# Automaten und Sprachen | AutoSpr

## Zusammenfassung

## INHALTSVERZEICHNIS


1. Vorwort ... 4
2. Prädikate ... 4
   - 2.1. Normalformen ... 4
   - 2.2. Negation ... 4
3. Mengen ... 5
   - 3.1. Konstruktion ... 5
   - 3.2. Operationen ... 5
   - 3.3. Tupel ... 5
4. Beweise ... 5
5. Sprachen ... 5
   - 5.1. Wortlänge ... 6
   - 5.2. Deterministische Endliche Automaten (DEA) ... 7
   - 5.3. Wörter einer regulären Sprache ... 7
     - 5.3.1. Übergangsfunktionen für Wörter ... 7
     - 5.3.2. Wort akzeptieren ... 7
     - 5.3.3. Akzeptierte Sprache, reguläre Sprache ... 7
   - 5.4. Myhill-Nerode Automat ... 7
   - 5.5. Minimalautomat ... 8
   - 5.6. Pumping Lemma ... 8
     - 5.6.1. Beweis ... 9
   - 5.7. Nichtdeterministische endliche Automaten (NEA) ... 9
     - 5.7.1. Umgang mit mehrdeutigen Übergängen ... 10
     - 5.7.2. Thompson-NEA ... 10
     - 5.7.3. $NEA_\varepsilon$ ... 11
   - 5.8. Mengenoperationen ... 11
   - 5.9. Reguläre Operationen ... 13
     - 5.9.1. Alternative ... 13
     - 5.9.2. Verkettung ... 13
     - 5.9.3. *-Operation ... 13
   - 5.10. Reguläre Ausdrücke ... 13
     - 5.10.1. Verallgemeinerter NEA (VNEA) ... 13
     - 5.10.2. Teststrategie ... 15
   - 5.11. Kontextfreie Sprachen ... 15
     - 5.11.1. Kontextfreie Grammatik und Sprache ... 15
     - 5.11.2. Reguläre Operationen ... 16
     - 5.11.3. Kontextfrei ... 16
     - 5.11.4. Chomsky-Normalform (CNF) ... 16
6. Parsing ... 17
   - 6.1. Cocke-Younger-Kasami Algorithmus (CYK) ... 17
     - 6.1.1. Ableitungsdreieck ... 18

**6.2. Backus-Naur-Form (BNF) ... 18**
**6.3. Extended Backus-Naur-Form (EBNF) ... 19**
**6.4. Rechtsrekursion ... 19**
**7. Stack ... 20**
**7.1. Stackautomat / Pushdown Automaton (PDA) ... 20**
7.1.1. Ableitung aus CNF ... 20
7.1.2. PDA $\rightarrow$ CFG ... 20
**8. Nicht kontextfreie Sprachen ... 21**
**8.1. Pumping Lemma für CFL ... 21**
**9. Unendlich ... 22**
**10. Turing-Maschine (TM) ... 23**
**10.1. Arbeitsweise ... 23**
**10.2. Protokollierung ... 24**
**10.3. Varianten ... 25**
10.3.1. Nichtdeterministische TM ... 26
10.3.2. Verschiedene Bandalphabete ... 26
10.3.3. Mehrspurmaschine ... 26
10.3.4. Mehrbandmaschine ... 26
10.3.5. Einseitig unendliches Band ... 26
**10.4. Turing-erkennbare Sprachen ... 26**
10.4.1. Aufzähler ... 27
10.4.2. Entscheider und entscheidbare Sprachen ... 27
**11. Entscheidbarkeit ... 27**
**11.1. Entscheider ... 27**
**11.2. Akzeptanzproblem für Turing-Maschinen ... 27**
**11.3. Halteproblem ... 28**
**11.4. Reduktion ... 28**
**11.5. Satz von Rice ... 28**
**12. Komplexität ... 28**
**12.1. Hardwareeinfluss ... 29**
**12.2. Polynomielle und Exponentielle Laufzeit ... 29**
**12.3. Sprachen ... 29**
**12.4. Polynomielle Verifizierer ... 29**
**12.5. Polynomielle Reduktion ... 29**
12.5.1. Polynomieller Laufzeit-Vergleich ... 29
12.5.2. Satz ... 29
**12.6. Polynomielle Ausfüllrätsel ... 30**
**12.7. NP-Vollständigkeit ... 30**
12.7.1. Cook-Levin ... 31
12.7.2. $SAT \leq_P 3SAT$ ... 31
12.7.3. Karp-Katalog ... 31
**13. Turing-Vollständigkeit ... 39**
**13.1. Dinge, die einer TM fehlen ... 39**
**13.2. Universelle Turing-Maschine ... 39**
**13.3. Programmiersprachen und Turing-Vollständigkeit ... 40**
**13.4. Grundelemente für Sprachen ... 40**
**13.5. LOOP ... 40**

13.6. WHILE ..... 40
13.7. GOTO ..... 40
Bibliografie ..... 41

# 1. VORWORT

Diese Zusammenfassung basiert auf dem Buch «Automaten und Sprachen» [1] , geschrieben von Prof. Dr. Andreas Müller.

# 2. PRÄDIKATE

Prädikate sind Aussagen über mathematische Objekte, die wahr oder falsch sein können. «Funktionen» mit booleschen Rückgabewerten: $P, Q(n), R(x, y, z)$.

$(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$ $(A \Leftrightarrow B) \Leftrightarrow (A \wedge B) \vee (\neg A \wedge \neg B)$ $A \vee (\neg A \wedge B) \Leftrightarrow A \vee B$

$(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$ $\neg(A \Rightarrow B) \Leftrightarrow A \wedge \neg B$ $(A \Rightarrow B \Rightarrow C) \Leftrightarrow (A \Rightarrow B) \wedge (B \Rightarrow C)$

| ***Begriff*** | ***Definition*** |
|---|---|
| Abtrennungsregel | $(A \wedge (A \Rightarrow B)) \Rightarrow B$ |
| Kommutativität | $(A \wedge B) \Leftrightarrow (B \wedge A)$<br>$(A \vee B) \Leftrightarrow (B \vee A)$ |
| Assoziativität | $A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C$<br>$A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee C$ |
| Distributivität | $A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$<br>$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$ |
| Absorption | $A \vee (A \wedge B) \Leftrightarrow A$<br>$A \wedge (A \vee B) \Leftrightarrow A$ |
| Idempotenz | $A \vee A = A$<br>$A \wedge A = A$ |
| Doppelte Negation | $\neg(\neg A) \Leftrightarrow \neg\neg A \Leftrightarrow A$ |
| Konstanten | $W = \text{wahr}$<br>$F = \text{falsch}$ |
| de Morgan | $\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$<br>$\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$ |

## 2.1. NORMALFORMEN

Normalformen (allgemein, ***kanonische Formen***) helfen, das Vergleichsproblem zu lösen (ob zwei Aussagen dieselben sind).

$$P_1 \wedge ... \wedge P_n = \forall i \in \{1, ..., n\}.\ (P_i) = \bigwedge_{i=1}^{n} P_i$$

$$P_1 \vee ... \vee P_n = \exists i \in \{1, ..., n\}.\ (P_i) = \bigvee_{i=1}^{n} P_i$$

## 2.2. NEGATION

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

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

# 3. MENGEN

## 3.1. KONSTRUKTION

Grundmenge $G$, Prädikat $P(x)$. Konstruierte Menge $A = \{x \in G \mid P(x)\}$

## 3.2. OPERATIONEN

| *Begriff* | *Definition* |
|---|---|
| Vereinigung | $A \cup B = \{x \mid x \in A \vee x \in B\}$ |
| Schnittmenge | $A \cap B = \{x \mid x \in A \wedge x \in B\}$ |
| Komplement | $\overline{A} = \{x \mid x \notin A\}$ |
| Differenz | $A \setminus B = \{x \in A \mid x \notin B\}$ |

## 3.3. TUPEL

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

$n$-Tupel:

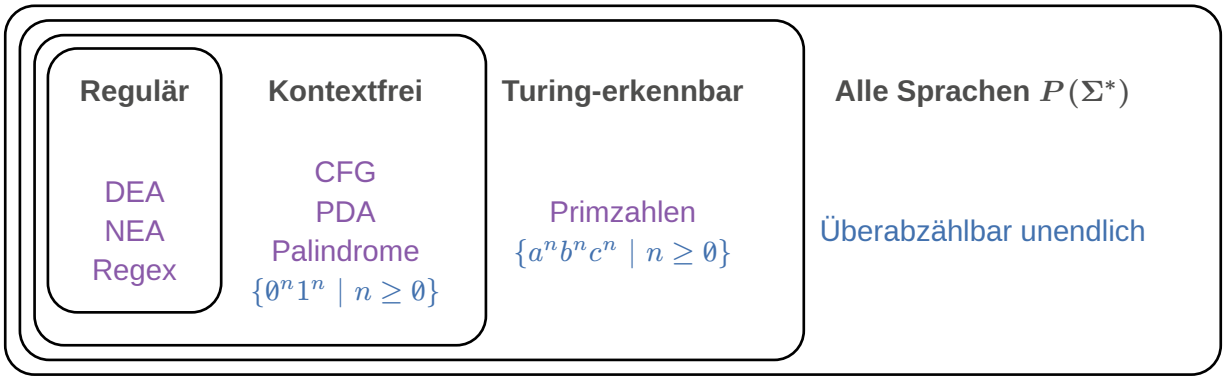
$$\bigtimes_{i=1}^{n} A_i = \{(a_1, a_2, ..., a_n) | a_i \in A_i\}$$

# 4. BEWEISE

Eine Folge von logischen Schlüssen, die zeigen, dass die Aussage aus den gegebenen Voraussetzungen folgt.

| *Beweistechnik* | *Anwendungsfall* |
|---|---|
| Konstruktiver Beweis | Liefert explizite «Lösung» / Algorithmus |
| Widerspruchsbeweis | Geeignet für Unmöglichkeitsaussagen. Z.B. $\sqrt{2} \notin \mathbb{Q}$ |
| Vollständige Induktion | Für Aussagen, die für alle $n \in \mathbb{N}$ gelten. |

# 5. SPRACHEN



| **Begriff** | **Definition** |
|---|---|
| Alphabet | Eine nichtleere endliche menge $\Sigma$ heisst ***Alphabet***. Die Elemente von $\Sigma$ heissen ***Zeichen***. |
| Wort | Eine Zeichenkette der Länge $n$ ist ein $n$-Tupel in $\Sigma^n = \Sigma \times ... \times \Sigma$. Ein Element von $\Sigma^n$ heisst ***Wort*** der Länge $n$. Wörter haben immer endliche Länge. |
| Leeres Wort | Die Zeichenkette $\varepsilon \in \Sigma^0 = \{\varepsilon\}$ der Länge $0$ heisst das ***leere Wort***. |
| Menge aller Wörter | Leeres Wort ist **immer** drin $$\Sigma^* = \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup ... = \bigcup_{k=0}^{\infty} \Sigma^k$$ |
| Abgekürzte Schreibweisen von Wörtern | $$(A,u,t,o,S,p,r) = AutoSpr$$ $$a^3b^5 = aaabbbbb$$ $$b^5a^3 = bbbbbaaa$$ |
| Sprache | Teilmenge $L \subset \Sigma^*$ |

## 5.1. WORTLÄNGE

| **Begriff** | **Definition** |
|---|---|
| Länge des Wortes | $w \in \Sigma^n \Rightarrow \lvert w\rvert = n$, $n$ ist ***Länge des Wortes*** $w$ |
| Anzahl Zeichen | Sei $w \in \Sigma^n, a \in \Sigma$. Dann ist $\lvert w\rvert_a$ die ***Anzahl Zeichen*** $a$ im Wort $w$ |

***Beispiel** 1:* **Wortlänge**

$$|\varepsilon| = 0 \qquad |01010|_1 = 2 \qquad |a^3b^5| = 8$$
$$|01010|_0 = 3 \qquad |01010|_3 = 0 \qquad |(1291)^7| = 7|1291| = 7 \cdot 4 = 28$$

***Beispiel** 2:* **Sprache**

$$L = \Sigma^* \subset \Sigma^*$$
$$L = \varnothing \subset \Sigma^*, \text{die leere Sprache}$$
$$\Sigma = \{0,1\}, \text{Sprache aller Binärstrings: } L = \Sigma^*$$
$$\Sigma = \text{Unicode}, J = \{w \in \Sigma^* \mid w \text{ wird vom Java-Compiler akzeptiert}\}$$
$$M \text{ eine "Maschine"}, L(M) = \{w \in \Sigma^* \mid w \text{ wird von } M \text{ akzeptiert}\}$$

***Beobachtungen** 1:*

1. $|w^n| = n \cdot |w|$
2. Zu jeder Maschine $M$ gibt es eine Sprache $L(M)$

## 5.2. DETERMINISTISCHE ENDLICHE AUTOMATEN (DEA)

***Definition***

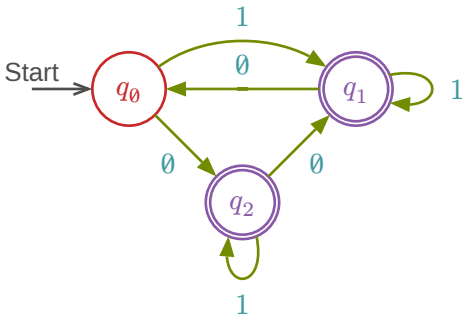
$$A = (Q, \Sigma, \delta, q, F)$$

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

***Tabellenform***

| | | 0 | 1 | |
|---|---|---|---|---|
| $Q$ | $q_0$ | $q_2$ | $q_1$ | $\Sigma$ |
| $F$ | $q_1$ | $q_0$ | $q_1$ | $\delta$ |
| | $q_2$ | $q_1$ | $q_2$ | |

