationen

$L_1, L_2$ 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 $R$

$\Rightarrow G = (V, \Sigma, R, S_0)$

### 5.3.1 Alternative

Regeln für $L_1 \cup L_2$:

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

Im Wesentlichen bleiben die einzelnen Regeln dieselben, es müssen nun einfach alle Regeln der Sprachen $L_1, L_2$ untereinander geschrieben werden. Wichtig ist jedoch, dass wir einen neuen Startzustand konstruieren, den wir mit Regeln in $S_1, S_2$ überführen.

### 5.3.2 Verkettung

Regeln für $L_1 L_2$:

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

### 5.3.3 *-Operation

Regeln für $L_1^*$:

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

Diese Regel besagt einfach, dass $S_1$ beliebig oft wiederholt werden kann.

**Satz 5.3.3.1**: Die Klasse der kontextfreien Sprachen ist abgeschlossen unter regulären Operationen.

## 5.4 Reguläre Sprachen sind kontextfrei

### 5.4.1 Bausteine

- Einzelne Buchstaben:
  - $A \to a$
  - $B \to b$
  - ...
  - $Z \to z$
- Leeres Wort
  - empty $\to \varepsilon$

### 5.4.2 regex → CFG

*Beispiel*: Grammatik zum regulären Ausdruck `(ab)*|c`:

$$\text{Alternative: } S \to U \mid C$$
$$\text{*-Operation: } U \to UP \mid \varepsilon$$
$$\text{Verkettung: } P \to AB$$
$$\text{a: } A \to a$$
$$\text{b: } B \to b$$
$$\text{c: } C \to c$$

## 5.5 Arithmetische Ausdrücke

$$\begin{aligned}
\text{expression} &\to \text{expression} + \text{term} \\
&\to \text{term} \\
\text{term} &\to \text{term} * \text{factor} \\
&\to \text{factor} \\
\text{factor} &\to (\text{expression}) \\
&\to \text{N} \\
\text{N} &\to \text{NZ} \\
&\to \text{Z} \\
Z &\to 0|1|2|3|4|5|6|7|8|9
\end{aligned}$$

## 5.6 Chomsky Normalform

### 5.6.1 Anforderungen an eine Grammatik

1. Keine Unit-Rules $A \to B$
2. Keine Regeln $A \to \varepsilon$ ausser wenn nötig $S \to \varepsilon$ (Startvariable $S$)
3. Keine Regeln mit mehr als 2 Variablen auf der rechten Seite

$\Rightarrow$ Rechte Seite enthält genau zwei Variablen oder genau ein Terminalsymbol

### 5.6.2 Transformation in CNF

1. Neue Startvariable $S_0 \to S$
2. $\varepsilon$-Regeln
3. Unit-Rules
4. Verkettungen auflösen

*Beispiel*:

$$\begin{aligned}
S &\to ASA \mid aB \\
A &\to B \mid S \\
B &\to b \mid \varepsilon
\end{aligned}$$

**Neue Startvariable:**

$$\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid aB \\
A &\to B \mid S \\
B &\to b \mid \varepsilon
\end{aligned}$$

**ε eliminieren:**

$$\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid aB \mid a \\
A &\to B \mid S \mid \varepsilon \\
B &\to b
\end{aligned}$$

$$\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid SA \mid AS \mid S \mid aB \mid a \\
A &\to B \mid S \\
B &\to b
\end{aligned}$$

**Unit-Rules:**

$$\begin{aligned}
S_0 &\to S \\
S &\to ASA \mid SA \mid AS \mid aB \mid a \\
A &\to b \mid S \\
B &\to b
\end{aligned}$$

$$\begin{aligned}
S_0 &\to ASA \mid SA \mid AS \mid aB \mid a \\
S &\to ASA \mid SA \mid AS \mid aB \mid a \\
A &\to ASA \mid SA \mid AS \mid aB \mid a \mid b \\
B &\to b
\end{aligned}$$

**Verkettungen, Terminalsymbole:**

$$\begin{aligned}
S_0 &\to AA_1 \mid SA \mid AS \mid aB \mid a \\
S &\to AA_1 \mid SA \mid AS \mid aB \mid a \\
A &\to AA_1 \mid SA \mid AS \mid aB \mid a \mid b \\
B &\to b \\
A_1 &\to SA
\end{aligned}$$

$$\begin{aligned}
S_0 &\to AA_1 \mid SA \mid AS \mid UB \mid a \\
S &\to AA_1 \mid SA \mid AS \mid UB \mid a \\
A &\to AA_1 \mid SA \mid AS \mid UB \mid a \mid b \\
B &\to b \\
A_1 &\to SA \\
U &\to a
\end{aligned}$$

**Definition 5.6.2.1** (Chomsky-Normalform): 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.

### 5.6.3 Anwendungen der Chomsky-Normalform

**Gegeben:** Grammatik $G$ in Chomsky-Normalform

**Satz 5.6.3.1**: Ableitung eines Wortes $w \in L(G)$ ist immer in $2|w| - 1$ Regelanwendungen möglich.

*Beweis*:
- $|w| - 1$ Regeln der Form $A \to BC$ um aus $S$ ein Wort aus $w$ Variablen zu erzeugen
- $|w|$ Regeln der Form $A \to a$, um das Wort $w$ zu erzeugen

■

### 5.6.4 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.

### 5.6.5 CYK-Ideen

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

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

### 5.6.6 CYK-Algorithmus

```
boolean ableitbar(Variable A, String w) {
  if (w.length() == 0) {
    return A → ε ∈ R;
  }
  if (w.length() == 1) {
    return A → w ∈ R;
```

```
  }
  foreach Unterteilung w = w1w2 {
    foreach A → BC ∈ R {
      if (ableitbar(B, w1)
        && ableitbar(C, w2)) {
          return true;
        }
    }
  }
  return false;
}
```

## 5.7 Stack-Automaten

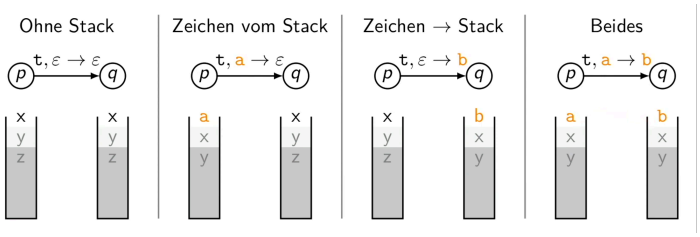



Abbildung 3: Stackübergänge

$$a, b \to c$$

- $a$: vom Input
- $b$: vom Stack entfernen (Bedingung)
- $c$: auf den Stack

*Beispiel*: $\{0^n 1^n \mid n \geq 0\}$

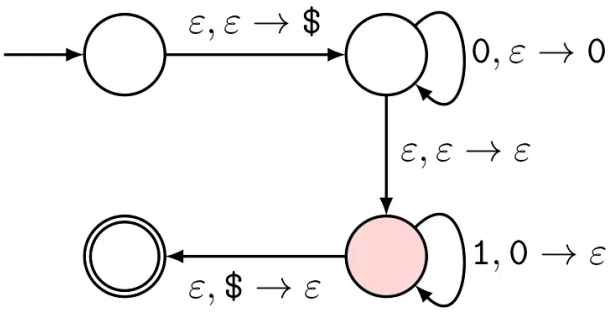



Abbildung 4: Stackautomat

**Definition 5.7.1** (Stackautomat):

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

1. $Q$: Zustände
2. $\Sigma$: Eingabe-Alphabet
3. $\Gamma$: Stack-Alphabet
4. $\delta$: $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$, Regeln
5. $q_0 \in Q$: Startzustand
6. $F \subset Q$: Akzeptierzustände

Die Regeln hängen ab vom aktuellen Zustand, vom aktuellen Inputzeichen und vom Zeichen, das zuoberst auf dem Stack liegt.

