Theoretische Grundlagen der Informatik

Skript zur Vorlesung von
Prof. Dr. Dorothea Wagner
WS 98/99

ausgearbeitet von
Marco Gaertler und Dagmar Handke

Vorwort

Diese Vorlesungsausarbeitung beruht auf der Vorlesung *Theoretische Grundlagen der Informatik*, die ich im Wintersemester 1998/1999 an der Universität Konstanz gehalten habe. Im Sommersemester 1993 und im Wintersemester 1995/1996 habe ich in Halle bzw. Konstanz Vorlesungen etwa gleichen Inhalts angeboten. Inhaltlich habe ich mich dabei sehr stark auf das Buch *Theoretische Informatik* von Ingo Wegener (erschienen 1993 bei Teubner) und "den Garey & Johnson", also das Buch *Computers and Intractability: A Guide to the Theory of NP-Completeness* von Michael R. Garey und David S. Johnson, gestützt.

Marco Gaertler hat auf Basis seiner eigenen Mitschrift und meinen handschriftlichen Unterlagen dieses Skript ausgearbeitet und nach ausführlichen Vorschlägen von Dagmar Handke überarbeitet. Beiden möchte ich für diese Arbeit ganz herzlich danken. Von verschiedenen Studierenden, die im Wintersemester 1998/1999 meine Vorlesung besucht haben, wurden Kommentare und Korrekturvorschläge beigetragen, wofür ich mich ebenfalls bedanke.

Das vorliegende Skript ist eine reine Vorlesungsausarbeitung, kein Lehrbuch. Entsprechend ist es sprachlich eher knapp gehalten und inhaltlich sicher an einigen Stellen weit weniger umfangreich, als es ein entsprechendes Lehrbuch sein sollte. Ich hoffe, dass es trotzdem den Studierenden bei der Prüfungsvorbereitung hilfreich sein wird und als Unterstützung zukünftiger Vorlesungen zu diesem Thema geeignet ist.

Konstanz, im Juli, 2000 Dorothea Wagner

Inhaltsverzeichnis

Kapitel 1

Einführung

Inhalt: Theoretische Grundlagen der Informatik

Im Gegensatz zu Vorlesungen wie „Einführung in die Rechner–Architektur“ oder „Datenstrukturen und effiziente Algorithmen“ werden hier Themen behandelt, die weiter von den Anwendungen entfernt sind. Es geht um prinzipielle Fragestellungen, d.h. Fragen, die zum Beispiel unabhängig von „Programmierungsaspekten“ oder „konkreten Rechnern“ sind.

Typische Fragestellungen:

- Gibt es Aufgaben, die von einem Rechner — unabhängig von der Art der Programmierung beziehungsweise von physikalischen und elektronischen Beschränkungen — nicht gelöst werden können?
- Welche Aufgaben können — prinzipiell — effizient (in vernünftiger Rechenzeit, mit vernünftigem Speicherplatzbedarf) gelöst werden?

Um diese Fragestellungen sinnvoll zu behandeln, müssen Konzepte entwickelt werden wie:

- ein grundlegendes (naives?) Rechnermodell
- eine grundlegende Problemformulierung

Dafür muß geklärt werden wie ein Rechner (der nur „Nullen“ und „ Einsen“ kennt) ein Problem überhaupt löst.

1.1 Einführende Beispiele — Automatenmodell, Spracherkennung, Entscheidungsproblem

Beispiel: Automatenmodell

Wir betrachten einen „primitiven Fahrkartenautomaten“ mit folgenden Funktionen. Der Automat

- kennt nur eine Sorte von Fahrscheinen zum Preis von 3 DM,
- akzeptiert nur Münzen zu 1 DM und 2 DM,
- gibt kein Wechselgeld heraus.
- warnt bei Überzahlung (zweimal 2 DM) und erwartet eine Reaktion (er spuckt sonst jede weitere Münze aus):
  - **Ü**berzahlungsbestätigung („habe absichtlich überbezahlt“)
  - **R**eset (gib alles „mögliche“ Geld zurück; vergiß was sonst gelaufen ist ...)

Modell für den Fahrkartenautomaten

Unser Modell vernachlässigt die Kartenausgabe und die Münzprüfung. Der Automat muß sich merken, wieviel bezahlt wurde, und den Wert in Abhängigkeit von der Eingabe ändern.