***Zustandsdiagramm von $A$***




**Wichtig:** Ein Pfeil für jedes Zeichen in jedem Zustand für validen DEA

## 5.3. WÖRTER EINER REGULÄREN SPRACHE

### 5.3.1. Übergangsfunktionen für Wörter

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

***Übergänge*** ausghend von $q$ für Zeichen $a_1, ..., a_n$ nacheinander anwenden.

### 5.3.2. Wort akzeptieren

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

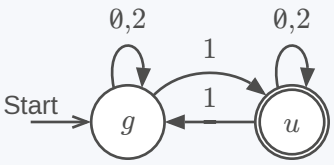
### 5.3.3. Akzeptierte Sprache, reguläre Sprache

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

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

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

***Beispiel 3:*** **DEA, der die Sprache $L = \{w \in \Sigma^* \mid w \text{ ist eine ungerade Zahl im Dreiersystem}\}$ akzeptiert**




| | 0 | 1 | 2 |
|---|---|---|---|
| g | g | u | g |
| u | u | g | u |

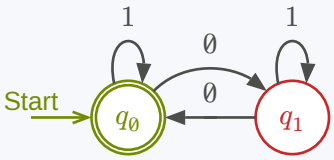
## 5.4. MYHILL-NERODE AUTOMAT

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

Gegeben: reguläre Sprache $L$ über $\Sigma$. Rekonstruiere $A$ mit:
- $Q = \{L(w) \mid w \in \Sigma^*\}$
- $q_0 = L(\sigma) = L$
- $F = \{L(w) \in Q \mid \varepsilon \in L(w)\}$
- $\delta(L(w), a) = L(wa)$

Gleicher Zustand $\Leftrightarrow$ gleiches $L(w)$

**Beispiel 4:** $\Sigma = \{0,1\}, L = \{w \in \Sigma^* \mid |w|_0 \text{ gerade}\}$



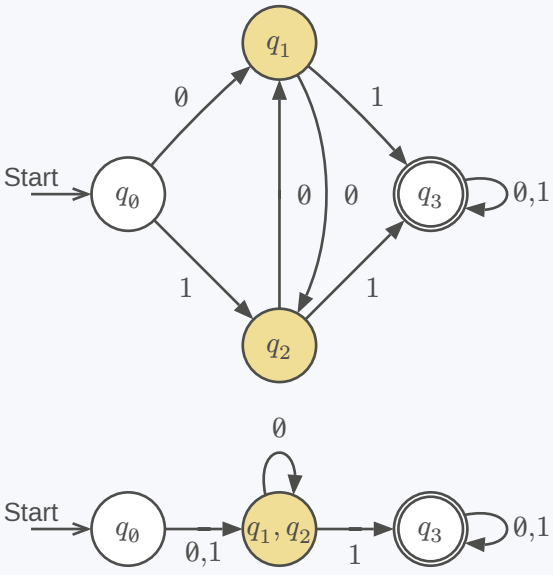

| $w$ | $L(w)$ | $Q$ |
|---|---|---|
| $\varepsilon$ | $L(\varepsilon) = L$ | $q_0$ |
| 0 | $L(0) = \{w \in \Sigma^* \mid \lvert w\rvert_0 \text{ ungerade}\}$ | $q_1$ |
| 1 | $L(0) = \{w \in \Sigma^* \mid \lvert w\rvert_0 \text{ gerade}\}$ | $q_0$ |
| ⋮ | ⋮ | ⋮ |

## 5.5. MINIMALAUTOMAT

**Ziel:** Nicht unterscheidbare Zustände zusammenlegen

**Vorgehen:**
1) Akzeptierzustand $\neq$ Nichtakzeptierzustand
2) Iteration: alle unterscheidbaren Paare suchen
3) Verbleibende Paare sind ununterscheidbar $\rightarrow$ zusammenlegen

**Beispiel 5:**




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

***Algorithmus***

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

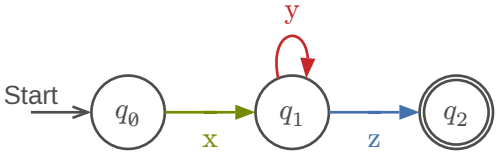
$\Rightarrow q_1$ und $q_2$ sind nicht unterscheidbar

## 5.6. PUMPING 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) $\forall k \in \mathbb{N}.\ (xy^k z \in L)$

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



Was zeichnet reguläre Sprachen aus?

– Reguläre Ausdrücke
– Lange Wörter: * kommt im regulären Ausdruck vor
– Blöcke mit * können beliebig oft wiederholt werden
– ⇒ Wörter können «aufgepumpt» werden

### 5.6.1. Beweis

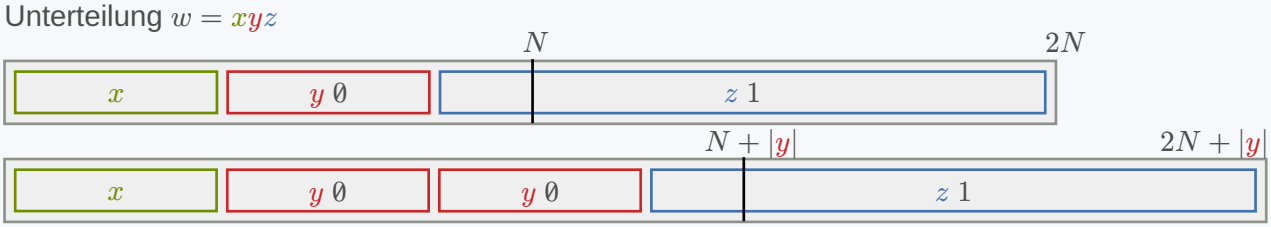
Behauptung: Die Sprache $L \subset \Sigma^*$ ist nicht regulär.

Beweis (Widerspruch):

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

***Beispiel** 6:* $L = \{0^n 1^n \mid n \geq 0\}$ **ist nicht regulär**

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




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

## 5.7. NICHTDETERMINISTISCHE ENDLICHE AUTOMATEN (NEA)

### *Definition*

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

– Endliche Menge von Zuständen: $Q$
– Alphabet: $\Sigma$
– Übergangsfunktion: $\delta : Q \times \Sigma \to P(Q)$
– Startzustand: $q \in Q$
– Akzeptierzustände: $F \subset Q$

### *Akzeptieren*

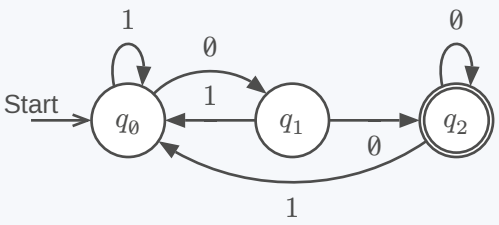
Ein NEA $A$ ***akzeptiert*** das Wort $w \in \Sigma^*$, wenn es eine **Wahl** von Übergängen gibt, derart, dass das Wort $w$ den Automaten in einen Akzeptierzustand überführt.

### *Faustregeln*

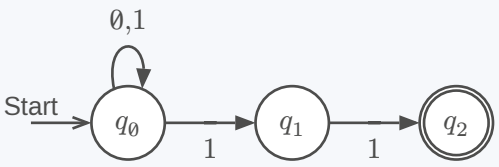
– Nur genau diejenigen Pfeile einzeichnen, die man zum Akzeptieren braucht.
– Erlaube Alternativen

***Beispiel 7:*** $L = \{w \in \Sigma^* \mid w \text{ endet mit zwei } 0\}$

***DEA-Lösung***




***NEA-Lösung***

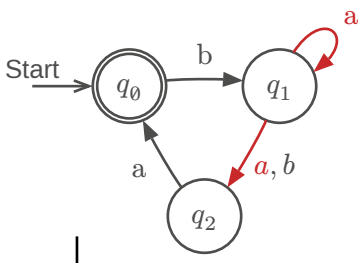



### 5.7.1. Umgang mit mehrdeutigen Übergängen

a-Übergang von $q_1$ aus nicht eindeutig

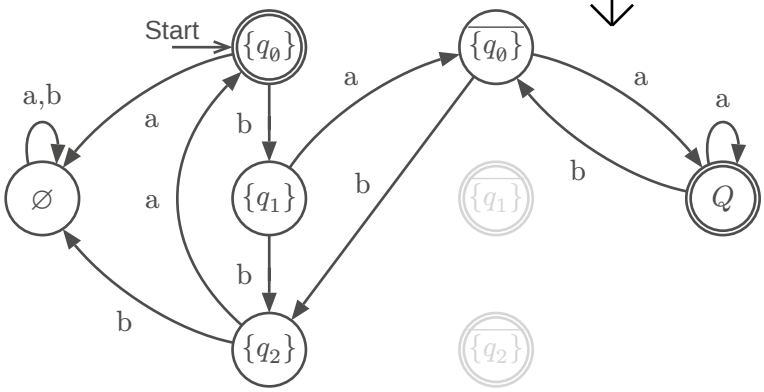
Welchen Übergang soll man nehmen?

- Beide Möglichkeiten probieren und akzeptieren, falls eine der Möglichkeiten auf einen Akzeptierzustand führt.
- Zufallsgenerator (probabilistischer endlicher automat, PEA)




### 5.7.2. Thompson-NEA

- Zustand = **Menge** der möglichen Zustände
- Akzeptieren = Ob NEA im Zustand sein **könnte**




#### 5.7.2.1. Transformation NEA → DEA

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

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

***Satz***

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

- $Q' = P(Q)$
- $\Sigma' = \Sigma$
- $\delta'$ wie oben definiert
- $q'_0 = \{q_0\}$
- $F' = \{M \in P(Q) \mid F \cap M \neq \varnothing\}$

***Implementation***

Thompson-NEA:

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

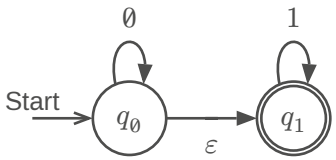
### 5.7.3. $NEA_\varepsilon$

| Begriff | Definition |
|---|---|
| $\varepsilon$-Übergang | Kann ohne Verarbeitung eines Zeichens genommen werden. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$ |
| $\mathrm{NEA}_\varepsilon$ | NEA mit $\varepsilon$-Übergängen |

Einen $\mathrm{NEA}_\varepsilon$ kann man immer in einen NEA umwandeln. Beweis:

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

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

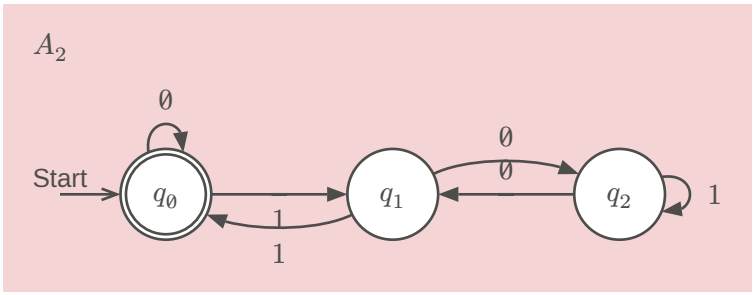


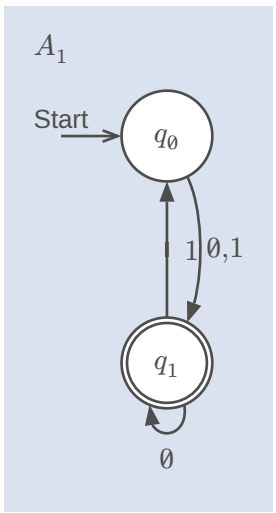

## 5.8. MENGENOPERATIONEN

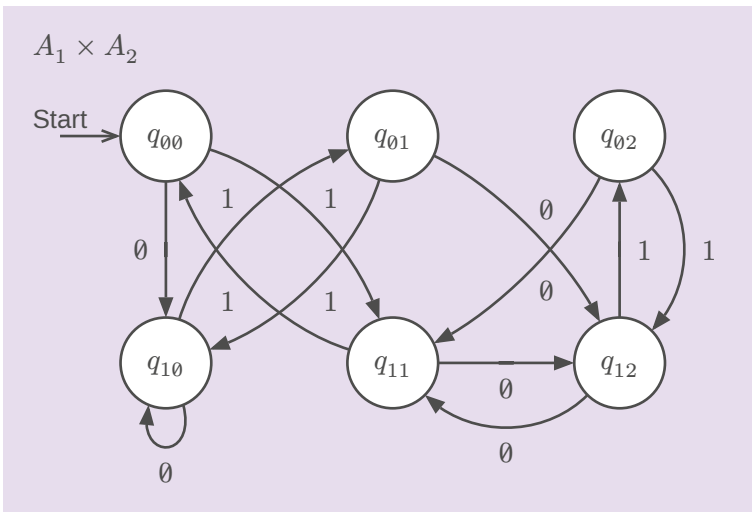
Lassen reguläre Sprachen regulär sein.