$\delta$: Funktion mit drei Inputs (aktueller Zustand, Input-Zeichen, oberstes Zeichen auf dem Stack) und gibt ein Tupel zurück mit neuem Zustand und neuem obersten Zeichen auf dem Stack.

**Beachte:**
- Immer nicht-deterministisch
- $\Gamma \neq \Sigma$ möglich (z.B. $-Symbol auf den Stack)

### 5.7.1 Visualisierung eines Stacks

*Beispiel*:

Grammatik:

$$S \to 0S1$$
$$\to \varepsilon$$

Input: 0 0 0 1 1 1

1. $-Symbol auf den Stack (um später zu prüfen, ob der Stack leer ist)
2. Variable $S$ auf den Stack
3. Input 0 matcht nicht → Variable $S \to 0S1$ (1 auf den Stack, dann $S$, dann 0)
4. 0 matcht → 0 vom Stack entfernen und weiter
5. 0 matcht nicht $S \to 0S1$

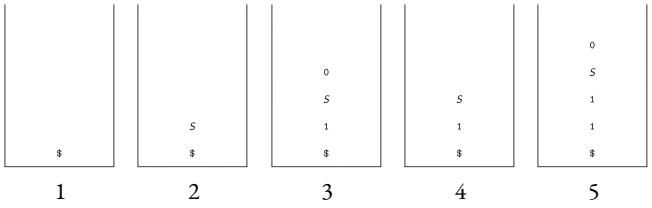



Abbildung 5: Stack-Visualisierung

**Zustandsdiagramm:**

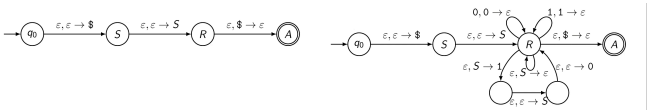


Abbildung 6: Zustandsdiagramm erstellen

### 5.7.2 Grammatik → Stackautomat

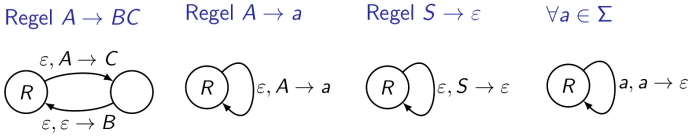



Abbildung 7: CNF zu Stackautomat

### 5.7.3 Backus-Naur-Form (BNF)

Grammatik:

$$S \to \varepsilon$$
$$\to 0S1$$

BNF:

```
<string> ::= '' | 0 <string> 1
```

*Beispiel (Expression-Term-Factor)*:

```
<expression> ::= <expression> + <term> | <term>
<term> ::= <term> * <factor> | <factor>
<factor> ::= ( <expression> ) | <number>
<number> ::= <number> <digit> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

## 5.8 PDA zu CFG

**Variablen**

Wörter beschreiben Pfade durch den Automaten:

Variable $A_{pq}$ = Wörter, die von $p$ nach $q$ führen mit leerem Stack

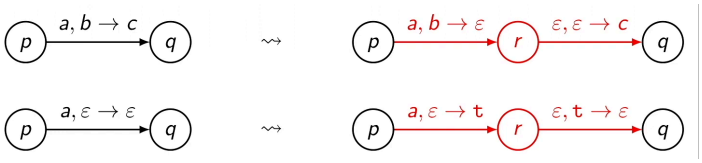
**Regeln**

Regeln beschreiben, wie sich Wege zerlegen lassen.

$$A_{pq} \to A_{pr} A_{rq}$$

### 5.8.1 Stackautomat standardisieren

1. Nur ein Akzeptierzustand
   - $\forall q \in F$: Übergang in neuen Akzeptierzustand $q_a$
2. Stack leeren:
   - Markierungszeichen zu Beginn auf den Stack
   - Am Schluss: $\varepsilon, . \to \varepsilon$ und dann $\varepsilon, \$ \to \varepsilon$ von $q_a$ nach $q'_a$
3. Jeder Übergang legt entweder ein Zeichen auf den Stack oder entfernt eines




### 5.8.2 Regeln

$A_{pq}$ wird, falls sich der Stack dazwischen nicht leert, zu folgendem:

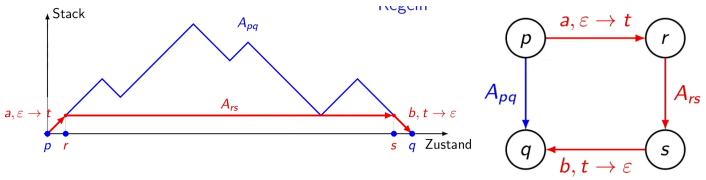



Abbildung 8: Regel, falls der Stack dazwischen nie leer wird

$$A_{pq} \to aA_{rs}b$$

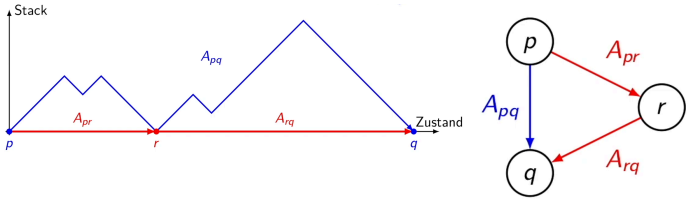


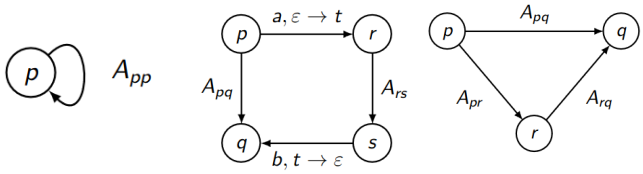

Abbildung 9: Falls der Stack zwischendurch leer wird

$$A_{pq} \to A_{pr}A_{rq}$$

### 5.8.3 Grammatik ablesen

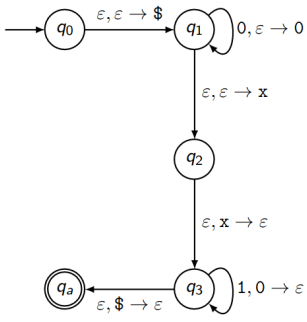
Ausgangspunkt: Standardisierte Grammatik mit Startzustand $q_0$ und $F = \{q_a\}$.

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




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

*Beispiel (PDA zu CFG)*:




**Grammatik**

$$A_{q_0q_a} \to \varepsilon A_{q_1q_3}\varepsilon$$
$$A_{q_1q_3} \to 0A_{q_1q_3}1$$
$$\to \varepsilon A_{q_2q_2}\varepsilon$$
$$A_{q_2q_2} \to \varepsilon$$

**Vereinfachung**

$$S_0 \to S$$
$$S \to 0S1$$
$$\to \varepsilon$$

$\to$ Grammatik für $L = \{0^n1^n \mid n \geq 0\}$

# 6 Nicht kontextfreie Sprachen

## 6.1 Pumping-Lemma für kontextfreie Sprachen

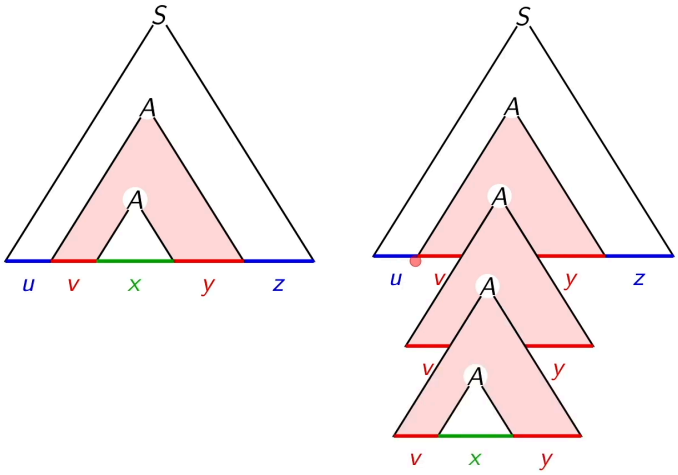



Abbildung 11: Herleitung Pumping Lemma für CFGs

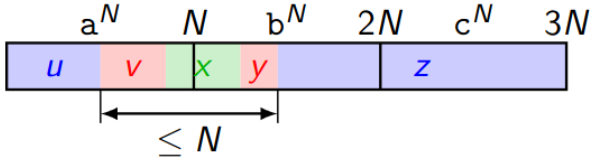
**Definition 6.1.1** (Pumping Lemma für CFL): Ist $L$ eine CFL, dann gibt es eine Zahl $N$, die Pumping Length, derart, dass jedes Wort $w \in L$ mit $|w| \geq N$ zerlegt werden kann in fünf Teile $w = uvxyz$ derart, dass:

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*: $\{a^nb^nc^n \mid n \geq 0\}$

1. AnnahmeL $L$ kontextfrei
2. Pumping Length $N$
3. Wort: $w = a^Nb^Nc^N$
4. Zerlegungen (mehrere):




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

# 7 Turing-Maschinen

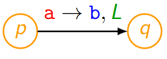
**Definition 7.1**: Eine Turing-Maschine ist ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, wobei:

1. $Q$ heisst die Menge der Zustände
2. $\Sigma$ heisst das Inputalphabet, es enthählt das Blank-Zeichen **nicht**[1]
3. $\Gamma$ ist das Bandalphabet, es enthält das Blank-Zeichen und $\Sigma \subset \Gamma$
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ ist die Übergangsfunktion

## 7.1 Spielregeln

1. Der Speicher ist unbegrenzt
2. In jeder Zelle steht genau ein Zeichen
3. Es ist immer nur eine Speicherzelle einsehbar
4. Der Inhalt der aktuellen Zelle kann beliebig verändert werden
5. Bewegung immer nur eine Zelle nach links oder rechts
6. Kein weiterer Speicher (nur die Zustände eines endlichen Automaten)

## 7.2 Zustandsdiagramm

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


## 7.3 Von einer TM erkannte Sprache

[1] Wäre das Blank-Zeichen in $\Sigma$, könnte man das leere Band nicht vom Input unterscheiden.

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

**Definition 7.3.1**: Ein Entscheider ist eine Turing-Mashchine, die auf jedem Input $w \in \Sigma^*$ anhält. Eine Sprache heisst entscheidbar, wenn ein Entscheider sie erkennt.

Eine Turing-entscheidbare Sprache ist auch Turing-erkennbar. Die Eigenschaft «Turing-entscheidbar» unterscheidet sich von der Eigenschaft «Turing-erkennbar» nur dadurch, dass bei einer entscheidbaren Sprache die Turing-Maschine auf jedem beliebigen Input anhalten muss, während bei einer nur erkennbaren Sprache einzelne Input-Wörter auch dazu führen können, dass die Turing-Maschine endlos weiterrechnet.

## 7.4 Turing Maschinen und moderne Computer

| **Turing Maschine** | **Moderner Computer** |
|---|---|
| 1. Zustände Q | 1. Zustände der CPU: Statusbits, Registerwerte |
| 2. Band, unendlich grosser Speicher | 2. Virtueller Speicher: praktisch unbegrenzt |
| 3. Schreib-/Lesekopf | 3. Adress-Register, Programm-Zähler |
| 4. Anhalten, $q_{\text{accept}}$ und $q_{\text{reject}}$ | 4. `exit(EXIT_SUCCESS)`, `exit(EXIT_FAILURE)` |
| 5. Problemspezifisch | 5. Kann beliebige Programme ausführen |

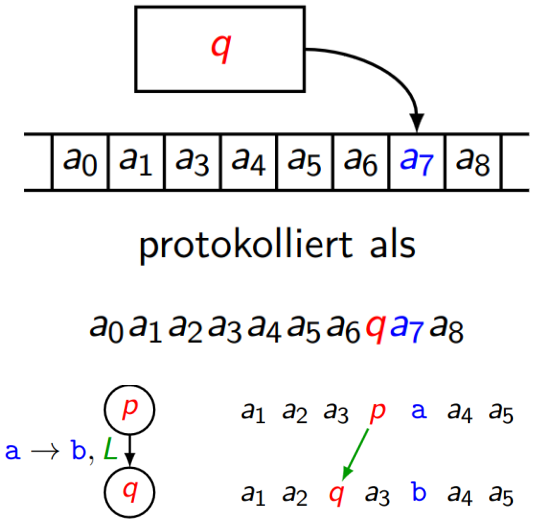
## 7.5 Aufzähler

**Definition 7.5.1** (Aufzähler): Ein Aufzähler ist eine TM, die alle akzeptierbaren Wörter mit einem Drucker ausdruckt.

**Definition 7.5.2** (Rekursiv aufzählbare Sprache): Eine Sprache $L$ heisst rekursiv aufzählbar, wenn es einen Aufzähler gibt, der sie aufzählt.

Aufzählbare Sprache $\iff$ Turing-erkennbare Sprache.