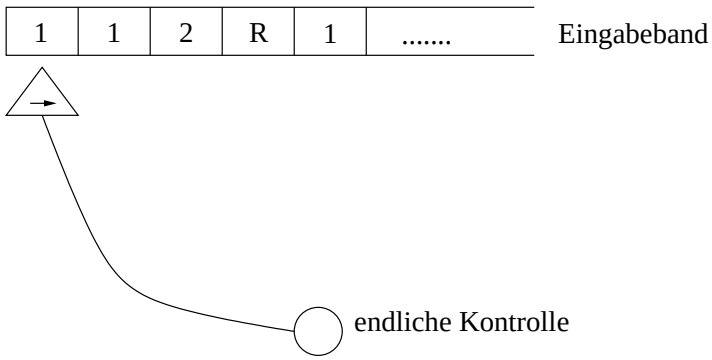

Abbildung 1.1: Modell für den Fahrkartenautomaten

Das Eingabeband enthält ein Folge von Zeichen aus $\{1, 2, \ddot{\mathrm{U}}, \mathrm{R}\}$. Die endliche Kontrolle kennt verschiedene Zustände: 0, 1, 2, 4, A(usgabe), $A \oplus 1$, die den aktuellen Zustand des Automaten beschreiben. Das Eingabeband wird zeichenweise gelesen und bewirkt *Übergänge* von einem Zustand zu einem anderen Zustand. Zur Beschreibung der endlichen Kontrolle dient ein Graph:

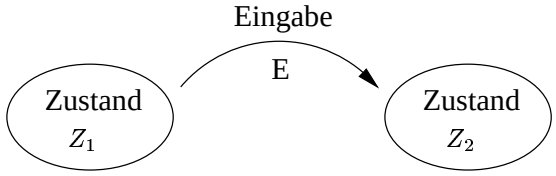

Abbildung 1.2: Zustandsübergang

Dieser Graph entspricht folgender Aktion: Befindet sich der Automat im Zustand Z_1, so geht er bei Eingabe von E in den Zustand Z_2 über.

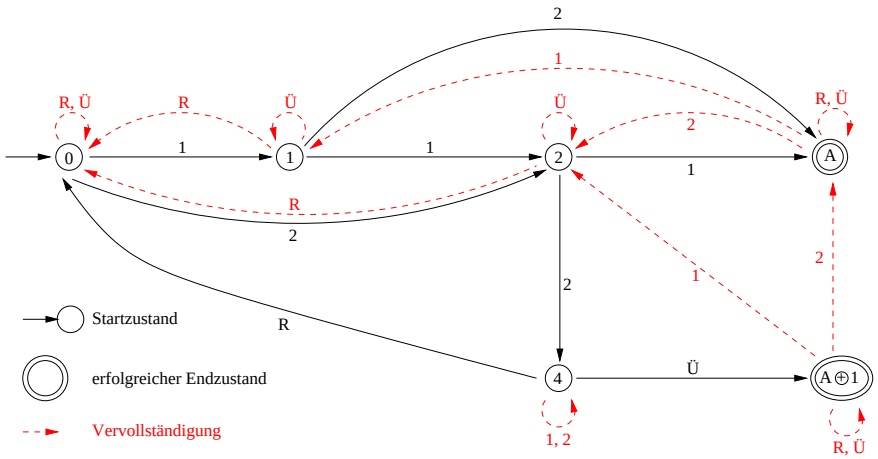

Abbildung 1.3: Übergangsgraph des Fahrkartenautomaten

Für den Fahrkartenautomaten ergibt sich folgender Übergangsgraph (siehe Abbildung 1.3).

Der Automat erkennt (d.h. akzeptiert) alle korrekten Bedienungsvorgänge (Eingabefolgen), d.h. alle Folgen, die in A oder $A \oplus 1$ enden. Möglicherweise werden mehrere Fahrkarten hintereinander erworben. Dieser Graph ist jedoch unvollständig: zum Beispiel kann die Eingabefolge $1, 1, R$ (Kunde hat zuwenig Geld) nicht abgearbeitet werden. Der Graph muß also vervollständigt werden.

Diese Art von Automaten ist offenbar leicht realisierbar. ■

Beispiel Spracherkennung:
Wir betrachten folgendes Problem: Ein Rechner (Automat) soll ein Programm einer bestimmten Programmiersprache bearbeiten.

Dabei muß zum Beispiel folgendes Teilproblem gelöst werden: „Handelt es sich bei einem gegebenem Stück Programmtext (Zeichenfolge) um einen Namen (zum Beispiel von Variablen)?“ Eine Beschreibungsform für die Syntax von Programmiersprachen ist zum Beispiel die **Backus-Naur-Form**. Namen lassen sich darin wie folgt beschreiben:

```
    <Name>   ::=  <Buchstabe> {<Symbol>}
  <Symbol>   ::=  <Buchstabe> | <Ziffer>
<Buchstabe>  ::=  A | B | C | ... | Z
   <Ziffer>  ::=  0 | 1 | ... | 8 | 9
```

Dabei steht | für eine Alternative und { } für eine beliebigfache Wiederholung, zu ersetzende Variablen werden durch < > gekennzeichnet.