Produktautomat: $L_i = L(A_i)$

$L = \{w \in \Sigma^* \mid |w|_1 \text{ ungerade}\} \cap \{w \in \Sigma^* \mid w \text{ ist eine durch drei teilbare Binärzahl}\}$

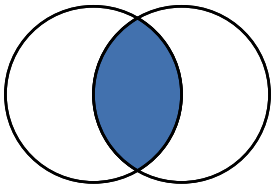


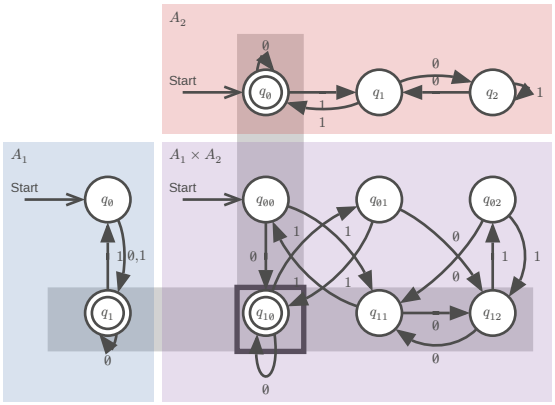



Schnittmenge $L(A_1) \cap L(A_2)$

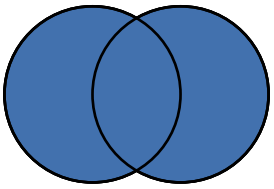
$A_1 \cap A_2$

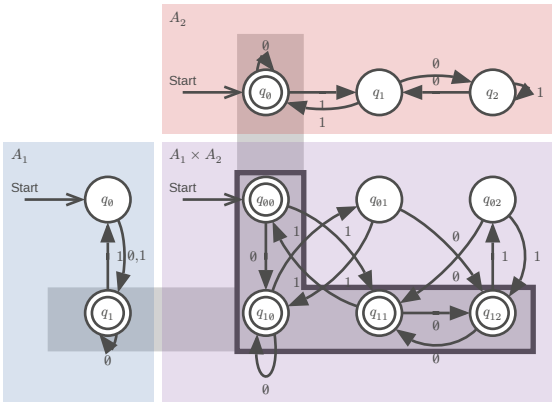





Vereinigungsmenge $L(A_1) \cup L(A_2)$

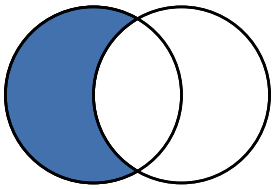
$A_1 \cup A_2$

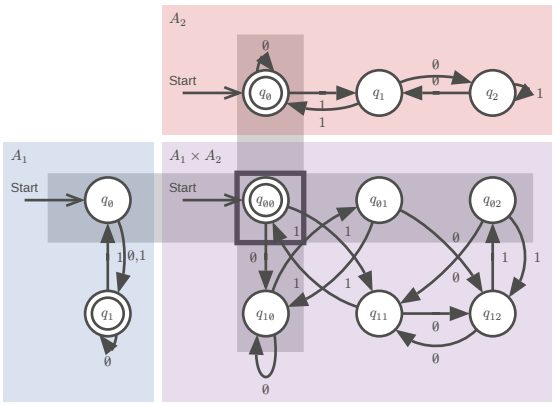





Differenzmenge $L(A_1) \setminus L(A_2)$

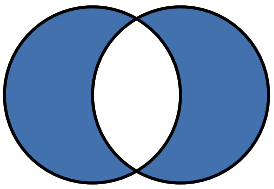
$A_1 \setminus A_2$

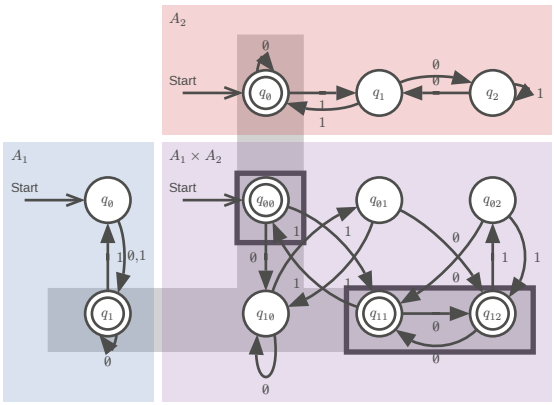





Symmetrische Differenz $L(A_1) \triangle L(A_2)$

$A_1 \triangle A_2$






Operationen verändern nur Endzustände:

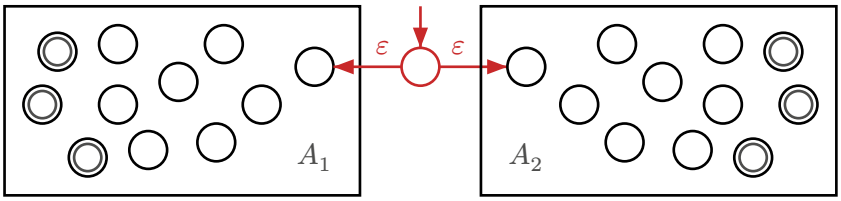
| ***Begriff*** | ***Definition*** |
|---|---|
| Schnittmenge $L(A_1) \cap L(A_2)$ | $F = F_1 \times F_2$ |
| Vereinigungsmenge $L(A_1) \cup L(A_2)$ | $F = F_1 \times Q_2 \cup Q_1 \times F_2$ |
| Differenzmenge $L(A_1) \setminus L(A_2)$ | $F = F_1 \times (Q_2 \setminus F_2)$ |
| Symmetrische Differenz $L(A_1) \triangle L(A_2)$ | $F = F_1 \times (Q_2 \setminus F_2) \cup (Q_1 \setminus F_1) \times F_2$ |

## 5.9. REGULÄRE OPERATIONEN

WICHTIG: $\varepsilon$-Übergänge immer hinzufügen

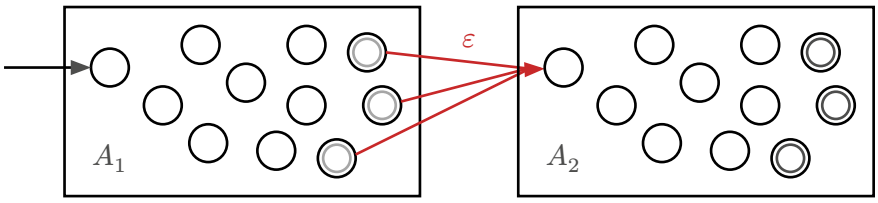
### 5.9.1. Alternative

$$\begin{aligned} L &= L_1 \cup L_2 \\ &= L(A_1) \cup L(A_2) \\ &= L(A_1) \mid L(A_2) \end{aligned}$$



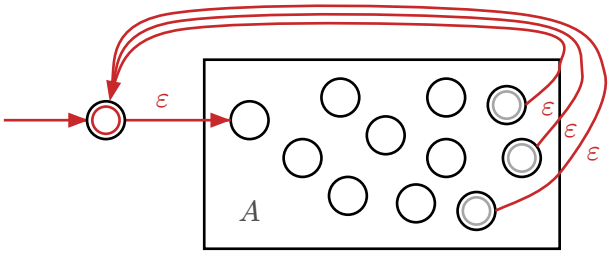

### 5.9.2. Verkettung

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




### 5.9.3. *-Operation

$$\begin{aligned} L^* &= \{\varepsilon\} \cup L \cup L^2 \cup ... \\ &= \bigcup_{k=0}^{\infty} L^k \end{aligned}$$




## 5.10. REGULÄRE AUSDRÜCKE

Formeln, die Zeichenketten beschreiben.

1) Buchstaben stehen für sich selbst, mit Ausnahme der Metazeichen $()[]*?\mid . \setminus$ (escape character)
2) Verkettung: Zeichen und Formeln hintereinanderschreiben
3) . = ein beliebiges Zeichen, $\Sigma$
4) |: Alternative, $a|A$ = a oder A
5) *: Wiederholung, beliebige viele
6) $()$: Gruppierung
7) [] Zeichenklassen: $[abc] = a|b|c$, $[\hat{}abc]$ = alles ausser a, b, oder c

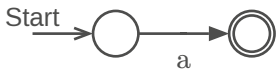
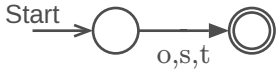
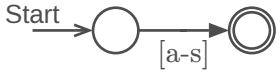
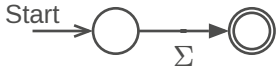
Erweiterungen / Dialekte

1) $\{n, m\}$: zwischen $n$ un $m$ Wiederholungen
2) $+$: mindestens eines, $\{1,\}$
3) $?$: optional, $\{0, 1\}$
4) Symbole für Zeichenklassen: $\setminus d$ Ziffern, $\setminus s$ whitespace, $[: space :]$, $[: lower :]$

### 5.10.1. Verallgemeinerter NEA (VNEA)

| *Begriff* | *Definition* |
|---|---|
| Regulärer Ausdruck | Zeichenkette $r$ zur Beschreibung einer regulären Sprache $L = L(r)$ |

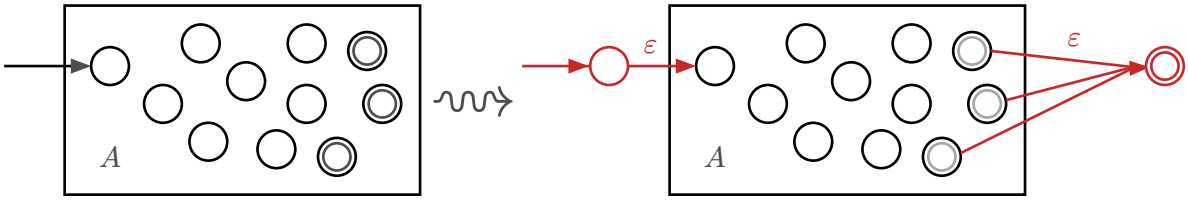
| **Begriff** | **Definition** |
|---|---|
| Reguläre Operationen | $L(r_1) \cup L(r_2) = L(r_1 \mid r_2)$<br>$L(r_1)L(r_2) = L(r_1 r_2)$<br>$L(r_1)* = L(r_1*)$ |
| Verallgemeinerter NEA | $\mathrm{NEA}_\varepsilon$, dessen Übergänge mit regulären Ausdrücken beschriftet sind |
| Primitive reguläre Ausdrücke | Reguläre Ausdrücke für Wörter mit Länge $\leq 1$ |

| $L = L(r)$ | $r$ | NEA |
|---|---|---|
| $\varnothing$ | $\varnothing$ |   |
| $\{\varepsilon\}$ | $\{\varepsilon\}$ |   |
| $\{a\}$ | $a$ |   |
| $\{o,s,t\}$ | $[ost]$ |   |
| $\{a,b,...,s\}$ | $[a-s]$ |   |
| $\Sigma$ | . |   |

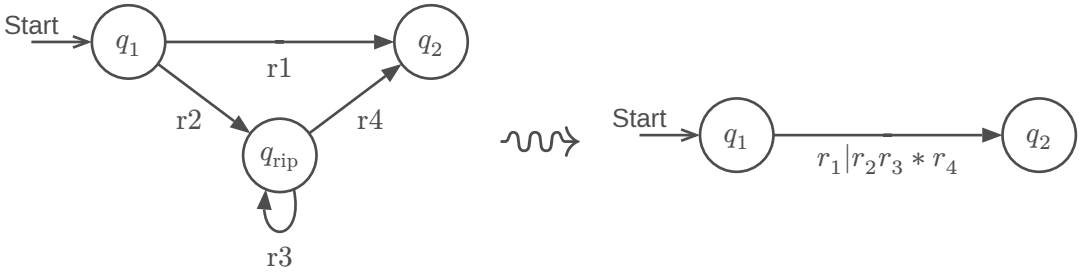
$\Rightarrow$ zu jedem regulären Ausdruck gibt es einen DEA

#### 5.10.1.1. VNEA $A$ umwandeln in Regex $r$

***Keine Übergänge nach $q_\emptyset$ und nur ein Akzeptierzustand***

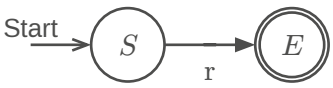



***Reduktion***




***Regulärer Ausdruck***

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



$\Rightarrow$ jede reguläre Sprache lässt sich mit einem regulären Ausdruck beschreiben $\{L(A) \mid A \text{ ein DEA}\} = \{L(r) | r \text{ ein regulärer Ausdruck}\}$

Interessantes projekt (DEA Lexer DSL): Ragel

### 5.10.2. Teststrategie

| *Messung* | *Folgerung* |
|---|---|
| Für $n = 1, 2, 3, ...$<br>– Regulären Ausdruck erzeugen<br>$r = \underbrace{a? \, a? \, a? \, a? \, ...a?}_{n \cdot a?} \underbrace{aaaa...a}_{n \cdot a}$<br>– Laufzeit für Akzeptieren von $a^n$ durch $r$ messen | – Laufzeit $O(2^n)$: NEA-Implementation<br>– Laufzeit $O(n)$: DEA-Implementation |

## 5.11. KONTEXTFREIE SPRACHEN

| *Begriff* | *Definition* |
|---|---|
| CFL | Context Free Language |
| CFG | Context Free Grammar |
| PDA | Pushdown Automata |

### 5.11.1. Kontextfreie Grammatik und Sprache

$$G = (V, \Sigma, R, S)$$

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

#### *Ableitung und erzeugte Sprache*

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

***Erzeugte Sprache:*** Die Menge der Wörter, die von einer kontextfreien Grammatik $G$ aus der Startvariablen abgeleitet werden können, wird mit $L(G)$ bezeichnet und heißt die von $G$ erzeugte Sprache.

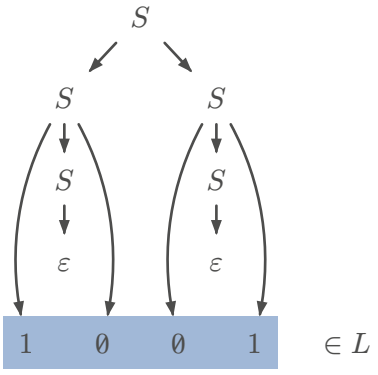
***Kontextfreie Sprache:*** Eine Sprache $L$ heißt kontextfrei, wenn es eine kontextfreie Grammatik gibt, die die Sprache $L = L(G)$ erzeugt.

#### *Parse Tree*

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

mit der Grammatik

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




***Beispiel** 8:* $L = \{0^n 1^n \mid n \geq 0\}$

Grammatik $L = \{0^n 1^n \mid n \geq 0\}$
1) Variablen: $V = \{S\}$
2) Terminalsymbole: $\Sigma = \{0, 1\}$
3) Regeln: $R = \{S \to \varepsilon | 0S1\}$
4) Startvariable: $S$

Wörter, die erzeugt werden müssen

$\varepsilon$
$01 = 0\varepsilon 1$
$0011 = 0011$

Es ist nicht garantiert, dass es für die Ableitung eines Wortes nur einen einzigen Syntaxbaum gibt. Man sagt, die Grammatik ist ***nicht eindeutig***, wenn sie mehrere Syntaxbäume für die gleichen Wörter erlaubt.