## 7.6 Berechnungsgeschichte




## 7.7 Vergleich von Maschinen

**Definition 7.7.1**: Eine TM $M_1$ ist «leistungsfähiger» als eine TM $M_2$, wenn $M_1$ die Maschine $M_2$ simulieren kann

$$M_2 \lesssim_S M_1$$

$M_2$ ist simulierbar auf $M_1$.

Beispiele:
- TM $\leq_S$ Minecraft
- 8-Bit CPU $\leq_S$ Minecraft $\leq_S$ TM

## 7.8 Unendlichkeit

Die Mengen $\mathbb{N}$ und $\mathbb{R}$ sind nicht gleich mächtig, da es keine bijektive Abbildung $\mathbb{N} \to \mathbb{R}$ gibt.

**Definition 7.8.1**: Mengen $A$ und $B$ heissen *gleich mächtig*, $A \simeq B$, wenn es eine Bijektion $A \to B$ gibt.

**Definition 7.8.2**: Eine Menge $A$ heisst *unendlich*, wenn sie gleich mächtig wie eine echte Teilmenge ist.

**Definition 7.8.3**: $A$ heisst *abzählbar unendlich*, wenn $A \simeq \mathbb{N}$, d.h. $A$ gleich mächtig wie $N$ ist.

**Satz 7.8.1**: Die Potenzmenge $P(A)$ einer abzählbar unendlichen Menge $A$ ist immer *überabzählbar unendlich*.

**Satz 7.8.2**: Die Vereinigung von endlich vielen abzählbaren Mengen ist abzählbar. Das kartesische Produkt $A \times B$ zweier abzählbaren Mengen $A, B$ ist abzählbar.

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

# 8 Entscheidbarkeit

Eine Sprache heisst entscheidbar, wenn es einen Entscheider gibt, der prüfen kann, ob ein Wort $w$ in der Sprache liegt.

*Beispiel*: Sei $A$ ein DEA. Dann kann man folgenden Algorithmus $M_A$ mit Input $w$ bauen:
1. Simuliere $A$ auf $w$
2. Falls $A$ in Akzeptierzustand: $q_{\text{accept}}$
3. Andernfalls: $q_{\text{reject}}$

Daraus folgt $L(A) = L(M_A)$, oder $M_A$ ist ein Entscheider für die Sprache $L(A)$.

**Satz 8.1**: Reguläre Sprachen sind entscheidbar.

*Beispiel*: Sei $G$ eine CFG. Dann kann man folgenden Algorithmus $M_G$ mit Input $w$ bauen:
1. Wandle $G$ in Chomsky-Normalform $G'$ um
2. Wende den CYK-Algorithmus auf $G'$ und Wort $w$ an
3. Wenn der CYK-Algorithmus das Wort $w$ akzeptiert: $q_{\text{accept}}$
4. Andernfalls: $q_{\text{reject}}$

## 8.1 Berechenbare Zahlen

**Definition 8.1.1** (Berechenbare Zahl): Eine Zahl $w$ heisst berechenbar, wenn es eine Turingmaschine $M$ gibt, die auf leerem Band startet und auf dem Band nacheinander die Stellen der Zahl ausgibt.

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

*Beispiel*: $\pi, e, \sqrt{2}, \gamma, \varphi = \frac{\sqrt{5}-1}{2}$

### 8.1.1 Wieviele berechenbare Zahlen gibt es?

*Sind alle 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. $\Rightarrow$ fast alle reelle Zahlen sind nicht berechenbar

## 8.2 Hilberts 10. Problem

Gibt es ganzzahlige Lösungen für Polynomgleichungen?

$$x^2 - y = 0$$
$$x^n + y^n = z^n$$

Eine TM, die verschiedene Möglichkeiten ausprobiert, ist hierfür nicht geeignet; falls es keine Lösung gibt, würde die TM nicht anhalten. $\Rightarrow$ Es bräuchte einen Entscheider. Dieses Problem wurde jedoch 1970 von Yuri Matiyasevich als nicht entscheidbar bewiesen.

## 8.3 Sprachprobleme

Ein normales Problem soll zunächst in eine Ja/Nein-Frage übersetzt werden.

*Beispiel*: Problem: Finden sie eine Lösung der quadratischen Gleichung:

$$x^2 - x - 1 = 0$$

Dies ist jedoch nicht wirklich als Sprache formuliert. Wir können folgende Formulierung postieren:

Sei $L$ die Sprache bestehend aus Wörtern der Form

$$w = \mathrm{a} = a, \mathrm{b} = b, \mathrm{c} = c, \mathrm{x} = x$$

wobei $a, b, c, x$ Dezimaldarstellungen von Zahlen sind. Ein Wort gehört genau dann zur Sprache $w$, wenn $ax^2 + bx + c = 0$ gilt. Ist $\mathrm{a} = 1, \mathrm{b} = -1, \mathrm{c} = -1, \mathrm{x} = 3 \in L$?

*Beispiel*: Gegeben ist eine natürliche Zahl $n$, berechne die ersten 10 Stellen der Dezimaldarstellung von $\sqrt{n}$.

Dies ist wieder kein normales Sprachproblem $\rightarrow$ Als Entscheidungsproblem mit Ja/Nein-Antwort formulieren:

Sind die ersten 10 Stellen der Dezimaldarstellung von $\sqrt{2} = 1.414213562$?

**Als Sprache formuliert:** Sei $L$ die Sprache bestehend aus Zeichenketten der Form $n, x$ wobei n die Dezimaldarstellung einer natürlichen Zahl ist und $x$ die ersten 10 Stellen einer Dezimalzahl. Gilt $2, 1.414213562 \in L$?

### 8.3.1 $\varepsilon$-Akzeptanzproblem für endliche Automaten

**Problem:** Kan der endliche Automat $A$ das leere Wort akzeptieren?

**Als Sprachproblem:** Ist die Sprache $L = \{\langle A\rangle \mid \varepsilon \in L(A)\}$ entscheidbar?

**Entscheidungsalgorithmus:**
1. Wandle $A$ in einen DEA um
2. Ist der Startzustand ein Akzeptierzustand $q_0 \in F$?

### 8.3.2 Gleichheitsproblem für DEAs

**Problem:** Akzeptieren die endlichen Automaten $A_1$ und $A_2$ die gleiche Sprache, $L(A_1) = L(A_2)$?

**Sprachproblem:** Ist die Sprache $L = \{\langle A_1, A_2\rangle \mid L(A_1) = L(A_2)\}$ entscheidbar?

**Entscheidungsalgorithmus:**
1. Wandle $A_1$ in einen minimalen Automaten $A_1'$ um
2. Wandle $A_2$ in einen minimalen Automaten $A_2'$ um
3. Ist $A_1' = A_2'$?

### 8.3.3 Akzeptanzproblem für DEAs

**Problem:** Akzeptiert der endliche Automat $A$ das Wort $w$?

**Spachproblem:** Ist die Sprache $L = \{\langle A, w\rangle \mid w \in L(A)\}$ entscheidbar?

**Entscheidungsalgorithmus:**
1. Wandle A in einen DEA A' um
2. Simuliere A' mit Hilfe einer TM
3. Hält die Turing-Maschine im Zustand $q_{\text{accept}}$?

### 8.3.4 Akzeptanzproblem für CFGs

**Problem:** Kann das Wort $w$ aus der Grammatik $G$ produziert werden?

**Als Sprachproblem:** Ist die Sprache $L = \{\langle G, w\rangle \mid w \in L(G)\}$ entscheidbar?

**Entscheidungsalgorithmus:**
1. Kontrollieren, dass $\langle G\rangle$ wirklich eine Grammatik beschreibt