Für die Konstruktion eines Automaten zur Erkennung von Namen gehen wir davon aus, daß die Eingabe nur aus Symbolen (also Buchstaben oder Ziffern und nicht Sonderzeichen oder ähnliches) bestehe (siehe Abbildung 1.4).

■

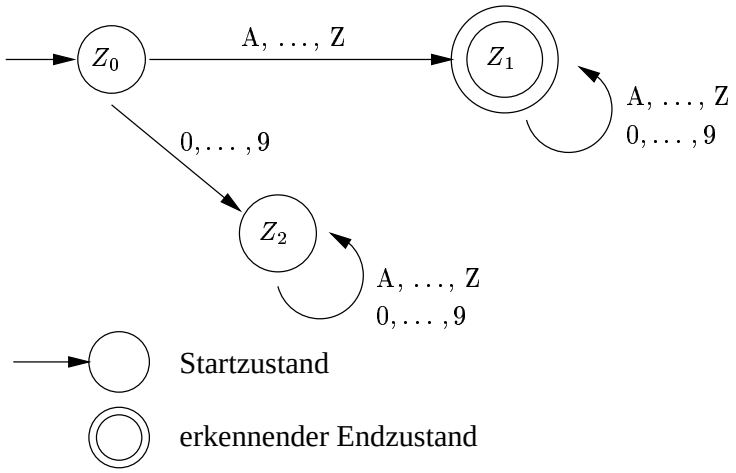

Abbildung 1.4: Dieser Automat erkennt alle korrekten Namen

Beispiel: Entscheidungsprobleme:

(1) Primzahlen:

Gegeben: Eine natürliche Zahl

Frage: Ist die Zahl eine Primzahl?

Aufgabe: Konstruiere einen Automaten, der alle Primzahlen erkennt.

(2) Lösen von Gleichungssystemen:

Gegeben: Ein lineares Gleichungssystem (z.B. in Form einer Matrix)

Frage: Ist das Gleichungssystem lösbar? (Dazu äquivalent: Ist das Gleichungssystem invertierbar?)

Aufgabe: Erkenne alle invertierbaren Matrizen.

■

Die zentrale Aufgabe der theoretischen Informatik ist das Erkennen einer „Sprache“.

Die Ziele sind dabei:

1. Exakte Formulierung der verwendeten Begriffe.

   Zum Beispiel: Automat, Zustand, Sprache, ...

2. Beantwortung der Frage „Was bedeutet es, daß ein Rechner ein Problem löst?“ mit Hilfe eines formalen Begriffapparates

3. Aussagen über Möglichkeiten und Grenzen dieser Konzepte

Kapitel 2

Endliche Automaten und reguläre Ausdrücke

2.1 Deterministische endliche Automaten und formale Sprachen

2.1 Definition
Ein (deterministischer) endlicher Automat (D)EA besteht aus:

- Q, *einer endlichen Menge von* ***Zuständen***;
- Σ, *einer endlichen Menge von* ***Eingabesymbolen***, ***Alphabet***;
- $\delta\colon Q \times \Sigma \to Q$, *einer* ***Übergangsfunktion***;
- $s \in Q$, *einem* ***Startzustand***;
- $F \subseteq Q$, *einer Menge von* ***Endzuständen***.

Bemerkung:
Der Automat heißt

- endlich, da die Zustandsmenge (vgl. mit Speicher, Gedächtnis) endlich ist;
- deterministisch, da δ eine Funktion ist und der Automat somit in jedem Schritt eindeutig arbeitet. Es gibt keine Zufälligkeiten oder Wahlmöglichkeiten.

Notation
Wir bezeichnen endliche Automaten auch kurz mit $(Q, \Sigma, \delta, s, F)$.

Was kann ein endlicher Automat?

Gegeben ist eine Eingabe als endliche Folge von Eingabesymbolen. Der Automat entscheidet, ob die Eingabe zulässig ist oder nicht, indem er in einem

Endzustand endet oder nicht.

Formales Sprachkonzept

2.2 Definition

- *Ein endliches* ***Alphabet*** Σ *ist eine endliche Menge von Symbolen.*
- *Eine endliche Folge von Symbolen aus* Σ *heißt* ***Wort*** *(über* Σ*).*
- *Die Menge aller Wörter über* Σ *heißt* $\boldsymbol{\Sigma^*}$*.*
- *Die Anzahl der Symbole eines Wortes* w *ist die* ***Länge*** *von* w*, sie wird durch die Kardinalität von* w *(*$|w|$*) bezeichnet.*
- *Das* ***leere Wort*** *heißt* ε *(*$|\varepsilon| = 0$*); es gilt* $\varepsilon \in \Sigma^*$ *für alle* Σ*.*
- *Aus zwei Wörtern* w_1, w_2 *erhält man die* ***Konkatenation****, d.h. ein Wort* $w = w_1 \cdot w_2$*, durch Hintereinanderschreiben.*

$$w^i := \underbrace{w \cdot \ldots \cdot w}_{i-Mal} \qquad w^0 := \varepsilon$$

Oft schreiben wir statt $w_1 \cdot w_2$ *auch nur* $w_1 w_2$*.*

Beispiele:

(1) Sei $\Sigma := \{0, 1\}$. Dann ist

$$\Sigma^* = \{ \underbrace{\varepsilon}_{\text{Länge 0}} , \underbrace{0, 1}_{\text{Länge 1}} , \underbrace{00, 01, 10, 11}_{\text{Länge 2}}, \underbrace{000, 001, 010, 011, \ldots}_{\text{Länge 3}}, \ \ldots \}$$

(2) $01 \cdot 01 = 0101, \qquad 01 \cdot \varepsilon = 01$

■

2.3 Definition

Eine Menge L *von Wörtern über einem Alphabet* Σ*, d.h.* $L \subseteq \Sigma^*$*, heißt* ***(formale) Sprache*** *über* Σ*.*

Beispiele:

(1) Sei $\Sigma = \{\text{A}, \ldots, \text{Z}\}$.

$L := \{w \in \Sigma^* \mid w \text{ ist ein Wort der deutschen Sprache}\}$
$= \{\text{AAL, AAS, AASEN, AASFLIEGE, AASGEIER, } \ldots \}$(siehe Duden)

(2) Sei $\Sigma = \{0, 1, \ldots, 9\}$.
$L := \{w \in \Sigma^* \mid w \text{ ist Primzahl}\} = \{2, 3, 5, \ldots\}$

(3) Sei Σ beliebig, $a \in \Sigma$.
Jede Sprache über Σ ist Element der Potenzmenge 2^Σ, zum Beispiel

$$L = \Sigma^*, \ L = \emptyset, \ L = \{\varepsilon\}, \ L = \{a\}, \ \ldots$$

■

2.4 Definition
Läßt sich ein Wort w schreiben als $w = u \cdot v \cdot x$, wobei u, v, x beliebige Wörter sind, so heißt:

$$\left.\begin{array}{ll} u & \text{Präfix} \\ v & \text{Teilwort} \\ x & \text{Suffix} \end{array}\right\} \text{von } w$$

Beispiel :
Das Wort TAL hat:

Präfixe	$P = \{\varepsilon, \text{T}, \text{TA}, \text{TAL}\}$
Suffixe	$S = \{\varepsilon, \text{L}, \text{AL}, \text{TAL}\}$
Teilworte	$\{A\} \cup P \cup S$

■

Konstruktion weiterer Sprachen aus bereits bestehenden Sprachen

2.5 Definition
Seien $L, L_1, L_2 \subseteq \Sigma^*$ Sprachen.

Produktsprache: $L_1 \cdot L_2 := \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$

k–faches Produkt: $L^k := \{w_1 \cdot w_2 \cdot \ldots \cdot w_k \mid w_i \in L \text{ für } 1 \leq i \leq k\}$;
$L^0 := \{\varepsilon\}$

Quotientensprache: $L_1/L_2 := \{w \in \Sigma^* \mid \exists\, z \in L_2 \text{ mit } w \cdot z \in L_1\}$

Kleene'scher Abschluß: $L^* := \bigcup_{i \geq 0} L^i = \{w_1 \cdot \ldots \cdot w_n \mid w_i \in L, n \in \mathbb{N}_0\}$

positiver Abschluß: $L^+ := \bigcup_{i > 0} L^i$

Komplementsprache: $L^c := \Sigma^* \backslash L$

Bemerkungen und Beispiele

(1) Σ^* ist Kleene'scher Abschluß von Σ.

(2) Sei $L_1 = \{0, 10\}$ und $L_2 = \{\varepsilon, 0\}$. Dann sind:

$$\begin{aligned} L_1 \cdot L_2 &= \{0, 10, 00, 100\} \\ L_1^* &= \{\underbrace{\varepsilon}_{L_1^0}, \underbrace{0, 10}_{L_1^1}, \underbrace{00, 010, 100, 1010}_{L_1^2}, \ldots\} \\ L_1/L_2 &= \{\varepsilon, 1, 0, 10\} \end{aligned}$$

■

2.6 Definition

- *Ein endlicher Automat* ***erkennt*** *oder* ***akzeptiert*** *eine Sprache L, d.h. eine Menge von Wörtern über dem Alphabet des Automaten, wenn er nach Abarbeitung eines Wortes w genau dann in einem Endzustand ist, wenn das Wort w in der Sprache L ist ($w \in L$).*
- *Eine formale Sprache heißt* ***endliche Automatensprache****, wenn es einen endlichen Automaten gibt, der sie erkennt.*

Frage: Welche Sprachen sind endliche Automatensprachen?