## 5.11.2. Reguläre Operationen

Die Klasse der kontextfreien Sprachen ist abgeschlossen unter regulären Operationen.

**Reguläre Sprachen sind kontextfrei**.

### 5.11.2.1. Grammatik für reguläre Operationen

Seien $L_1$ und $L_2$ kontextfreie Sprachen mit Grammatiken $G_i = (V_i, \Sigma, R_i, S_i)$. Wir nehmen an, dass die Variablen- und Regelmengen disjunkt sind, also $V_1 \cap V_2 = \varnothing \wedge R_1 \cap R_2 = \varnothing$. Zudem sei $S_\emptyset$ eine Variable, die nicht in $V_1 \cup V_2$ enthalten ist. Dann gilt:

***Alternative** $L_1 \mid L_2$ **ist kontextfrei mit der Grammatik***

$$G = (V_1 \cup V_2 \cup \{S_\emptyset\}, \Sigma, R = R_1 \cup R_2 \cup \{S_\emptyset \to S_1 \mid S_2\}, S_\emptyset)$$

***Verkettung** $L_1 L_2$ **ist kontextfrei mit der Grammatik***

$$G = (V_1 \cup V_2 \cup \{S_\emptyset\}, \Sigma, R = R_1 \cup R_2 \cup \{S_\emptyset \to S_1 S_2\}, S_\emptyset)$$

****-Operation** $L_1^*$ **ist kontextfrei mit der Grammatik***

$$G = (V_1 \cup \{S_\emptyset\}, \Sigma, R = R_1 \cup \{S_\emptyset \to S_\emptyset S_1, S_\emptyset \to \varepsilon\}, S_\emptyset)$$

## 5.11.3. Kontextfrei

### 5.11.3.1. Regeln ohne Kontext

Es spielt keine Rolle, in welchem Kontext das Zeichen $A$ vorkommt, die Regel kann immer angewendet werden

$$S \to aAb \to aab$$

$$S \to A \mid C$$
$$A \to a$$
$$A \to aAb$$
$$C \to bA$$

### 5.11.3.2. Regeln mit Kontext

Je nach Kontext kann eine Regel nicht unbedingt angewendet werden

$$S \to aC \to A$$
$$S \to bC \to B$$
$$S \to C \to ?$$

$$S \to C \mid aC \mid bC$$
$$aC \to A$$
$$bC \to B$$

## 5.11.4. Chomsky-Normalform (CNF)

Eine CFG ist in ***Chomsky-Normalform***, 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.

### 5.11.4.1. Umwandlung

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

### 5.11.4.2. Folgerungen

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
   - $\Rightarrow$ insgesamt $2|w| - 1$ Regelanwendungen
2) Deterministischer Parse-Algorithmus mit Laufzeit $O(|w|^3)$

***Beispiel** 9:*

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

1) Startvariable

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

2) $\varepsilon$-Regel

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

3) Unit-Regel

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

4) Komplexe Regeln

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

# 6. PARSING

## 6.1. COCKE-YOUNGER-KASAMI ALGORITHMUS (CYK)

***Gegeben***

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

***Frage***

Ist $w$ ableitbar? Formell geschrieben: $A \overset{*}{\Rightarrow} w$

– Spezialfall $w = \varepsilon$ : $A \overset{*}{\Rightarrow} \varepsilon \quad \Leftrightarrow \quad A \to \varepsilon \in R$ (Epsilonregel, $A = S$)
– Spezialfall$|w| = 1$ : $A \overset{*}{\Rightarrow} w \quad \Leftrightarrow \quad A \to w \in R$ (Terminalsymbolregel)
– Fall $|w| > 1$ :

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

### 6.1.1. Ableitungsdreieck

#### Ideen

– Parse Tree aus $A \to BC$ und $T \to t$
– Variablen in Tabelle füllen

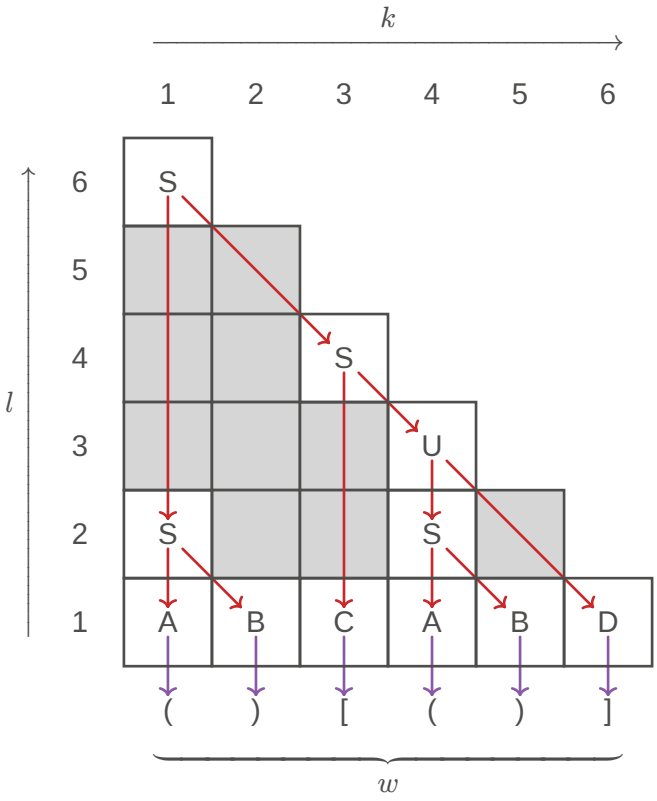
#### Prinzip

– Einem Teilwort entspricht ein Feld der Tabelle (rot hinterlegt)
– Das Feld wird mit den Variablen gefüllt, aus denen das Teilwort abgeleitet werden kann.

#### Beispiel

Parsen des Wortes `()[()]`

$S \to AB \mid CD \mid CU \mid SS$
$U \to SD$
$A \to ($
$B \to )$
$C \to [$
$D \to ]$




#### Definitionen

Die Felder $(k, l_1)$ und $(k + l_1, l - l_1)$ heissen ***korrespondierende Felder*** im Ableitungsdreieck des Feldes $(k, l)$.

## 6.2. BACKUS-NAUR-FORM (BNF)

Spezifikation der Regeln in maschinen-lesbarer Form:

– Variablen: `<variablen-name>`
– Einzelne Zeichen: `A`
– Zeichenketten: `'BEISPIEL'`
– Regeln: `<variablen-name> ::= Ausdruck`
– Ausdrücke sind Folgen von Variablen, einzelnen Zeichen oder Zeichenketten, getrennt durch |

**Beispiel 10:** **BNF für Expression-Term-Factor Grammatik**

#### Ausgangsgrammatik

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

#### Backus-Naur-Form

```
<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
```

## 6.3. EXTENDED BACKUS-NAUR-FORM (EBNF)

| *Definition* | *Achtung!* |
|---|---|
| – Literale in Anführungszeichen oder Apostroph<br>– = statt ::=<br>– Variablen ohne < und >, dürfen Leerzeichen enthalten<br>– Komma für Verkettung ,<br>– Regel-Endzeichen ;<br>– Kommentare (* ... *)<br>– Optionale Wiederholung { ... }<br>– Option [ ... ]<br>– Gruppierung ( ... )<br>– Ausnahme: - | – Keine Operatorpriorisierung<br>– Zweideutig!<br>– Keine eindeutige Evaluation! |

*Beispiel 11:* **EBNF für Expression-Term-Factor Grammatik**

```
expression  = expression, "+", term | term ;
term        = term, "*", factor, | factor ;
factor      = "(", expression, ")" | number ;
number      = digit, { digit } ;
digit       = "0" | "1" | "2" | "3" | "4" |
              "5" | "6" | "7" | "8" | "9" ;
```

## 6.4. RECHTSREKURSION

Expression-Term-Factor-Grammatik verwendet Links-Rekursion ⇒ korrekte Auswertung von links nach rechts. Parsetree wertet aber Ausdrücke von rechts nach links aus! Alternative Expression-Term-Factor definition mit Rechtsrekursion statt Linksrekursion:

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

# 7. STACK

Unendlich grosser Speicher

– Immer nur das oberste Element sichtbar
– Beliebig tief

## 7.1. STACKAUTOMAT / PUSHDOWN AUTOMATON (PDA)

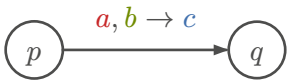
Kann Sprachen akzeptieren, die nicht regulär sind, ist aber nicht deterministisch. **Beachte**: $\Gamma \neq \Sigma$ möglich

***Definition***

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

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

***Übergänge***

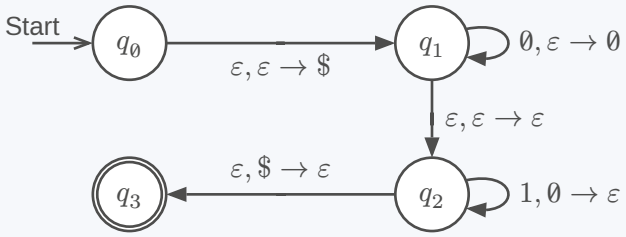



$a$ vom Input

$b$ vom Stack entfernen (Bedingung)

$c$ auf den Stack

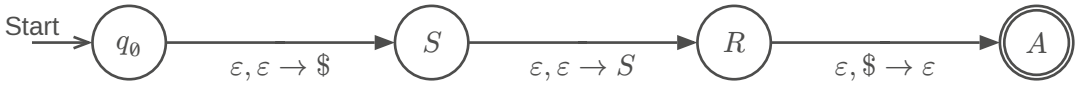
***Beispiel 12:*** **Stackautomat für die Sprache** $\{0^n 1^n \mid n \geq 0\}$

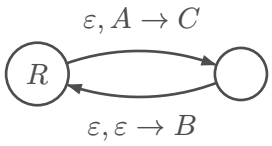
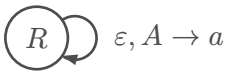
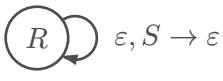
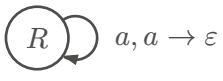



### 7.1.1. Ableitung aus CNF

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

***Grundgerüst***




| ***Regel*** $A \to BC$ | ***Regel*** $A \to a$ | ***Regel*** $S \to \varepsilon$ | $\forall a \in \Sigma$ |
|---|---|---|---|
|   |   |   |   |

### 7.1.2. PDA → CFG

***Variablen***

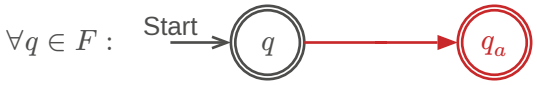
Wörter beschreiben Pfade durch den Automaten: Variable $A_{pq}$ = Wörter, die von $p$ nach $q$ führen mit leerem Stack

***Regeln***

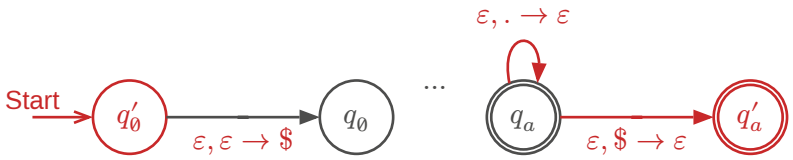
Regeln beschreiben, wie sich Wege zerlegen lassen: $A_{pq} \to A_{pr}A_{rq}$ = Wege von $p$ nach $q$ können verstanden werden als Wege von $p$ nach $r$ und von dort nach $q$

#### 7.1.2.1. Stackautomat standardisieren

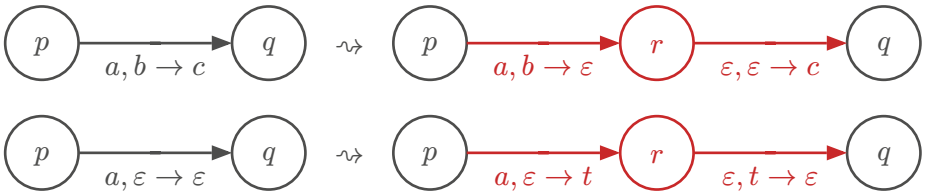
1) Nur ein Akzeptierzustand: Neuen Akzeptierzustand $q_a$ und Übergänge von allen vorherigen Endzuständen zu $q_a$ erstellen.




2) Stack leeren: Mit einem Element \$ erweitern, sodass garantiert werden kann, dass der Stack am schluss leer ist.



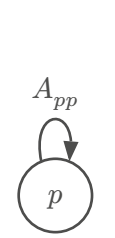

3) Jeder Übergang legt entweder ein Zeichen auf den Stack oder entfernt eines



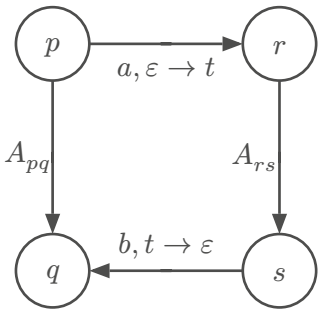

#### 7.1.2.1.1. Grammatik ablesen

Ausgangspunkt: standardisierter PDA mit Startzustand $q_0$ und $F = \{q_a\}$.