2. Grammatik in Chomsky-Normalform bringen
3. Mit dem CYK-Algorithmus prüfen, ob $w$ aus $G$ produziert werden kann

## 8.4 Defintion Entscheidbarkeit

**Definition 8.4.1** (Entscheider): Ein Entscheider ist eine Turing Maschine, die auf jedem beliebigen Input anhält.

**Definition 8.4.2** (Entscheidbar): Eine Sprache $L$ heisst entscheidbar, wenn es einen Entscheider $M$ gibt mit $L = L(M)$. Man sagt, $M$ entscheidet $L$.

Jedes Problem kann in ein Sprachproblem übersetzt werden:

$$L_P = \{w \in \Sigma^* \mid w \text{ ist Lösung des Problems } P\}$$

*Beispiel (Leerheitsproblem)*:

$$E_{\text{DEA}} = \{\langle A\rangle \mid A \text{ ein DEA und } L(A) = \emptyset\}$$

*Beispiel (Gleichheitsproblem)*:

$$EQ_{\text{CFG}} = \{\langle G_1, G_2\rangle \mid G_i \text{ CFGs und } L(G_1) = L(G_2)\}$$

*Beispiel (Akzeptanzproblem)*:

$$A_{\text{DEA}} = \{\langle A, w\rangle \mid A \text{ ein DEA, der } w \text{ akzeptiert}\}$$

*Beispiel (Halteproblem)*:

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ hält auf Input } w\}$$

## 8.5 Nicht entscheidbare Probleme

**Satz 8.5.1** (Alan Turing): $A_{\text{TM}}$ ist nicht entscheidbar.

*Beweis*: Konstruiere aus dem Entscheider $H$ für $A_{\text{TM}}$ eine Maschine $D$ mit Input $\langle M\rangle$.
1. Lasse $H$ auf Input $\langle M, \langle M\rangle\rangle$ laufen
2. Falls $H$ akzeptiert: $q_{\text{reject}}$
3. Falls $H$ verwirft: $q_{\text{accept}}$

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! ■

**Satz 8.5.2** (Halteproblem): Das spezielle Halteproblem

$$HALT\varepsilon_{\text{TM}} = \{\langle M\rangle \mid M \text{ ist eine Turingmaschine und } M \text{ hält auf leerem Band}\}$$

ist nicht entscheidbar.

**Satz 8.5.3** (Allgemeines Halteproblem): Das allgemeine Halteproblem

$$HALT_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ ist eine Turingmaschine und } M \text{ hält auf Input } w\}$$

ist nicht entscheidbar.

Weitere nicht entscheidbare Probleme:
- Leerheitsproblem $E_{\text{TM}}$

## 8.6 Reduktion

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

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

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.

### 8.6.1 Reduktionsbeispiele

In folgenden Beispielen ist $M$ eine Maschine, die ein Wort $w$ entweder akzeptiert oder verwirft. Wie wir bewiesen haben, ist es unmöglich, einen Entscheider für das Akzeptanzproblem $A_{\text{TM}}$ zu konstruieren.

*Beispiel (Reduktion für das spezielle Halteproblem)*:

Programm $S$:
1. Führe $M$ auf $w$ aus
2. $M$ hält in $q_{\text{accept}}$: akzeptiere
3. $M$ hält in $q_{\text{reject}}$: Endlosschleife

Wenn $H$ ein Entscheider für $HALT\varepsilon_{\text{TM}}$ wäre, könnte man daraus einen Entscheider für $A_{\text{TM}}$ konstruieren:
1. Konstruiere das Programm $S$
2. Wende $H$ auf $\langle S \rangle$ an

*Beispiel (Reduktion für das Leerheitsproblem $A_{\text{TM}} \leq E_{\text{TM}}$)*: Ist die Sprache $L(M)$ leer? $\to \overline{E}_{\text{TM}}$ ist $L(M) \neq \emptyset$

Programm $S$ mit Input $u$:
1. $M$ auf $w$ laufen lassen
2. $M$ akzeptiert $w$: $q_{\text{accept}}$
3. Andernfalls: $q_{\text{reject}}$

$$M \text{ akzeptiert } w \Leftrightarrow S \text{ akzeptiert } L(S) = \Sigma^* \neq \emptyset$$

Wenn $H$ ein Entscheider für $E_{\text{TM}}$ wäre, könnte man daraus einen Entscheider für $A_{\text{TM}}$ konstruieren:
1. Konstruiere das Programm $S$
2. Wende $H$ auf $\langle S \rangle$ an

*Beispiel (Regularitätsproblem $A_{\text{TM}} \leq REGULAR_{\text{TM}}$)*: 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 $w$: $q_{\text{accept}}$
4. $q_{\text{reject}}$

$M$ akzeptiert $w$ $\Leftrightarrow S$ akzepiert $\{0^n 1^n \mid n \geq 0\}$, nicht regulär

$M$ akzeptiert $w$ nicht $\Leftrightarrow S$ akzeptiert $\emptyset$, regulär

Wäre $H$ ein Entscheider für $REGULAR_{TM}$, könnte man daraus einen Entscheider für $A_{\text{TM}}$ konstruieren:
1. Konstruiere das Programm $S$
2. Wende $H$ auf $\langle S \rangle$ an

## 8.7 Satz von Rice

Eigenschaften Turing-erkennbarer Sprachen
- $REGULAR$: $L(M)$ ist regulär
- $E$: $L(M)$ ist leer

**Definition 8.7.1**: Eine Eigenschaft $P$ Turing-erkennbarer Sprachen heisst nichttrivial, wenn es zwei Turingmaschinen $M_1$ und $M_2$ gibt, mit:

$$L(M_1) \text{ hat Eigenschaft } P$$
$$L(M_2) \text{ hat Eigenschaft } P \text{ nicht}$$

**Satz 8.7.1** (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.

# 9 Komplexität

## 9.1 P – NP

### 9.1.1 Folgerungen aus P = NP

1. Für jedes Problem in NP gibt es einen polynomiellen Algorithmus
2. Es gibt keine «schwierigen» Probleme
3. Mit Moore's Law lässt sich jedes Problem «lösen»
4. Es gibt keine sichere Kryptographie

### 9.1.2 Folgerungen aus P ≠ NP

1. NP-vollständige Probleme haben nicht skalierende Lösungen
2. Moore's Law hilft nicht in NP \ P

### 9.1.3 NP-vollständig

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

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

Wenn ein Problem NP-vollständig ist:
- Lösung braucht typischerweise exponentielle Zeit
- Korrektheit der Lösung ist leicht (in polynomieller Zeit) zu prüfen
- Andernfalls wären Tests schon exponentiell und somit in Software nicht sinnvoll