2.7 Definition
*Eine Sprache $L \subseteq \Sigma^*$ heißt **regulär**, wenn für sie einer der folgenden Punkte gilt: (induktive Definition)*

I. Verankerung:

(1) $L = \{a\}$ mit $a \in \Sigma$ oder
(2) $L = \emptyset$

II. Induktion: Seien L_1, L_2 reguläre Sprechen

(3) $L = L_1 \cdot L_2$ oder
(4) $L = L_1 \cup L_2$ oder
(5) $L = L_1^*$

Regulär sind also die Sprachen, die sich aus Sprachen vom Typ (1), (2) oder durch endlich viele Operationen vom Type (3), (4), (5) erzeugen lassen.

Bemerkungen und Beispiele

(1) Die Sprache aller Wörter über $\{0,1\}$, die als vorletztes Zeichen eine 0 haben:

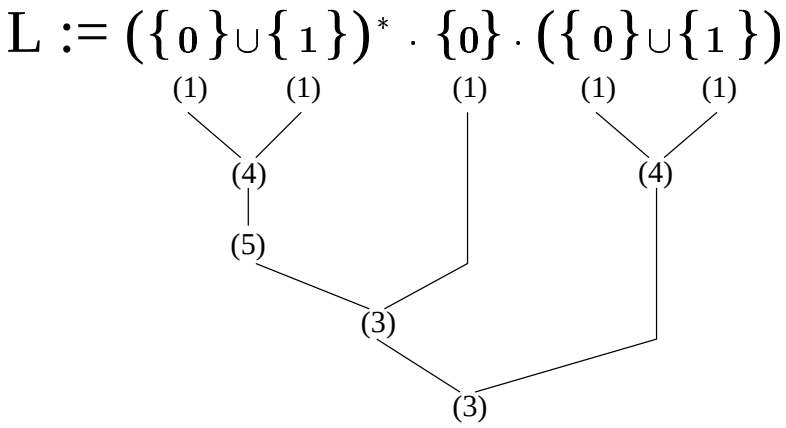

Abbildung 2.1: Aufbau der Sprache L

(2) $\{\varepsilon\}$ ist eine reguläre Sprache, denn $\{\varepsilon\} = \emptyset^*$. ■

2.8 Definition
*Sei Σ eine Alphabet. Eine reguläre Sprache über Σ kann durch einen **regulären Ausdruck** beschrieben werden. Dabei bezeichnet:*

- *$\emptyset$ den regulären Ausdruck, der die leere Menge beschreibt.*
- *ε den regulären Ausdruck, der die Menge $\{\varepsilon\}$ beschreibt.*

- $\boldsymbol{a}$ *den regulären Ausdruck, der die Menge* $\{a\}$ *beschreibt.*

Wenn α, β *reguläre Ausdrücke sind, die die Sprachen* $L(\alpha), L(\beta)$ *beschreiben, so schreiben wir* $\alpha \cup \beta, \alpha \cdot \beta, \alpha^+$ *bzw.* a^* *für die regulären Ausdrücke, die die Sprachen* $L(\alpha) \cup L(\beta), L(\alpha) \cdot L(\beta), L(\alpha)^+$ *bzw.* $L(\alpha)^*$ *beschreiben.*

Notation
Wir schreiben auch α statt $L(\alpha)$ und $w \in \alpha$ statt $w \in L(\alpha)$.

Beispiele:

(1) $L := \mathbf{(0 \cup 1)^* 0 (0 \cup 1)}$ ist der reguläre Ausdruck für die Sprache des obigen Beispiels.

(2) $L := \{w \in \{0,1\}^* \mid w \text{ enthält 10 als Teilwort}\} = \mathbf{(0 \cup 1)^* 10 (0 \cup 1)^*}$

(3) $L := \{w \in \{0,1\}^* \mid w \text{ enthält 10 nicht als Teilwort}\} = \mathbf{0^* 1^*}$, denn

- $w \in \mathbf{0^* 1^*} \Rightarrow w \in L$; also ist $\mathbf{0^* 1^*} \subseteq L$.
- Sei $w \in L$. Dann kommen nach der ersten Eins keine Nullen mehr vor, d.h. $w = w'1 \ldots 1$ wobei w' keine 1 enthält. Also ist $w \in \mathbf{0^* 1^*}$.