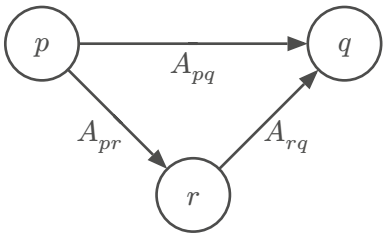
1) Startvariable: $A_{q_0,q_a}$
2) Regeln:




$A_{pp} \to \varepsilon$




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




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

# 8. NICHT KONTEXTFREIE SPRACHEN

## 8.1. PUMPING LEMMA FÜR CFL

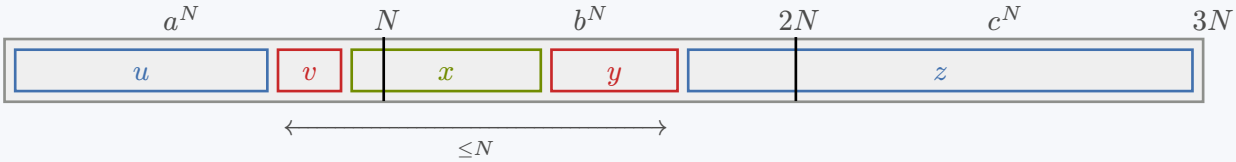
Ist $L$ eine CFL, dann gibt es eine Zahl $N$, die Pumping Length, derart, dass jedes Wort $w \in L$ mit $|w| \geq N$ zerlegt werden kann in fünf Teile $w = uvxyz$ derart, dass

1) $|vy| > \emptyset$
2) $|vxy| \leq N$
3) $\forall k \in \mathbb{N}. \left(uv^k xy^k z \in L\right)$

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

***Beispiel 13:*** $\{a^n b^n c^n \mid n \geq 0\}$

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




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

#### 8.1.0.1. Herleitung

***Grammatik $G$ in CNF***

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

***Wiederverwendete Variable***

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

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

***Pumpen***

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

# 9. UNENDLICH

| ***Begriff*** | ***Definition*** |
|---|---|
| Endlich | Eine Menge $A$ heisst ***endlich***, wenn jede Injektion $A \to A$ auch eine Bijektion ist |
| Abzählbar unendlich | Eine Menge $A$ heisst ***abzählbar unendlich***, wenn es eine Bijektion $\mathbb{N} \to A$ gibt, also $A \simeq \mathbb{N}$. Bsp: $\mathbb{R}, \mathbb{Z}, \mathbb{Q}, \Sigma^*$, Menge aller DEAs/NEAs/PDAs/CFGs |
| Überabzählbar unendlich | Eine Menge $A$ heisst ***überabzählbar unendlich***, wenn sie nicht abzählbar unendlich ist. Bsp: $\mathbb{C}, P(\mathbb{N})$, Menge aller Sprachen $P(\Sigma^*)$ |
| Gleich mächtig | Mengen $A$ und $B$ heissen ***gleich mächtig***, $A \simeq B$, wenn es eine Bijektion $A \to B$ gibt |
| Unendlich | Eine Menge $A$ heisst ***unendlich***, wenn sie gleich mächtig wie eine Teilmenge ist |

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

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

# 10. TURING-MASCHINE (TM)

Merhere Stacks (Speicherzellen) = RAM (Band).

Jeder DEA oder NEA kann damit emuliert werden.

1) Der Speicher ist unbegrenzt
2) In jeder Zelle 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 und das eine Zeichen)

| *Begriff* | *Definition* |
|---|---|
| Record | Speichereinheit fester grösse |
| Bandalphabet | Alphabet $\Gamma$, welches aus Speicherzellen (Stacks) besteht |
| Schreib-/Lesekopf | Aktuelle Schreib-/Leseposition |

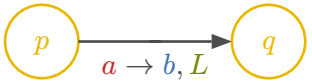
### *Definition*

(deterministische) Turing-Maschine
$M = (Q, \Sigma, \Gamma, \delta, q_\emptyset, q_{\text{accept}}, q_{\text{reject}})$

- $Q$: Zustände
- $\Sigma$: Alphabet
- $\Gamma$: Bandalphabet, $\textvisiblespace \in \Gamma \setminus \Sigma$
- $\delta$: $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- $q_\emptyset \in Q$: Startzustand
- $q_{\text{accept}} \in Q$: Akzeptierzustand
- $q_{\text{reject}} \in Q$: Ablehnzustand, $q_{\text{accept}} \neq q_{\text{reject}}$

### *Zustandsdiagramm*

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




### *Übergänge*

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

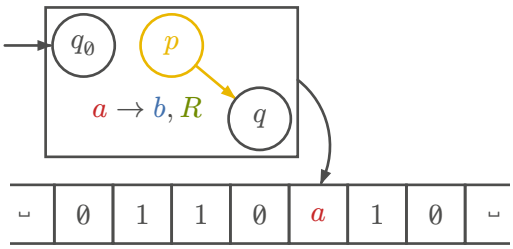
## 10.1. ARBEITSWEISE

### *Programmablauf*

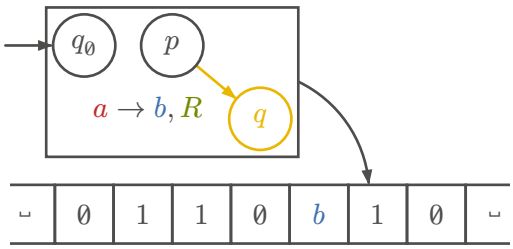
- Input-Wort $w$ auf Band
- Schreib- / Lesekopf positioniert auf 1. Zeichen
- Maschine starten, $t(w)$ Einzelschritte ausführen
- Maschine hält in $q_{\text{accept}}$ oder $q_{\text{reject}}$: Wort $w$ akzeptiert / Verworfen

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

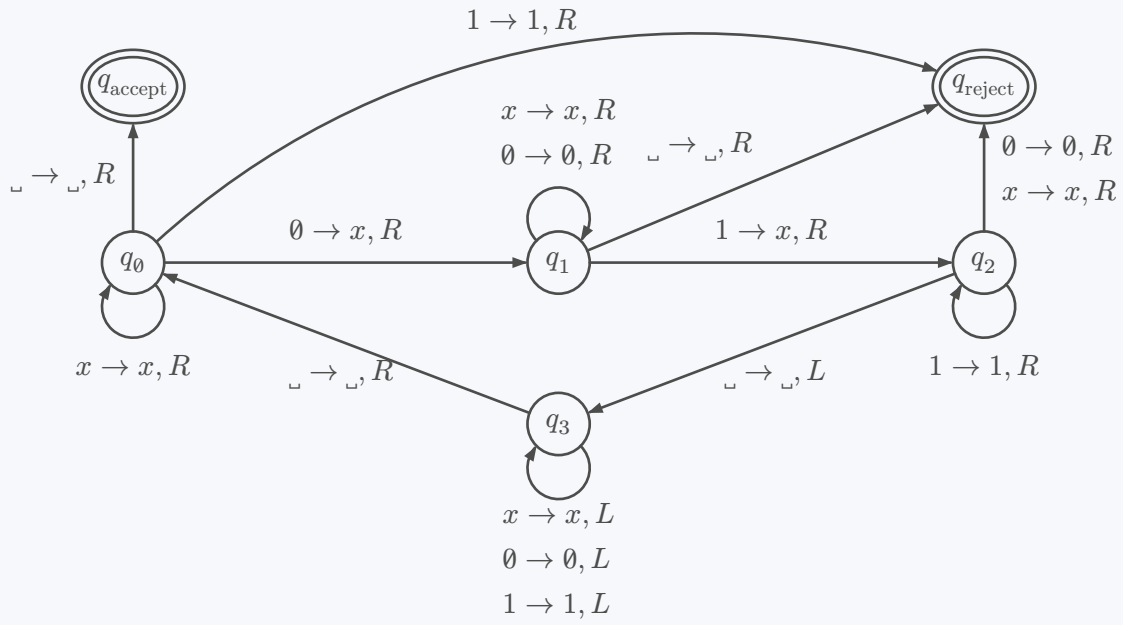
**Vorher**




**Nachher**



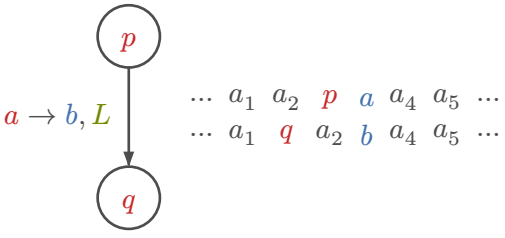

**Beispiel 14:** $\{0^n 1^n \mid n \geq 0\}$



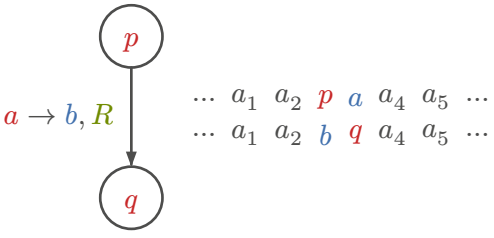

## 10.2. PROTOKOLLIERUNG

Der Gang der Berechnung mit einer TM kann dokumentiert werden, indem der Bandinhalt nach jedem einzelnen Verarbeitungsschritt protokolliert wird.

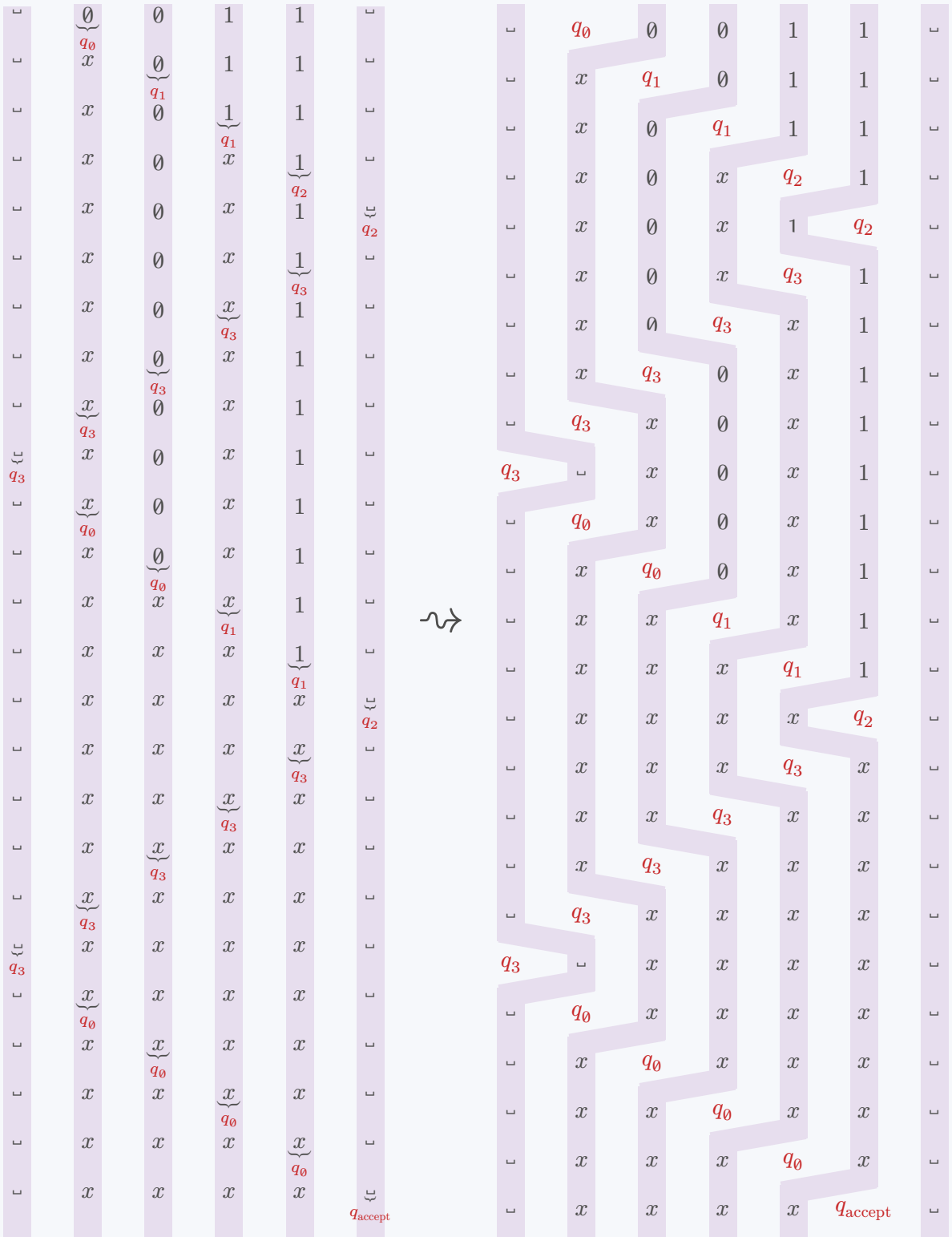
**Links**




**Rechts**



**Beispiel 15:** $L = \{0^n 1^n \mid n \geq 0\}, w = 0011$




## 10.3. VARIANTEN

***Laufzeit:*** Die Laufzeit einer TM $M$ bei der Verarbeitung eines Wortes $w \in \Sigma^*$ ist die Anzahl $t(w)$ der Schritte, die die Turing-Maschine ausführt, bis sie anhält.

### 10.3.1. Nichtdeterministische TM

#### *Übergangsfunktion*

Bei jedem Übergang maximal $N$ verschiedene Möglichkeiten: $\delta : Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$
$\Rightarrow$ Mehrere mögliche Berechnungswege

#### *Wort akzeptieren*

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

#### *Simulationsidee*

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

Simulierbar in $O\left(N^{t(n)}\right) = 2^{O(t(n))}$

#### *Simulation auf Standard-TM*

Verwende 3 Bänder:

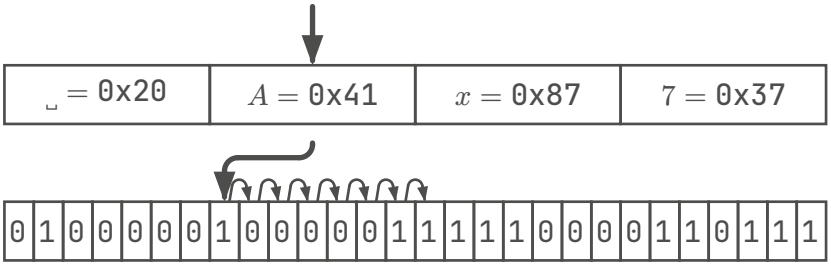
1) Arbeitsband
2) Kopie von w
3) Liste aller Folgen von Wahlmöglichkeiten

#### *Simulation*

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

### 10.3.2. Verschiedene Bandalphabete

Jedes andere Alphabet als $\Sigma = \{0, 1\}$ kann binär codiert werden.