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

## 9.2 Aufüllrätsel

Viele Ausfüllrätsel wie z.B. Sudoku sind exponentiell lösbar. Man versucht dabei einfach jede Möglichkeit und falls eine Möglichkeit nicht zu einer korrekten Lösung führt, machen wir ein Backtracking. Man spricht von einer Nicht-Deterministischen Turing-Maschine, bei welcher wir alle Möglichkeiten durchprobieren müssen. Eine solche Maschine kann aber polynomiell verifiziert werden, in dem man den Pfad durchgeht, welcher verwendet worden ist für die Lösung des Rätsels und prüft, ob dieser in $q_{\text{accept}}$ landet.

## 9.3 Polynome Verifizierer

## 9.4 Reduktion Sudoku → CONSTRAINTS → SAT

Sudoku-Regeln werden als logische «Constraints» formuliert. Dabei müssen alle Constraints erfüllt sein. Es gibt Software/Libraries wie python-constraint für diese Probleme.

**Allgemein**: Jedes Ausfüllrätsel lässt sich auf CONSTRAINTS = SAT reduzieren.

### 9.4.1 SAT

Eine logische Formel ist in SAT genau dann, wenn sie erfüllt werden kann. Constraints werden miteinander «verundet», diese Formel soll wahr ergeben.

$$\text{SAT} = \{\varphi \mid \varphi \text{ erfüllbar}\}$$

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

Beschreibe das Finden der Berechnungsgeschichte von $M$ als polynomielles Ausfüllrätsel.

## 9.5 Karp-Katalog

### 9.5.1 SAT

Das SAT-Problem prüft im Allgemeinen, ob ein Ausdruck `true` wird. Dann ist er *satisfiable*. Es sind folgende Dinge gegeben:

- Aussagenlogische Formel

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Variablen (boolesche Werte)
- Aktion, die den Wahrheitszustands einer Variable verändert
- (falls vorhanden) Zwischenausdrücke von logischen Formeln
- Finaler Logischer Ausdruck zur Erfüllung des Ausgangproblems

SAT eignet sich für Probleme, bei welchen Elemente (Variablen) nur zwei Zustände einnehmen können. Jedes Ausfüllrätsel ist mit SAT beschreibbar. Man kann Wahrheitstabellen bilden. Jedes Problem, das sich auf SAT reduzieren lässt, ist NP-vollständig.

### 9.5.2 k-CLIQUE

**Gegeben:** Graph $G$ (bestimmte Anzahl und Anordnung von Knoten), Zahl $k$
**Fragestellung:** Gibt es $k$ Knoten, die alle miteinander verbunden sind? Diese Knoten bilden eine sogenannte k-CLIQUE.

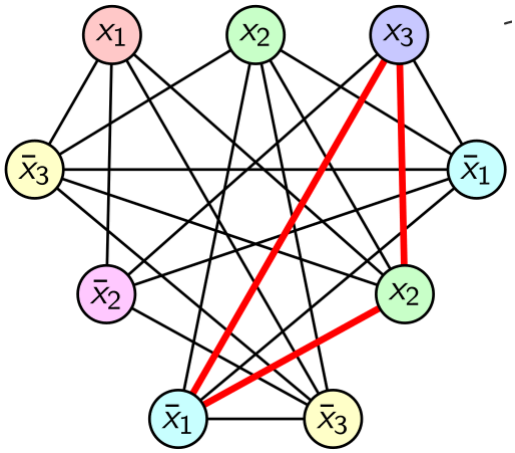


Abbildung 12: k-Clique

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Knoten (die miteinander via Kanten verbunden werden)
- Kanten
- k (Anzahl verbundener Knoten, bzw. Grösse der Clique)
- Clique (Was bildet die Clique?)

Eignet sich für Probleme, bei denen möglichst viele Elemente eine Bedingung erfüllen müssen.

### 9.5.3 SET-PACKING

**Gegeben:** Eine Familie $(S_i), i \in I$ und eine Zahl $k \in \mathbb{N}$
**Fragestellung:** Gibt es eine $k$-elementige Teilfamilie $(S_i), i \in J$ mit $J \subset I$ (also $|J| = k$) derart, dass die Menge der Teilfamilie paarweise disjunkt sind ($S_i \cap S_j = \emptyset$)?

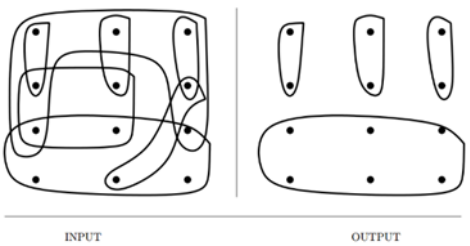


Abbildung 13: SET-PACKING

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Menge $I$ (Eigenschaft zum Vergleich von Elementen)
- Familie $S_i$ (Elemente mit Eigenschaften aus der Menge $I$)
- Teilmenge $J$ von $I$ (so dass |J| = k)
- Teilfamilie, so dass die enthaltenen Mengen unterscheidbar sind

### 9.5.4 Vertex-Cover

**Gegeben:** Graph $G$ und eine Zahl $k$.
**Fragestellung:** Gibt es eine Teilmenge von $k$ Vertizes so, dass jede Kante des Graphen ein Ende in dieser Teilmenge hat? Jeder Knoten ausserhalb des Graphen muss dementsprechend eine Kante zur Teilmenge besitzen.

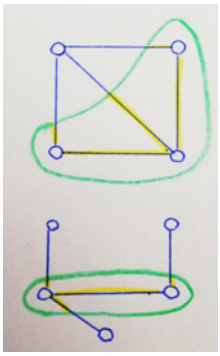


Abbildung 14: Vertex-Cover

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Knoten des Graphen
- Kanten (Verbindung zwischen den Knoten)
- $k$ (Anzahl Knoten, so dass jeder andere Knoten in dieser Teilmenge eine Kante besitzt)

### 9.5.5 Feedback-Node-Set

**Gegeben:** Gerichteter Graph $G$ und eine Zahl $k$
**Fragestellung:** Gibt es eine endliche Teilmenge von $k$ Vertizes in $G$ so, dass jeder Zyklus[2] in $G$ einen Vertex in der Teilmenge enthält?

**Info 9.5.5.1**: Gerichtete Graphen sind eine Klasse von Graphen, die keine Symmetrie zwischen den Knotenpunkten gebildeten Kanten voraussetzen.

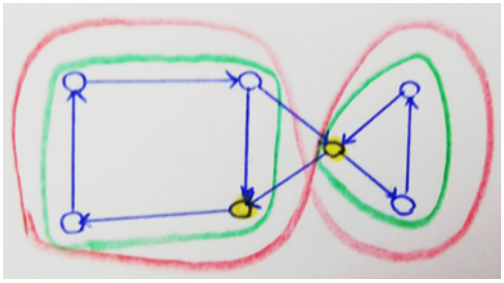


Abbildung 15: Feedback-Node-Set

**Reduktion:** Folgende Elemente müssen reduziert werden:
- gerichteter Graph
- Knoten
- Kanten
- Richtung
- Zyklen
- $k$ (Anzahl verbindende Knoten)
- Node Set (ausgewählte Knoten)

[2] Ein Zyklus ist ein sich wiederholender Ablauf

### 9.5.6 Feedback-Arc-Set

**Gegeben:** Gerichteter Graph $G$, Zahl $k$
**Fragestellung:** Gibt es eine Teilmenge von $k$ Kanten so, dass jeder Zyklus[2] in $G$ eine Kante aus der Teilmenge enthält?

Im Vergleich zum Feedback-Node-Set wird im Feedback-Arc-Set meist beschrieben, dass ein Vorgang währenddessen einer Verschiebung (auf einer Kante) gemacht wird.

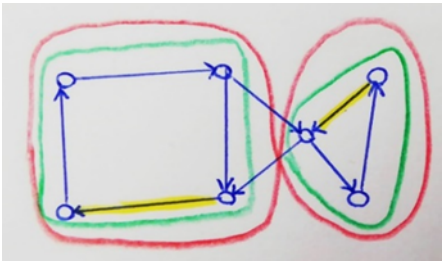


Abbildung 16: Feedback-Arc-Set

**Reduktion:** Folgende Elemente müssen reduziert werden:
- gerichteter Graph
- Knoten
- Kanten
- Richtung
- Zyklen
- k (Anzahl verbindende Kanten)
- Arc Set (ausgewählte Kanten)

### 9.5.7 Hampath (Hamiltonischer Pfad)

Das Hampath-Problem beschreibt die Suche nach einem geschlossenen Pfad in einem gerichteten Graphen, der genau einmal durch jeden Knoten geht.