(4) $L := \{w \in \{0,1\}^* \mid w \text{ enthält 101 als Teilwort}\} = \mathbf{(0 \cup 1)^* 101 (0 \cup 1)^*}$

(5) $L := \{w \in \{0,1\}^* \mid w \text{ enthält 101 nicht als Teilwort}\}$

Sei $w \in L$ und w enthalte 10 genau n-mal als Teilwort ($n > 0$). D.h.

$$w = w_1 10 w_2 10 \ldots w_n 10 w_{n+1} = \left(\prod_{i=1}^{n} (w_i 10) \right) w_{n+1},$$

wobei w_i 10 nicht enthält. Da $w \in L$, darf 101 nicht vorkommen. Das bedeutet, daß w_i mit Null beginnt für alle $i > 1$. Ausnahmen sind w_1, das auch mit einer Eins beginnen darf bzw. leer sein kann, und w_{n+1}, das auch leer sein darf.

Also gilt

$$w = \left(\prod_{i=0}^{n-1} (v_i 100) \right) v_n 10 w_{n+1}$$

mit $v_i \in \mathbf{0^* 1^*}$ für $1 \leq i \leq n$ und $w_{n+1} \in \varepsilon \cup \mathbf{0(0^* 1^*)}$.

Also für ein $w \in L$, in dem 10 n-mal vorkommt, gilt:

$$w \in \left(\prod_{i=1}^{n-1} \mathbf{0^* 1^* 100} \right) \mathbf{0^* 1^* 10} (\varepsilon \cup \mathbf{0(0^* 1^*)})$$

$$\Rightarrow w \in (\mathbf{0^* 1^* 100})^* \mathbf{0^* 1^* 10} (\varepsilon \cup \mathbf{00^* 1^*})$$

Insgesamt haben wir

$$L = \underbrace{\mathbf{0^* 1^*}}_{(*)} \cup (\mathbf{0^* 1^* 100})^* \mathbf{0^* 1^* 10} (\varepsilon \cup \mathbf{00^* 1^*})$$

$(*)$ 10 kommt kein Mal vor.

■

Ziel:

- Welche Sprachen werden durch endliche Automaten erkannt?
- Welche Beschreibung für diese Sprachen gibt es?
- Gibt es zu jeder regulären Sprache einen endlichen Automaten, der diese erkennt?
- Konstruktion eines erkennenden Automaten für reguläre Sprachen.

Zunächst widmen wir uns der „Erkennbarkeit".

2.9 Satz
Jede reguläre Sprache wird von einem (deterministischen) endlichen Automaten (DEA) akzeptiert.

Beweis: Sei L eine reguläre Sprache über Σ, d.h. L sei durch einen regulären Ausdruck beschreibbar. Sei n die Zahl der „$\cup$", „$\cdot$" bzw. „*"–Zeichen in diesem Ausdruck.

Induktion über n; also über die Struktur von L.

Induktionsanfang $n = 0$: D.h. $L = \emptyset$ oder $L = a$, dann existieren dazu DEA (siehe Abbildung 2.2).

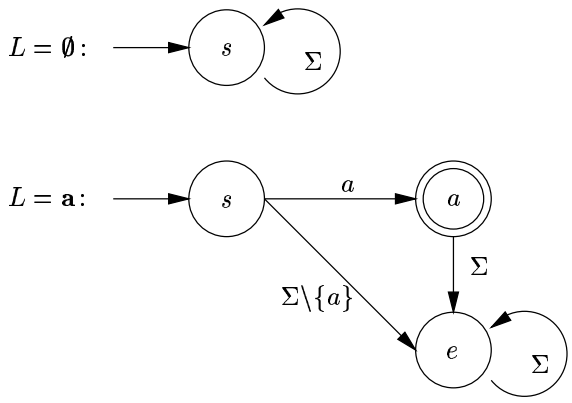

Abbildung 2.2: DEA's für den Induktionsanfang

Induktionsschluß: Annahme: Es existieren deterministische endliche Automaten für alle Sprachen, die durch reguläre Ausdrücke mit n Zeichen aus $\{\cup, \cdot, ^* \}$ beschreibbar sind. Nun soll mindestens ein Zeichen mehr enthalten sein.

- Sei $L = L_1 \cup L_2$, wobei L_1 und L_2 reguläre Sprachen sind, die durch einen regulären Ausdruck mit n oder weniger Zeichen beschreibbar sind. Nach Induktionsannahme existieren deterministische endliche Automaten, die L_1 bzw. L_2 akzeptieren. Die entsprechenden DEAs seien $\mathcal{A}_i := (Q_i, \Sigma, \delta_i, s_i, F_i)$ für L_i für $i = 1, 2$.