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

### 10.3.3. Mehrspurmaschine

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

### 10.3.4. Mehrbandmaschine

Um eine Mehrbandmaschine mit $n$ Bändern auf einer einfacheren Maschine zu simulieren, kann die unabhängige Bewegung der Schreib-/Leseköpfe mithilfe eines $2n$-spurigen Bandes (Daten + Zeiger für Schreib-/Lesekopf) nachgebildet werden.

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

#### 10.3.4.1. Harvard- und von Neumann-Architektur

Die AVR-Mikrokontroller-Familie von Atmel verwendet intern drei unabhängige Datenpfade:
– Flash-Speicher, der den Programmcode enthält
– Statisches RAM, welches zur Laufzeit veränderliche Daten speichert
– Peripheriegeräte.

Die von Neumann-Architektur vereinheitlicht Daten- und Programmspeicher.

### 10.3.5. Einseitig unendliches Band

Beidseitig unendliches Band kann durch ein Alphabet doppelter Wortbreite realisiert werden, somit äquivalent.

## 10.4. TURING-ERKENNBARE SPRACHEN

Eine TM erkennt das Wort $w \in \Sigma^*$, wenn die Maschine auf dem Input $w$ im Zustand $q_{\text{accept}}$ anhält.

Sind $L_1$ und $L_2$ Turing-erkennbare Sprachen, dann sind auch der Durchschnitt $L_1 \cap L_2$ und die Vereinigungsmenge $L_1 \cup L_2$ Turing-erkennbar.

Es gibt überabzählbare viele Sprachen $L \subset \Sigma^*$, die **nicht** Turing-erkennbar sind.

### 10.4.1. Aufzähler

***Aufzähler:*** TM mit einem Drucker, mit dem Wörter ausgedruckt werden können.

$L$ ist ***rekursiv aufzählbar*** wenn es einen Aufzähler gibt, der alle Wörter aufzählen kann. $L$ ist dann auch Turing-erkennbar.

### 10.4.2. Entscheider und entscheidbare Sprachen

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

***Turing-entscheidbare Sprache:*** $L$ heisst ***Turing-entscheidbar***, wenn es einen Entscheider $M$ gibt mit $L = L(M)$.

***Berechenbare Funktion:*** Eine Funktion $f : \Sigma^* \to \Sigma^*$ heisst ***berechenbar***, wenn es eine TM $M$ gibt, die auf jedem Inputwort $w \in \Sigma^*$ anhält und auf dem Band das Outputwort $f(w)$ zurücklässt.

# 11. ENTSCHEIDBARKEIT

> *«Die sich mit immer neuen Superlativen gegenseitig überbietenden Ankündigungen der IT-Industrie könnten den Eindruck erwecken, dass mit geeignet viel Rechenleistung und Speicherplatz kein Informatikproblem einer Lösung widerstehen könnte.»*
>
> *— Prof. Dr. Andreas Müller, [1, p. 267]*

```
> based
```

Serialisierung ermöglicht, jedes Entscheidungsproblem auf die Verarbeitung einer Zeichenkette zurückzuführen, die entweder akzeptiert oder verworfen wird. Jedes Problem wird auf diese Weise zu einem Sprachproblem.

## 11.1. ENTSCHEIDER

### Definition

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

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

### Sprachproblem

Jedes Problem $P$ kann in ein Sprachproblem übersetzt werden:

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

### Beispiele

Leerheitsproblem (Entscheidbar):

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

Gleichheitsproblem (Entscheidbar):

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

Akzeptanzproblem (Entscheidbar – nicht für TM!):

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

Halteproblem (Nicht entscheidbar):

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

## 11.2. AKZEPTANZPROBLEM FÜR TURING-MASCHINEN

$$A_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ ist eine TM, die das Wort } w \in \Sigma^* \text{ akzeptiert}\}$$

***Theorem***

$A_{\text{TM}}$ ist nicht entscheidbar.

***Beweis***

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

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

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

$$D(\langle D \rangle) \text{ akzeptiert} \Rightarrow D \text{ verwirft } \langle D \rangle$$
$$D(\langle D \rangle) \text{ verwirft} \Rightarrow D \text{ akzeptiert } \langle D \rangle$$

## 11.3. HALTEPROBLEM

$$\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid \text{die TM } M \text{ hält auf dem Input } w\}$$

$\Rightarrow$ Gleicher Widerspruch wie beim Akzeptanzproblem. Umschreiben des Problems in ein neues Programm $S$, welches «Akzeptieren» in «Anhalten» übersetzen soll = Reduktion ($A_{\text{TM}} \leq \text{HALT}_{\text{TM}}$)

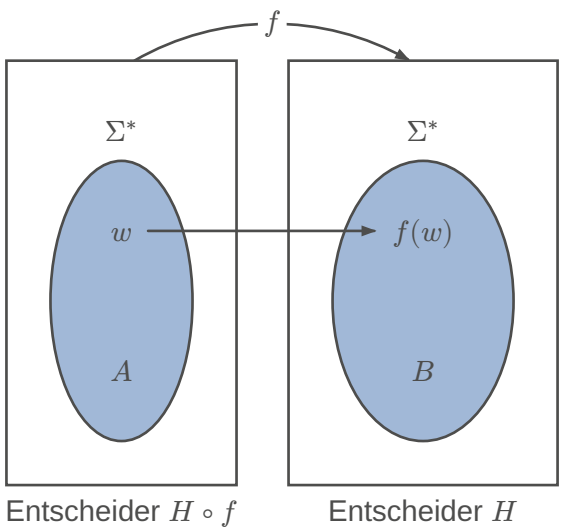
## 11.4. REDUKTION

***Reduktionsabbildung***

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

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

Notation: $f : A \leq B$ heisst «$A$ leichter entscheidbar als $B$»




***Entscheidbarkeit***

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

***Beweis***

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

***Folgerung***

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

## 11.5. SATZ VON RICE

Ist $P$ eine nichttriviale Eigenschaft Turing-erkennbarer Sprachen, dann ist $P_{TM}$ nicht entscheidbar.

# 12. KOMPLEXITÄT

***Ineffizient:*** Entscheidbare Probleme mit exorbitant langer Laufzeit sind praktisch «nicht lösbar»

***Problemgrösse:*** Laufzeit hängt von der Problemgrösse $n$ ab

***Worst case Laufzeit:*** Die worst case Laufzeit $t(n)$, die die Maschine zur Verarbeitung von Inputs der Länge $n$ braucht, ist $t(n) = \max\{t(w) \mid |w| \leq n\}$

## 12.1. HARDWAREEINFLUSS

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

## 12.2. POLYNOMIELLE UND EXPONENTIELLE LAUFZEIT

| ***Polynomielle Probleme*** | ***Exponentielle Probleme*** |
|---|---|
| – Gauss-Algorithmus $O(n^3)$<br>– … | – Faktorisierung von grossen Zahlen (RSA)<br>– … |
| Skalierbarkeit | Sicherheit |

Ein Algorithmus hat ***polynomielle Laufzeit***, wenn seine Laufzeit durch ein Polynom beschränkt ist: $t(n) = O(n^k)$.

## 12.3. SPRACHEN

Eine Sprache $L$ gehört zur Klasse NP, wenn $L$ von einer *nicht*deterministischen TM in polynomieller Zeit **entschieden** werden kann

## 12.4. POLYNOMIELLE VERIFIZIERER

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

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

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

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

## 12.5. POLYNOMIELLE REDUKTION

### 12.5.1. Polynomieller Laufzeit-Vergleich

Seien $A$ und $B$ entscheidbare Sprachen. Eine berechenbare Abbildung

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

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

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

***Polynomiell äquivalent:*** Zwei Sprachen $A$ und $B$ heißen ***polynomiell äquivalent***, geschrieben $A \equiv_P B$, wenn $A \leq_P B$ und $B \leq_P A$.

### 12.5.2. Satz

Sei $A \leq_P B$. Dann gilt

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

## 12.6. POLYNOMIELLE AUSFÜLLRÄTSEL

Ein polynomielles Ausfüllrätsel ist eine $n \times m$-Tabelle, in die Zeichen eines Alphabets $\Sigma$ eingefüllt werden müssen, so dass als logische Formel ausdrückbare Regeln eingehalten werden, die in polynomieller Zeit ausgewertet werden können. (Beispiel: Sudoku)

***SAT:*** Boolean satisfiability problem: Gegeben einer aussagenlogischen Formel, gibt es eine Zusammensetzung der Variablenwerte, die die Aussage «Wahr» werden lassen?

$$SAT = \{\varphi \mid \varphi \text{ ist eine erfüllbare logische Formel}\}$$

### Satz

Jedes polynomielle Ausfüllrätsel $A$ lässt sich polynomiell auf $SAT$ reduzieren.

### Variablen

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

### Genau ein Zeichen im Feld $(i,j)$

$$\varphi_{ij} = \bigvee_{c \in \Sigma} \underbrace{\left( x_{ijc} \wedge \bigvee_{d \neq c} x_{ijd} \right)}_{c \text{ und kein anderes Zeichen in } (i,j)}$$

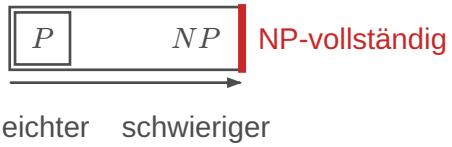
### Regeln

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

## 12.7. NP-VOLLSTÄNDIGKEIT

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

$$\forall A \in NP.\ (A \leq_P B)$$




NP-vollständige Probleme sind die «schwierigsten» Probleme in NP. Sie sind alle gleich schwierig:

$$A, B \text{ NP-vollständig} \Rightarrow (A \leq_P B \wedge B \leq_P A \ \Leftrightarrow \ A \equiv_P B)$$

### Reduktionsprinzip

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

Wenn ein Problem NP-vollständig ist:
– Lösung braucht typischerweise exponentielle Zeit
– Korrektheit der Lösung ist in polynomieller Zeit zu prüfen

Umgang mit NP-Vollständigkeit in der Praxis:
– Spezialfälle ausnützen
– Problem modifizieren
– Approximation
– Zufall oder Heuristik
– Genetische Algorithmen und ML

Beispiele:

– SAT / 3SAT
– k-VERTEX-COLORING

### 12.7.1. Cook-Levin

***3-SAT:*** Jede Klausel der Normalform hat maximal 3 Literale.

***Satz***

SAT ist NP-vollständig.

Das bedeutet:

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

***Vorgehensweise***

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

«Ausfüllrätsel» in diesem Fall die Berechnungsgeschichte der TM.

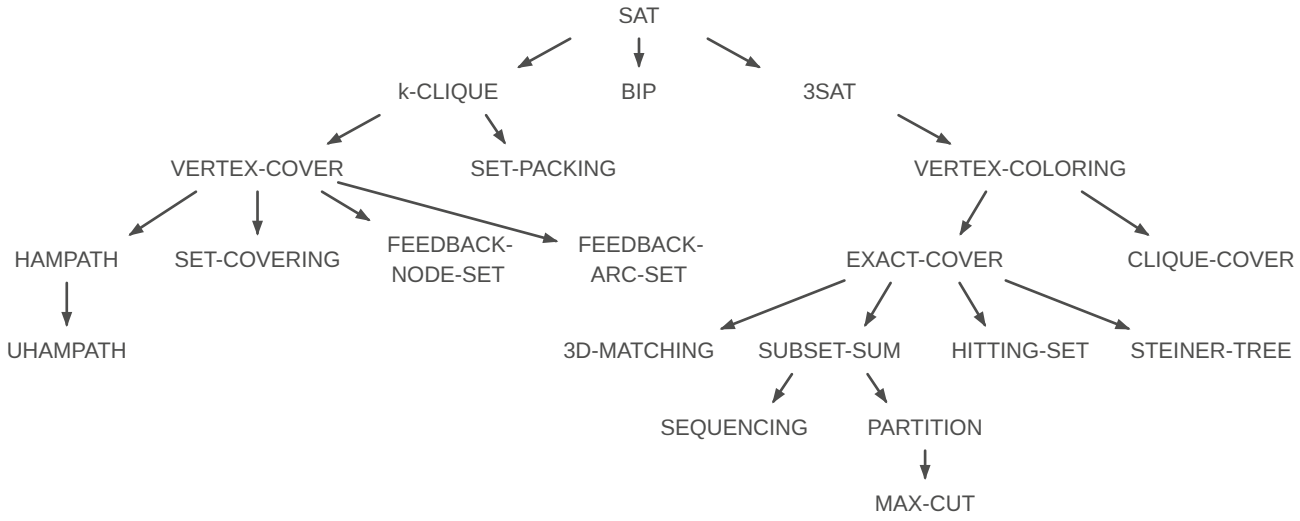
Überprüfe, ob alle Zustandsübergänge valid sind.

### 12.7.2. $SAT \leq_P 3SAT$

$$\{\varphi \mid \varphi \text{ erfüllbare logische Formel}\} \Rightarrow \{\varphi \mid \varphi \text{ erfüllbare logische Formel in 3CNF}\}$$

***Beweis von Cook-Levin***
– Jedes Problem lässt sich auf eine Formel reduzieren
– Die Reduktion ist polynomiell
– Die Formel wird aus der Berechnungsgeschichte abgeleitet
– Die Formel kann in CNF konstruiert werden

### 12.7.3. Karp-Katalog



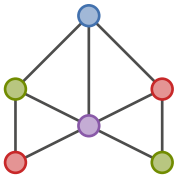

#### 12.7.3.1. Graph-Überdeckungen

Handeln davon, dass ein gegebener Graph von einer Menge beschränkter Größe mit einer besonderen Eigenschaft aus erreicht werden kann.