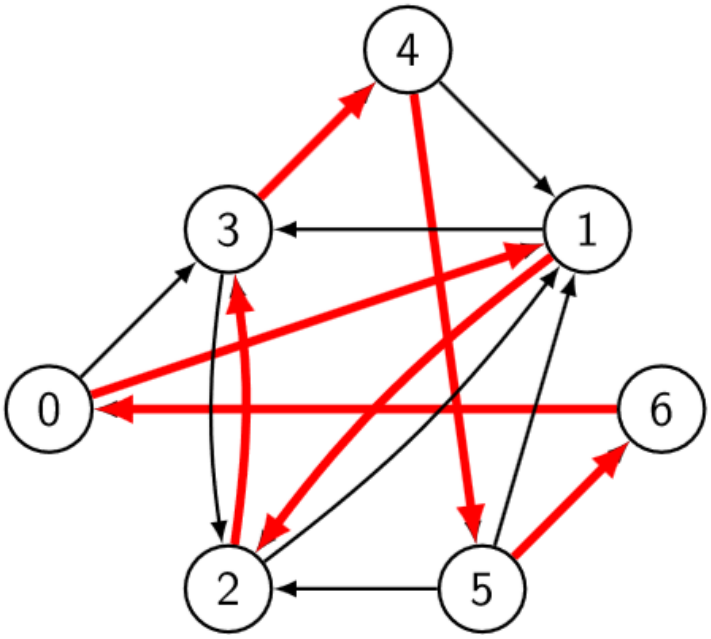



Abbildung 17: Hampath

### 9.5.8 UHampath

Das UHampath-Problem beschreibt die Suche nach einem hamiltonischen Pfad in einem **ungerichteten Graphen** zu finden.

**Info 9.5.8.1**: Bei einem ungerichteten Graphen sind die Verbindungen zwischen den Knoten (Kanten) symmetrisch.

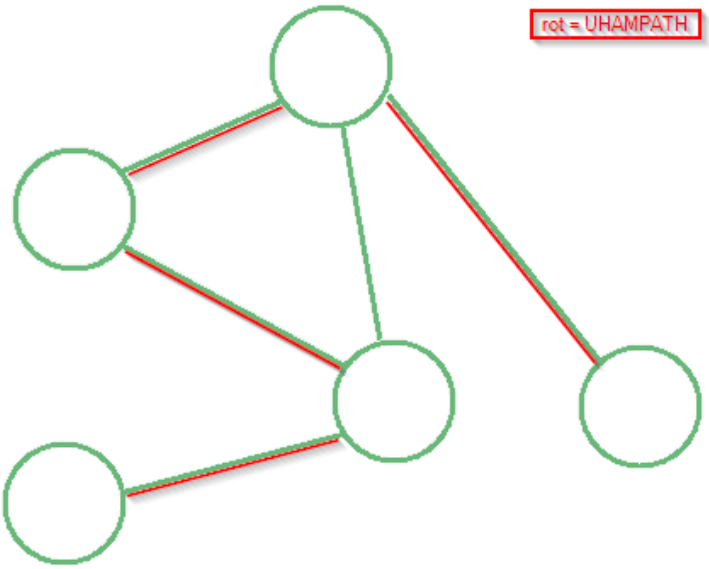



Abbildung 18: UHampath

### 9.5.9 Set-Covering

**Gegeben:** endliche Familie endlicher Mengen $(S_j)_{1\leq j\leq n}$ und eine Zahl $k$
**Fragestellung:** Gibt es eine Unterfamilie bestehend aus $k$ Mengen, die die gleiche Vereinigung hat? Kann man $k$ Teilmengen bilden, welche die Menge $S$ komplett abdecken?

Ziel ist es, alle Elemente abzudecken. Überschneidungen der Mengen sind möglich.

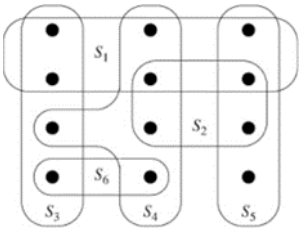



Abbildung 19: Set-Covering

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Nummerierung von 1 bis $n$
- Familie endlicher Mengen
- Unterfamilie bestehend aus $k$ Mengen
- Bedingung der beiden Vereinigungen

### 9.5.10 BIP

Das BIP-Problem (Binary Integer Programming) beschreibt, dass zu einer ganzzahligen Matrix $C$ und einem ganzzahligen Vektor $d$ ein binärer Vektor $x$ gefunden werden kann mit $C \cdot x = d$.

$$\begin{pmatrix} 1 & 3 & 0 \\ 0 & 2 & 5 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \end{pmatrix}$$

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Matrix $C$
- Vektor $d$
- Resultat für die Suche nach dem Vektor $x$ zur Erfüllung der Gleichung $C \cdot x = d$

### 9.5.11 3SAT

3SAT ist eine Variante vom SAT-Problem, wo die aussagenlogische Formel als konjunktive Normalform mit 3 Variablen pro Klausel gegeben ist. Beispiel:

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

Mit Erfüllungsequivalenz darf jede SAT-Formel in eine 3SAT-Formel umgewandelt werden.

Die Reduktion ist identisch zu Abschnitt 9.5.1 SAT.

### 9.5.12 Vertex-Coloring

Beim $k$-Vertex-Coloring-Problem sind folgende Dinge gegeben:

**Gegeben:** Graph $G$, Anzahl $k$ Farben
**Fragestellung:** Kann man die Knoten so mit $k$-Farben einfärben, dass benachbarte Knoten verschiedene Farben haben?

**Fragestellung:**

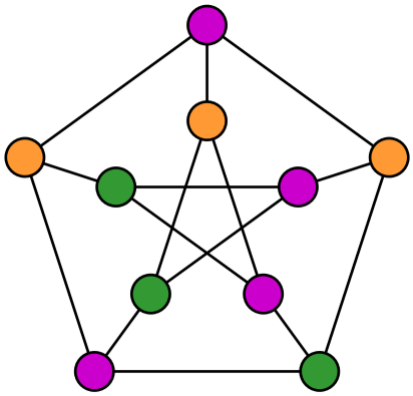


Abbildung 20: k-Vertex-Coloring

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Vertizes
- Kanten
- Farben
- $k$: Anzahl Farben

Die verbundenen Vertizes sollen alle verschiedene Farben haben:
- Die Farben stellen den (gesuchten) Unterschied zwischen den Vertizes dar (eine unterscheidbare Eigenschaft)
- Die Kanten zwischen den Vertizes stellen die Regeln für die unterscheidbaren Objekte dar.

Das Vertex-Coloring-Problem eignet sich für Probleme, bei denen Elemente, die miteinander in Beziehung stehen, unterschieden werden müssen.

### 9.5.13 Clique-Cover

**Gegeben:** Graph $G$, positive Zahl $k$
**Fragestellung:** Gibt es $k$ Cliquen so, dass jede Ecke in genau einer der Cliquen ist?

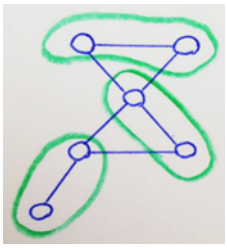


Abbildung 21: Clique-Cover

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Vertizes
- Kanten
- $k$ (Anzahl Vertizes)
- Clique (verbund von möglichst vielen Vertizes)
- Covering (Bedingung)

### 9.5.14 Exact-Cover

**Gegeben:** Eine Familie $(S_j)_{1\leq j\leq n}$ von Teilmengen einer Menge $U$.
**Fragestellung:** Gibt es eine Unterfamilie von Mengen, die disjunkt sind, aber dieselbe Vereinigung haben? Die Unterfamilie $(S_{j_i})_{1\leq i\leq m}$ muss also $S_{ji} \cap S_{jk} = 0$ und $\bigcup_{j=1}^{n} S_j = \bigcup_{i=1}^{m} S_{ji}$ erfüllen.

Jedes Element in $U$ soll genau in einer der Teilmengen einer Familie $S$ vorkommen. Die gesuchte Menge bildet eine exakte Überdeckung. Es darf keine Überschneidungen geben, aber alle Elemente sollen abgedeckt werden.

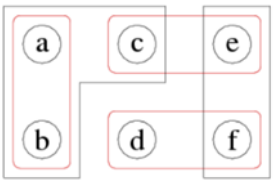



Abbildung 22: Exact-Cover

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Menge $U$
- Familie $S_j \subset U$
- Unterfamilie $S_{ji}$ (so, dass die gleiche Vereinigung wie die Familie $S_j$ erzielt wird)
- Bedingung $S_{ji} \cap S_{jk} = 0$
- Bedingung $\bigcup_{j=1}^{n} S_j = \bigcup_{i=1}^{m} S_{ji}$

### 9.5.15 3D-Matching

**Gegeben:** Endliche Menge $T$ und Menge $U$ von Tripeln $T^3$.
**Fragestellung:** Gibt es eine Teilmenge $W$ von $U$ so, dass $|W| = |T|$ und keine zwei Elemente von $W$ in irgendeiner Koordinate übereinstimmen?

Für jeden Wert im Tripel $(x, y, z)$ gibt es eine bestimmte Bedeutung abhängig vom Kontext.

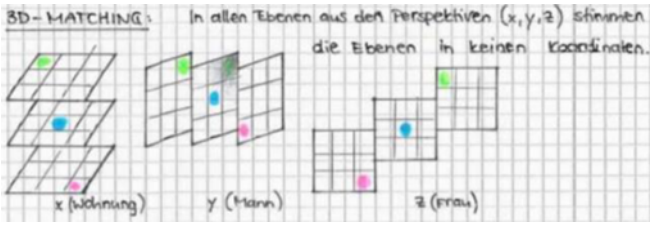



Abbildung 23: 3D-Matching

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Menge $T$ (normalerweise eine bestimmte Anzahl $n$ von Elementen)
- Tripel aus $T^3$
- Menge $U$ (bestehend aus Kombinationen der Tripel)
- Teilmenge $W$ von $U$ so, dass $n = |W| = |T|$ (jedes Tripel muss in jedem Element für $x, y, z$ eindeutig sein)

*Beispiel*:
- $T = \{0, 1\}$
- $U = \{(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1$
- $W = \{(1,0,1), (0,1,0)\}$ erfüllt die Bedingung

### 9.5.16 Steiner-Tree

**Gegeben:** Ein Graph $G$, eine Teilmenge $R$ von Verizes, eine Gewichtsfunktion und eine positive Zahl $k$.
**Fragestellung:** 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.

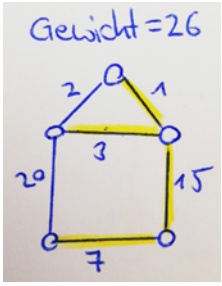


Abbildung 24: Steiner-Tree

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Vertizes
- Kanten (im Fall X)