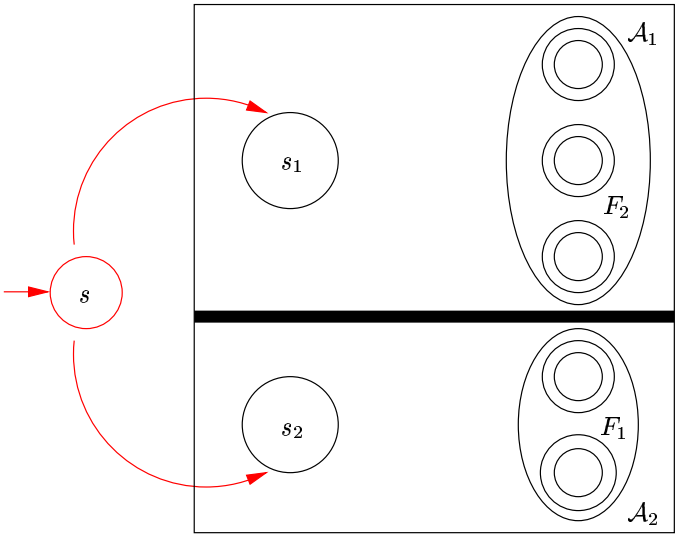

Abbildung 2.3: EA für $L = L_1 \cup L_2$

Erster Versuch, daraus einen deterministischen endlichen Automaten für $L_1 \cup L_2$ zu konstruieren:
Der deterministische endliche Automat $(Q, \Sigma, \delta, s, F)$ für L müßte dann wie folgt aussehen:

- $Q = Q_1 \cup Q_2 \cup \{s\}$ (es sind die Zustände in Q_1 bzw. Q_2 entsprechend zu benennen, daß $Q_1 \cap Q_2 = \emptyset$)
- $F = F_1 \cup F_2$
- $\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{falls } q \in Q_1 \\ \delta_2(q,a) & \text{falls } q \in Q_2 \\ ? & \text{falls } q = s \end{cases}$

Es müßte einen Übergang ohne Lesen eines Zeichens geben mit Wahlmöglichkeit. Dies geht jedoch nicht in einem DEA. Neuer Versuch über den „Umweg“ der nichtdeterministischen endlichen Automaten. Der Beweis von Satz 2.9 folgt später.

2.2 Nichtdeterministische endliche Automaten

2.10 Definition
*Ein **nichtdeterministischer endlicher Automat** (NEA) besteht aus:*

- *Q, einer Zustandsmenge;*
- *Σ, einem Alphabet;*
- *δ, einer Übergangsfunktion $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$, wobei 2^Q die Potenzmenge von Q darstellt.*

D.h., bei der Abarbeitung eines Eingabesymbols aus Σ kann der Automat sich — nichtdeterministisch — aussuchen, in welchen Zustand aus einer Teilmenge von Q er geht. Er kann auch ohne Lesen eines Eingabesymbols „spontan" sogenannte ε-Übergänge ausführen. $\delta(q,a)$ kann auch $\emptyset$ sein, d.h. es gibt zu q bei Lesen von a keinen Folgezustand.

- *s, einem Startzustand;*
- *F, einer Menge von Endzuständen;*

Ein nichtdeterministischer endlicher Automat **akzeptiert** eine Wort $w \in \Sigma^*$, wenn es eine Folge von Übergängen gibt (auch ε-Übergänge), so daß er bei Eingabe von w in einen Endzustand gelangt, d.h. bei Eingabe von w ein Endzustand *erreichbar* ist.

Beispiel :
Der Automat zur Erkennung von $L = L_1 \cup L_2$ aus obigem Beweisversuch ist ein nichtdeterministischer endlicher Automat mit:

$$\delta(q,a) = \begin{cases} \delta_i(q,a)\} & \text{falls } q \in Q_i,\ a \in \Sigma \\ \{s_1, s_2\} & \text{falls } q = s,\ a = \varepsilon \\ \emptyset & \text{falls } q \in Q_1 \cup Q_2,\ a = \varepsilon \\ \emptyset & \text{falls } q = s,\ a \in \Sigma \end{cases}$$

Je nach Verzweigung aus s heraus kann der nichtdeterministische endliche Automat in einem Zustand aus F enden, oder nicht. ■

Anscheinendend sind nichtdeterministische endliche Automaten wesentliche flexibler und mächtiger als deterministische !?

Wir werden sehen, daß dies jedoch nicht der Fall ist.

Notation: ε-Abschluß Für einen Zustand $q \in Q$ ist der ε–**Abschluß $E(q)$** wie folgt definiert:

$$E(q) := \{p \in Q \mid p \text{ ist von } q \text{ durch eine Folge von } \varepsilon\text{–Übergängen erreichbar}\}$$

Beachte es gilt:

- $E(q) \subseteq Q, \quad E(q) \in 2^Q$
- $q \in E(q)$

Erweiterung von δ

Um ε–Übergänge bei der Übergangsfunktion δ berücksichtigen zu können, müssen wir δ geeignet erweitern.