#### 12.7.3.1.1. VERTEX-COLORING

Gegeben ist ein Graph $G = (V, E)$ mit Knoten $V$ und Kanten $K$ und eine Zahl $k$. Ist es möglich, die Knoten des Graphen mit $k$ Farben so einzufärben, dass durch Kanten verbundene Knoten verschiedene Farben haben?

$k = 4$



$$\text{VERTEX-COLORING} = \left\{ \langle G, k \rangle \middle| \begin{array}{l} \text{Es gibt eine Einfärbung der Knoten von } G \\ \text{mit } k \text{ Farben derart, dass zwei verbundene} \\ \text{Knoten verschiedene Farben haben} \end{array} \right\}$$

*Beispiel 16:* **Stundenplan: Planung der Fächer $f \in F$ auf $k$ Zeitfenster, damit keine Anmeldungen $\{f_1, f_2\} \in A$ in gleichen Zeitfenstern liegen**

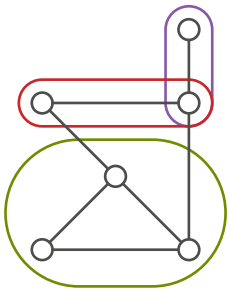
**Knoten $V$ =** Fach, **Kanten $E$ =** Anmeldungen, **Farben =** Zeitfenster

#### 12.7.3.1.2. CLIQUE-COVER

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

$k = 3$

$$\text{CLIQUE-COVER} = \left\{ \langle G, k \rangle \middle| \begin{array}{l} \text{Es gibt } k \text{ Cliquen } G_k = (V_k, E_k), \text{die zusammen} \\ \text{alle Knoten von } G \text{ abdecken, also } V = \bigcup_{i=1}^{k} V_i \end{array} \right\}$$

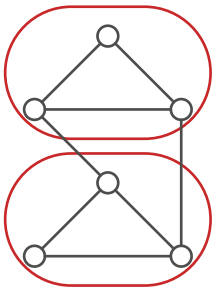


#### 12.7.3.1.3. EXACT-CLIQUE-COVER

Wie CLIQUE-COVER, nur müssen die Mengen disjunkt sein.

$k = 2$

$$\text{EXACT-CLIQUE-COVER} = \left\{ \langle G, k \rangle \middle| \begin{array}{l} \text{Es gibt } k \text{ disjunkte Cliquen } G_k = (V_k, E_k), \\ V_i \cap V_j = \varnothing \text{ für } i \neq j, \text{die zusammen alle} \\ \text{Knoten von } G \text{ abdecken, also } V = \bigcup_{i=1}^{k} V_i \end{array} \right\}$$



*Beispiel 17:* **Bilden von $k$ Gruppen von Leuten, die sich kennen. Alle Leute sollen eingeteilt sein.**

**Knoten $V$ =** Teilnehmer, **Kanten $E$ =** Kennen sich, **$k$ =** Anzahl Gruppen, **Clique $G_k$ =** Gruppe

#### 12.7.3.1.4. K-CLIQUE

Wie CLIQUE-COVER, nur ist $k$ bekannt.

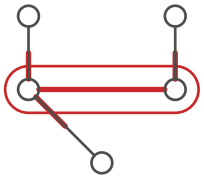
*Beispiel 18:* **Job-Parallelisierbarkeit: Menge von $n$ Jobs, die jeweils exklusiv auf $m$ Ressourcen zugreifen, Jobs dürfen nicht gleichzeitig laufen. Ziel: Entscheiden, ob zu irgendeinem Zeitpunkt mehr als $k$ Prozessoren ausgelastet werden können**

**Graph $G$ =** Job-Parallelisierbarkeit, **Knoten $V$ =** $n$ Job, **Kanten $E$ =** $m$ gleiche Ressourcen, **$k$-Clique $G_k$ =** Auswahl Knoten ohne Verbindung, **$k$ =** Auslastbare Anzahl Prozessoren

#### 12.7.3.1.5. VERTEX-COVER

Gibt es eine Menge $W$ von $k$ Knoten derart, dass jede Kante von $G$ einen Endpunkt in $W$ hat?

$k = 2$



$$\text{VERTEX-COVER} = \left\{ \langle G = (V, E), k \rangle \middle| \begin{array}{l} \text{Es gibt eine } k\text{-elementige Teilmenge} \\ W \subset V \text{ mit } W \cap e \neq \varnothing \forall e \in E \end{array} \right\}$$

***Beispiel** 19:* **Kontrolle des Verkehrsnetzes: Mitarbeiter haben ihre Basis an einzelnen Knotenpunkten. Ziel: An welchen Stationen muss man Kontrolleure stationieren, damit jede Strecke bei einem Kontrolleur endet?**

**Knoten $V$ =** Stationen, **Kanten $E$ =** Strecke, **$k$ =** Anzahl Kontrolleure, **$W \subset V$ =** Knoten mit Kontrolleur

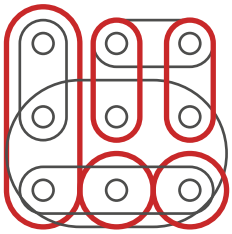
#### 12.7.3.2. Mengen

Handeln von einer Familie $(S_i)_{i \in I}$ von nichtleeren endlichen Mengen. In diesen Problemen geht es um die existenz einer Teilfamilie mit bestimmten Eigenschaften.

#### 12.7.3.2.1. SET-PACKING

Wie viele der Mengen $S_i$ kann man in $U$ hineinpacken, ohne dass sie sich überlappen?
Gibt es $k$ Teilmengen in $S$, welche disjunkt sind und die Menge $U$ abdecken?

$k = 5$



$$\text{SET-PACKING} = \left\{ \left\langle (S_i)_{i \in I}, k \right\rangle \middle| \begin{array}{l} \text{Es gibt eine } k \text{ -elementige Teilfamilie} \\ J \subset I, |J| = k \text{ von paarweise disjunkten} \\ \text{Mengen } S_k \cap S_l = \varnothing \forall k, l \in J \end{array} \right\}$$

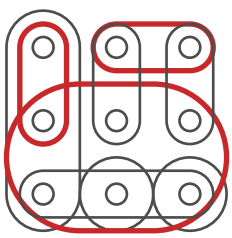
***Beispiel** 20:* **Medizinische Studie: 17 Allergene wählen, sodass jeder Proband maximal auf eines davon reagiert.**

$I =$ Allergene, $S_i =$ auf Allegien $i$ allergische Probanden, $J =$ Ausgewählte Allergene, $S_i \cap S_j = \varnothing \rightarrow$ Ausschlussbedingung zwischen Allergenen

#### 12.7.3.2.2. SET-COVERING

Gibt es eine Unterfamilie bestehend aus $k$ Mengen, die die gleiche Vereinigung $U$ hat?
Kann man $k$ Teilmengen bilden, welche die Menge $S$ komplett abdecken?

$k = 3$



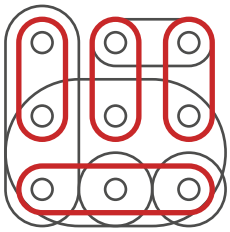
$$\text{SET-COVERING} = \left\{ \left\langle (S_i)_{i \in I}, k \right\rangle \middle| \begin{array}{l} \text{Es gibt eine } k \text{ -elementige Teilfamilie} \\ J \subset I, |J| = k \text{ mit} \bigcup_{j \in J} S_j = U \end{array} \right\}$$

**Beispiel 21:** **Ein Tourist möchte mit dem Besuch von nur $k$ Aussichtspunkten alles sehen, was man an Sehenswürdigkeiten von allen Aussichtspunkten aus sehen könnte.**

$U$ = Alle Sehenswürdigkeiten, $i \in I$ = Aussichtspunkt $i$, $S_i$ = Sichtbare Sehenswürdigkeiten von $i$ aus, $J \subset I$ = Ausgewählte Aussichtspunkte, $\bigcup_{j \in J} S_j = U$ = Bedingung, alles sehen zu können

#### 12.7.3.2.3. EXACT-COVER

Gibt es eine Unterfamilie von Mengen, die disjunkt sind, aber die gleiche Vereinigung haben? Jedes Element in $U$ soll genau in einer der Teilmengen der Familie $S$ vorkommen. Die gesuchte Menge bildet eine exakte Überdeckung.



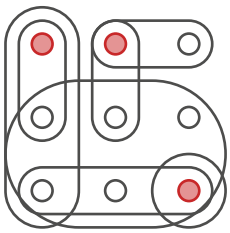
$$\text{EXACT-COVER} = \left\{ \left\langle (S_i)_{i \in I} \right\rangle \middle| \begin{array}{l} \text{Es gibt eine Teilfamilie } J \subset I \text{ paarweise} \\ \text{disjunkter Mengen mit der gleicher Vereinigung} \end{array} \right\}$$

**Beispiel 22:** **Es stehen binäre Eigenschaften von Kunden zur Verfügung, z.B. ob sie ein bestimmtes Produkt gekauft haben oder volljährig sind. Ziel: Eine Teilmenge von Kriterien finden, dass jeder Kunde genau eine der Eigenschaften hat.**

$I$ = Eigenschaften, $(S_i)_{i \in I}$ = Kunden, auf die die Eigenschaft zutrifft , $J \subset I$ = Teilmenge

#### 12.7.3.2.4. HITTING-SET

Sucht zu $U$ eine Menge von Punkten (Elemente), die jede Menge der Familie in genau einem Punkt treffen. Gibt es eine Menge $W \subset U$ derart, dass $W$ mit jeder Menge $S_i$ nur ein Element gemeinsam hat?



$$\text{HITTING-SET} = \left\{ \left\langle (S_i)_{i \in I} \right\rangle \middle| \begin{array}{l} \text{Es gibt eine Teilfamilie } W \subset U = \bigcup_{i \in I} S_i \\ \text{derart, dass } |W \cap S_i| = 1 \forall i \in I \end{array} \right\}$$

**Beispiel 23:** **Finden eines Einzelnen Vertreters für Gruppen von Menschen.**

$i \in I$ = Gruppierungskriterien, $S_i$ = Menschen in Gruppe $i$, $W \subset \bigcup_{i \in l} S_i$ = Teilmenge von Vertretern, $|W \cap S_i| = 1 \forall i \rightarrow H$ hat mit jeder der Mengen $S_i$ genau einen Vertreter gemeinsam

### 12.7.3.3. Aufteilung in zwei Mengen

#### 12.7.3.3.1. PARTITION

Gibt es eine Unterteilung einer Liste $S$ von ganzen Zahlen in zwei disjunkte Listen mit gleicher Summe?

$[\underbrace{1,2,4,5}_{\substack{T \\ \rightarrow \\ 12}}, \underbrace{6,8}_{\substack{\overline{T} \\ \rightarrow \\ 12}}]$

$$\text{PARTITION} = \left\{ S = [s_1, ..., s_n] \middle| \begin{array}{l} \text{Es gibt eine Teilliste } T \subset S \text{ , deren} \\ \text{Elemente die gleiche Summe haben} \\ \text{wie die Elemente der Liste } \overline{T} = S \setminus T \end{array} \right\}$$

**Beispiel 24:** **Unterteilung der Prominenten + Entourage auf zwei Säle des Film-Festivals.**

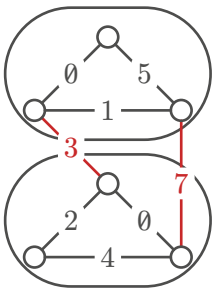
$i \in I$ = Promi, $c_i$ = Anz. Mitglieder seiner Entourage, $A$ = Stars samt Entourage für Saal A, $B$ = für Saal B, $I = A \cup B, \sum_{i \in A} c_i = \sum_{i \in B} c_i$

#### 12.7.3.3.2. MAX-CUT

Gegeben ein Graph $G = (V, E)$ mit einer Gewichtsfunktion $\varphi : E \to \mathbb{Z}$ der Kanten und einer ganzen Zahl $k \in \mathbb{Z}$, gibt es eine Aufteilung der Knotenmenge $V = V_1 \cup V_2$ in zwei disjunkte Teilmengen, so dass das Gewicht der Kanten, die $V_1$ und $V_2$ verbinden, mindestens $w$ ist:

$$\varphi(V_1, V_2) = \sum_{v_1 \in V_1, v_2 \in V_2, e=\{v_1,v_2\} \in E} \varphi(e) \geq k?$$

$k = 10$




**Beispiel** *25:* **Feindliche Übernahme einer Firma, mit resultierender Aufteilung der Abteilung, sodass diese möglichst ineffizient miteinander kommunizieren können.**

$V$ = Abteilung, $E$ = Kommunikationsbeziehung, $\varphi$ = Kommunikationsvolumen

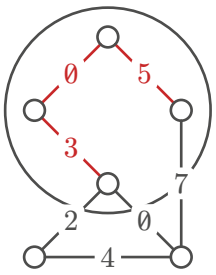
#### 12.7.3.4. Kombinatirische Optimierung

Handelt um das finden eines Maximums oder Minimums einer Zielfunktion.

#### 12.7.3.4.1. STEINER-TREE

Gibt es zu einem Graphen $G = (V, E)$ mit einer Gewichtsfunktion $\varphi : E \to \mathbb{Z}$, einer Teilmenge $R \subset V$ der Knoten und einer Zahl $k$ einen Teilbaum $T = (V_1, E_1) \subset G$ des Graphen, der alle Knoten von $R$ enthält und dessen Kantengewicht $\sum_{e \in E_1} \varphi(e) \leq k$ ist?