- Gewichtsfunktion (zum Vergleich der Kanten und Wahl des Baumes)
- Knoten in $R$ (bestimmte Auswahl von Vertizes aufgrund einer Bedingug)
- maximales Gewicht $k$ (so, dass es sich noch «lohnt»)
- Baum (der die einzelnen Vertizes schliesslich verbinden soll)

#### 9.5.17 Hitting-Set

**Gegeben:** Menge von Teilmengen $S_i \subset S$
**Fragestellung:** Gibt es eine Menge $H$, die jede Menge in genau einem Punkt trifft $\left(H \subset \bigcup_{i \in J} S_i\right)$, also $|H \cap S_i| = 1$?

*Beispiel*:
- Gegeben: $A = \{1, 2, \mathbf{3}\}, B = \{1, 2, \mathbf{4}\}, C = \{1, 2, \mathbf{5}\}$
- Gesucht: $H = \{3, 4, 5\}$

**Reduktion:** Folgende Elemente müssen reduziert werden:
- $i$ (Bedingung/Merkmal zur eindeutigen Auswahl eines Elements)
- Menge $S_i$ (Menge von Teilmengen)
- Menge $H$ (Resultat mit Bedingung $|H \cap S_i| = 1$)

#### 9.5.18 Subset-Sum

**Gegeben:** Menge $S$ von ganzen Zahlen
**Fragestellung:** Kann man eine Teilmenge finden, die als Summe einen bestimmten Wert $t$ hat?

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Elemente, zwischen denen entschieden werden muss
- Menge $S$ von Elementen
- bestimmter Wert $t$
- Teilmenge $T$, so dass die Werte der Elemente dem Wert $t$ entspricht: $\sum_{x \in T} x = t$

Der Name «Rucksack»-Problem rührt daher, dass man sich die Zahlen aus $S$ als «Grösse» von Gegenständen vorstellt, und wissen möchte, ob man einen Rucksack der Grösse $t$ exakt füllen kann mit einer Teilmenge von Gegenständen aus $S$.

#### 9.5.19 Sequencing

**Gegeben:** Ein Vektor von Laufzeiten von $p$ Jobs, ein Vektor von spätesten Ausführzeiten, ein Strafenvektor und eine positive ganze Zahl $k$.
**Fragestellung:** Gibt es eine Permutation der Zahlen $1, ..., p$, sodass die Gesamtstrafe für verspätete Ausführung bei der Ausführung der Jobs nicht grösser ist als $k$?

Vereinfachte Definition:

**Gegeben:** Eine Menge von Jobs, pro Job eine Ausführzeit, Deadline und eine Strafe sowie eine maximale Strafe $k$. Die Jobs müssen sequenziell abgearbeitet werden. Wird ein Job zu spät fertig, muss eine Strafe gezahlt werden.
**Fragestellung:** Gibt es eine Reihenfolge von Jobs so, dass die Strafe kleiner gleich $k$ ist?

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Ausführungszeit von Job
- Deadline
- Strafe
- Permutation/Reihenfolge
- Zusammensetzung der Gesamtstrafe

#### 9.5.20 Partition

**Gegeben:** Eine Folge von $n$ ganzen Zahlen $c_1, c_2, ...c_n$
**Fragestellung:** Kann man die Indizes $1, 2, ..., n$ in zwei Teilmengen $I$ und $\overline{I}$ teilen, so dass die Summe der zugehörigen Zahlen $c_i$ identisch ist ($\sum_{i \in I} c_i = \sum_{i \in \overline{I}} c_i$)? Gibt es zwei disjunktive Teilmengen mit derselben Summe?

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Indizes $i$ (konkretes Objekt zum Vergleich/zur Aufteilung)
- Werte der Vergleichsobjekte $c_i$
- Aufteilung in zwei Teilmengen $I$ und $\overline{I}$ so, dass der Wert der entsprechenden Vergleichsobjekte identisch ist

*Beispiel*: Eine Reihe von Wassergläsern ist unterschiedlich gefüllt. Es sollen 2 Behählter gleich voll mit den Gläsern gefüllt werden. Welche Gläser müssen in welche Behälter geleert werden?

#### 9.5.21 Max-Cut

**Gegeben:** Graph $G$ mit einer Gewichtsfunktion $w : E \to Z$ und eine ganze Zahl $W$.
**Fragestellung:** Gibt es eine Teilmenge $S$ der Vertizes, so dass das Gesamtgewicht der Kanten, die $S$ mit seinem Komplement verbinden, so gross wie $W$?

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

Der Max-Cut eines Graphen ist eine Zerlegung seiner Knotenmenge in zwei Teilmengen, sodass das Gleichgewicht der Zwischen den beiden Teilen verlaufenden Kanten mindestens $W$ wird.

Max-Cut sucht die maximalen Investitionen, die man in den Sand setzen muss, indem man eine Menge von Verbindungen durchschneidet.

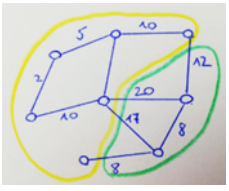


Abbildung 25: Max-Cut

**Reduktion:** Folgende Elemente müssen reduziert werden:
- Vertizes
- Kanten
- Gewichtsfunktion der Kante (zum Vergleich der Kanten)
- Wert $W$ (Ziel, das bei der Wahl der Teilmenge $S$ der Vertizes überstiegen werden sollte)
- Gesamtgewicht
- Teilmenge $S$ der Vertizes (Trennlinie zwischen zwei Gruppierungen)

# 10 Turing-Vollständigkeit

## 10.1 Die universelle Turing-Maschine

Gibt es eine TM, die jede beliebige TM simulieren kann?

Charles Babbage, Ada Lovelace: Analytical Engine

Alan Turing: Es gibt eine Turing-Maschine, die jede beliebige andere Turing-Maschine simulieren kann[3]:

- Eigenes Band für die 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

## 10.2 Programmiersprachen und Turing-Vollständigkeit

**Definition 10.2.1**: Eine Programmiersprache heisst Turing-vollständig, wenn in ihr jede beliebige Turing-Maschine siumuliert werden kann.

Frage: Gibt es einen in $A$ geschriebenen Turing-Maschinen-Simulator?

## 10.3 Turing-Vollständigkeit von Programmiersprachen

### 10.3.1 LOOP

Führe ein Programm $P$ genau $x_i$ mal aus:

```
LOOP x_i DO
  P
END
```

Daraus kann eine if-Kontrollstruktur erstellt werden:

```
IF x_i THEN
  P
END
```

wird folgendermassen realisiert:

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

### 10.3.2 Turing-Vollständigkeit

Für Turing-Vollständigkeit wird neben `LOOP` noch eine `GOTO`-Struktur benötigt.

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