$k = 8$




**Beispiel** *26:* **Stromnetzausbau zwischen Ortschaften gegeben eines Budgets.**

**STEINER-TREE** = Stromnetz, **Knoten** $V$ = Ortschaften, **Knoten** $R \subset V$ = zu erschliessende Ortschaften, **Gewicht** $\varphi$ = Baukosten einer Verbindung, **Max. Gewicht** $k$ = Budget

#### 12.7.3.4.2. SEQUENCING

Die Komponenten der drei Vektoren

$$\text{Laufzeiten: } T = (T_1, ..., T_p) \in \mathbb{Z}^p$$
$$\text{Deadlines: } D = (D_1, ..., D_p) \in \mathbb{Z}^p$$
$$\text{Strafen: } P = (P_1, ..., P_p) \in \mathbb{Z}^p$$

beschreiben je einen Job. Job $i$ hat Laufzeit $T_i$, sollte zur Zeit $D_i$ abgeschlossen sein und kostet die Strafe $P_i$, wenn er nicht rechtzeitig fertig ist. Die Jobs werden sequenziell ohne unterbruch abgearbeitet. Ist es möglich, die Reihenfolge der Ausführung der Jobs so zu planen, dass die entstehenden Strafen $\leq k$ sind?

***Beispiel** 27:* **Eine Firma hat eine bestimmte Anzahl laufende Verträge. Es ist nicht möglich, alle Verträge in einer bestimmten Zeit abzuarbeiten. Sie versucht also möglichst viele Verträge in der verbleibenden Zeit abzuarbeiten, die eine hohe Entlöhnung haben.**

$T$ = Zeitaufwände für Vertragserfüllung, $P$ = Entlöhnung der Verträge, $D$ = Deadlines der Verträge

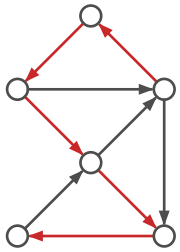
#### 12.7.3.5. Pfade in Graphen

Ein hamiltonscher Pfad in einem Graphen $G = (V, E)$ ist ein Pfad, der jeden Knoten des Graphen genau einmal besucht. Ein solcher Pfad definiert eine Reihenfolge $a_1, ..., a_n$ von Knoten, so dass jede Kante von $a_i$ nach $a_{i+1}$ in $E$ ist. Für einen gerichteten Graphen sind dies die Kanten $(a_i, a_{i+1}) \in E$, für einen ungerichteten Graphen gilt $\{a_i, a_{i+1}\} \in E$. Ein hamiltonscher Zyklus ist ein geschlossener hamiltonscher Pfad.

##### 12.7.3.5.1. HAMPATH

Gibt es einen gerichteten hamiltonschen Pfad im gerichteten Graphen $G$?

$$\text{HAMPATH} = \left\{ \langle G \rangle \middle| \begin{matrix} \text{Im gerichteten Graphen } G \text{ gibt es} \\ \text{einen gerichteten hamiltonischen Pfad} \end{matrix} \right\}$$



***Beispiel** 28:* **Ganze Schweiz bereisen, ohne eine Stadt mehrmals zu besuchen**

$G$ = Liniennetz, $V$ = Stadt, $E$ = ÖV-Verbindung, **Hamiltonscher Pfad =** Reise

##### 12.7.3.5.2. HAMCIRCUIT

Wie HAMPATH, aber zusätzlich muss der Pfad geschlossen sein

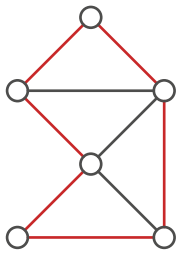
##### 12.7.3.5.3. UHAMPATH

Wie HAMPATH, aber ungerichtet

$$\text{UHAMPATH} = \left\{ \langle G \rangle \middle| \begin{matrix} \text{Im Graphen } G \text{ gibt es einen} \\ \text{hamiltonischen Pfad} \end{matrix} \right\}$$

##### 12.7.3.5.4. UHAMCIRCUIT

Wie UHAMPATH, aber zusätzlich muss der Pfad geschlossen sein



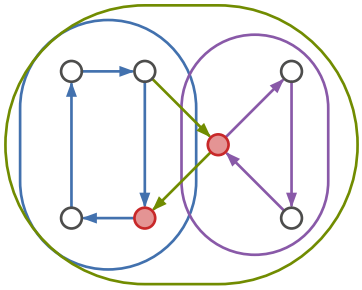
#### 12.7.3.6. Gerichtete Graphen

Es handelt sich um die Eigenschaften von Zyklen in gerichteten Graphen $G = (V, E)$. Die Kantenmenge $E$ besteht aus gerichteten Kanten $(a, e)$ mit $a, e \in V$. Ein gerichteter Zyklus in $G$ ist eine Folge von Knoten $a_\emptyset, ..., a_n = a_\emptyset$, die jeweils Enden von Kanten sind, also $(a_i, a_{i+1}) \in E, i = \emptyset, ..., n-1$

#### 12.7.3.6.1. FEEDBACK-NODE-SET

Gibt es zu einem gerichteten Graphen $G$ und einer Zahl $k$ eine Menge $W \subset V$ von $k$ Knoten derart, dass jeder geschlossene Zyklus durch einen dieser Knoten geht?

$k = 2$



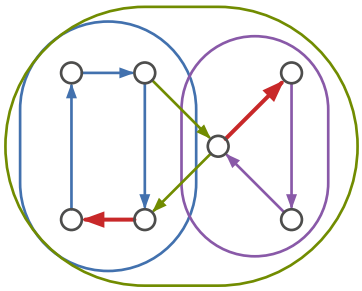
**Beispiel** *29:* **Es gibt mehrere Buslinien. Wo muss das Putzpersonal platziert werden, damit alle Linien geputzt werden können? Man möchte möglichst wenig Personal einsetzen.**

$\boldsymbol{V}$ = Stationen, $\boldsymbol{W \subset V}$ = Stationen, von denen aus alle Linien erreicht werden, $\boldsymbol{k}$ = Anzahl Personal, $\boldsymbol{G}$ = Buslinienplan, $\boldsymbol{E}$ = Gerichtete Buslinien

#### 12.7.3.6.2. FEEDBACK-ARC-SET

Gibt es zu einem gerichteten Graphen $G$ und einer Zahl $k$ eine Menge $F \subset E$ von $k$ Kanten derart, dass jeder geschlossene Zyklus eine dieser Kanten enthält?

$k = 2$



### 12.7.3.7. Logik

#### 12.7.3.7.1. SAT

Gegeben einer aussagenlogischen Formel, gibt es eine Zusammensetzung der Variablenwerte, die die Aussage «Wahr» werden lassen?

$$\mathrm{SAT} = \{\varphi \mid \varphi \text{ ist eine erfüllbare logische Formel}\}$$

**Beispiel** *30:* **Elektriker: Konfiguration der Schalter zweier Stromkreise, die in allen Räumen Licht brennen lassen sollten**

$\boldsymbol{x_i}$ = Schalter, $\boldsymbol{T \vee F}$ = Stromkreise (werte der Variable $x_i$, Lampen des ersten Kreises leuchten, wenn $x_i = T$), $\boldsymbol{x_{i_1} \vee \ldots \vee x_{i_k}}$ = ob Licht im Raum der mit $x_i$ verbundenen Schalter brennt

#### 12.7.3.7.2. 3SAT

Wie SAT, nur hat jede Klausel der Normalform maximal 3 Literale.

**Beispiel** *31:* **SAT $\rightarrow$ 3SAT**

$$(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_5 \vee x_6)$$

$$\rightarrow (x_1 \vee x_2 \vee z_1) \wedge (\overline{z_1} \vee x_3 \vee x_4) \wedge (x_5 \vee x_6 \vee x_6)$$

#### 12.7.3.8. Weitere

##### 12.7.3.8.1. SUBSET-SUM

Gegeben einer Liste $S$ von natürlichen Zahlen $s_i \in \mathbb{N}$ und einer natürlichen Zahl $t \in \mathbb{N}$, ist es möglich eine teilliste $T \subset S$ auszuwählen, sodass die Elemente von $T$ die Summe $t = \sum_{s \in T} s$ haben?

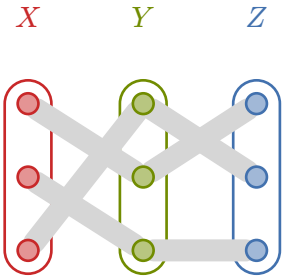
$$\begin{array}{rl} S = & [a,\ b,\ c,\ d,\ e] \\ & \downarrow \quad \downarrow \quad \downarrow \\ T = & [a, \quad c, \quad e] \\ t \overset{!}{=} & a \ + c + \ e \end{array}$$

**Beispiel 32:** **Rucksack-Problem (Ziel: Rucksack füllen)**

$\boldsymbol{S}$ = Grössen der Gegenstände, $\boldsymbol{t}$ = Max. Platz im Rucksack, $\boldsymbol{T \subset S}$ = Grössen der Gegenstände, die in den Rucksack passen

##### 12.7.3.8.2. 3D-MATCHING

Gegeben sind drei endliche Mengen $X, Y$ und $Z$ mit gleich vielen $n = |X| = |Y| = |Z|$ Elementen und $T \subset X \times Y \times Z$. Gibt es eine $n$-elementige Teilmenge $M \subset T$ derart, dass keine zwei Tripel von $M$ in einer Koordinate übereinstimmen?




**Beispiel 33:** **Ein Koch hat $n$ Rezepte für Vorspeisen, Hauptspeisen und Desserts. Nicht alle Vorspeisen lassen sich mit jeder Hauptspeise kombinieren, dasselbe gilt auch für Desserts. Er möchte $n$ Menus zusammenstellen, sodass jedes Rezept in genau einem der Menus vorkommt.**

$\boldsymbol{X}$ = Vorspeisen, $\boldsymbol{Y}$ = Hauptspeisen, $\boldsymbol{Z}$ = Desserts, $\boldsymbol{T \subset X \times Y \times Z}$ = Menge aller Menus, $\boldsymbol{M \subset T}$ = Menge der zusammengestellter Menus

##### 12.7.3.8.3. BIP

BIP = Binary Integer Programming

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

$$\text{BIP} = \left\{ \langle C, d \rangle \middle| \begin{array}{l} \text{Zu } C \in M_{n \times m}(\mathbb{Z}) \text{ und } d \in \mathbb{Z}^n \text{ gibt es einen} \\ \text{binären Vektor } x \in \{0,1\}^m \text{ derart, dass } Cx = d \end{array} \right\}$$

**Beispiel 34:** $C = \begin{pmatrix} 1 & 3 & 0 \\ 0 & 2 & 5 \end{pmatrix} \quad d = \begin{pmatrix} 1 \\ 5 \end{pmatrix}$

$$x = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad \text{da} \quad \begin{pmatrix} 1 & 3 & 0 \\ 0 & 2 & 5 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \end{pmatrix}$$

# 13. TURING-VOLLSTÄNDIGKEIT

| *Turing-Maschine* | *Moderner Computer* |
|---|---|
| Zustände Q | Zustände der CPU: Statusbits, Registerwerte |
| Band, d. h. ein unendlich grosser Speicher | virtueller Speicher: praktisch unbegrenzter Speicher |
| Schreib-/Lesekopf | Adress-Register, Programm-Zähler |
| Anhalten, qaccept und qreject | exit(EXIT_SUCCESS), exit(EXIT_FAILURE) |
| problemspezifisch | kann beliebige Programme ausführen |

## 13.1. DINGE, DIE EINER TM FEHLEN

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

### *Persistenter Speicher*

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

### *Interaktion*

Lösung eines vollständig spezifizierten Problems braucht keine Interaktion

### *Input/Output*

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

  Resultat: Input-Daten stehen irgendwo auf dem Band

$\Rightarrow$ Diese Punkte ändern nichts wesentliches

## 13.2. UNIVERSELLE TURING-MASCHINE

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

### *Konstruktion*

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

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

$$M_2 \leq_S M_1$$

lies: $M_2$ ist simulierbar auf $M_1$

## 13.3. PROGRAMMIERSPRACHEN UND TURING-VOLLSTÄNDIGKEIT

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.

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

## 13.4. GRUNDELEMENTE FÜR SPRACHEN

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

## 13.5. LOOP

```
LOOP x_i DO
  P
END
```

Führt das Teilprogramm $P$ genauso oft aus, wie $x_i$ beim Eintritt in die Schleife angibt. Der Wert von $x_i$ wird vor der ersten Ausführung von $P$ ermittelt. Eine Veränderung des Wertes von $x_i$ innerhalb von $P$ hat keinen Einfluss auf die Anzahl der Ausführungen von $P$. Somit ist LOOP nicht Turing-Vollständig, da keine unendliche Schleife erzeugt werden kann.

## 13.6. WHILE

```
WHILE x_i > 0 DO
  P
END
```

Führt den Block $P$ so lange aus, als $x_i > 0$ ist. Der Vergleich $x_i > 0$ ist die einzige Bedingung, die zur Verfügung steht. Im Gegensatz zu LOOP wird erwartet, dass der Programmteil $P$ beim Erreichen einer ausreichend großen Anzahl Iterationen die Variable $x_i$ auf $0$ setzt und damit die Schleife terminiert.

WHILE ist Turing-Vollständig.

## 13.7. GOTO

```
M_k : P
M_l : IF x_i > 0 THEN GOTO M_k
```

Ein WHILE-Programm kann immer in ein gleichwertiges GOTO-Programm übersetzt werden und umgekehrt. GOTO ist also auch Turing-Vollständig.

# BIBLIOGRAFIE

[1] A. Müller, «Reguläre Sprachen», in ***Automaten und Sprachen: Theoretische Informatik für die Praxis: Mathematik, Anwendung, Intuition***, Berlin, Heidelberg: Springer Berlin Heidelberg, 2024. doi: 10.1007/978-3-662-70146-1_1 .