1. berücksichtige ε-Abschluß:

$$\widehat{\delta}\colon Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$$

$$\widehat{\delta}(q,a) = \begin{cases} E(q) & \text{falls } a = \varepsilon \\ \bigcup\limits_{p \in \delta(q,a)} E(p) & \text{für } a \in \Sigma \end{cases}$$

2. mehrere Ausgangszustände, berücksichtige ε-Abschluß:

$$\widehat{\delta}\colon 2^Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$$

$$\widehat{\delta}(P,a) = \begin{cases} \bigcup\limits_{p\in P} E(p) & \text{falls } a = \varepsilon \\ \bigcup\limits_{p\in P} \widehat{\delta}(p,a) & \text{für } a \in \Sigma \end{cases}$$

3. Erweiterung auf Wörter:

$$\widehat{\delta}\colon Q \times \Sigma^* \to 2^Q$$

$$\widehat{\delta}(q,w) = \left\{ p \in Q \;\middle|\; \begin{array}{l} \text{es gibt eine beliebige Folge von } \delta\text{–Über–} \\ \text{gängen, so daß nach der Abarbeitung von } w \\ \text{der Zustand } p \text{ erreicht wird} \end{array} \right\}$$

$$\widehat{\delta}(q,v) = \begin{cases} E(q) & \text{falls } v = \varepsilon \\ \bigcup\limits_{p\in\widehat{\delta}(q,w)} \widehat{\delta}(p,a) & \text{falls } v = wa \end{cases}$$

4. Analog für den Fall, daß mehrere Ausgangszustände berücksichtigt werden:

$$\widehat{\delta}\colon 2^Q \times \Sigma^* \to 2^Q$$

$$\widehat{\delta}(P,w) = \bigcup_{p\in P} \widehat{\delta}(p,w)$$

Oft schreiben wir δ auch an Stellen, an denen eigentlich $\widehat{\delta}$ verwendet werden müßte.

2.11 Definition
Zwei endliche Automaten, die dieselbe Sprache akzeptieren, heißen ***äquivalent****.*

2.12 Satz (Äquivalenz von NEA's und DEA's)
Zu jedem nichtdeterministischen endlichen Automaten gibt es einen äquivalenten deterministischen endlichen Automaten.

Beweis Potenzmengenkonstruktion:
Gegeben sei ein nichtdeterministischer endlicher Automat $\mathcal{A}$. Eine Abarbeitung eines Wortes w in $\mathcal{A}$ besteht aus einer Folge von ε-Übergängen und „echten" Übergängen. Bei jedem echten Übergang wird ein Symbol abgearbeitet. Falls der nichtdeterministische endliche Automat ein Wort w akzeptiert, dann gibt es eine Abarbeitung, die in einem Endzustand endet. Der nichtdeterministische endliche Automat kann also ohne Abarbeitung eines Buchstabens in alle Zustände des ε-Abschlußes übergehen. Ist ein Zustand q bei der Eingabe eines Wortes erreichbar, so sind dies auch alle Zustände aus $E(q)$.

In einem DEA ist die Abarbeitung eines Wortes eindeutig, und es gibt nur „echte" Übergänge.

Idee: Jede Abarbeitung des nichtdeterministischen endlichen Automaten wird durch den deterministischen endlichen Automaten simuliert. Die Zustände des

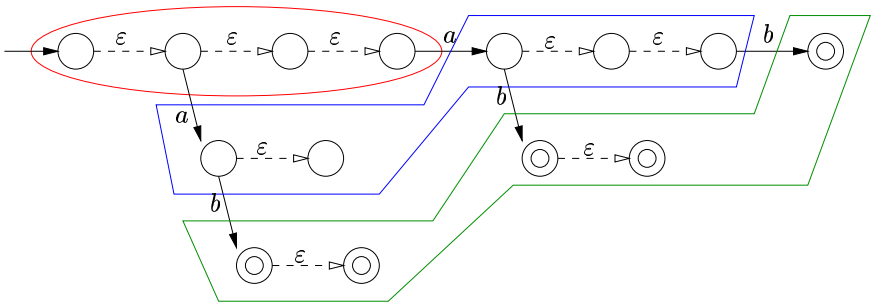

Abbildung 2.4: Mögliche Abarbeitungen von $w = ab$ in einem NEA

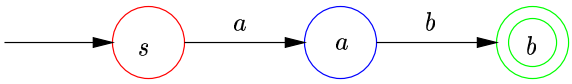

Abbildung 2.5: Die Abarbeitung von $w = ab$ bei einem äquivalenten DEA

DEA bestehen dafür aus Menge von Zuständen des NEA. Falls der nichtdeterministische endliche Automat das Wort w akzeptiert, dann gibt es eine Abarbeitung, die in einem Endzustand endet. Der deterministische endliche Automat muß also auch in einem seiner Endzustände enden. Dies sind die Zustände ($\equiv$ Menge von Zuständen des nichtdeterministischen endlichen Automaten), die einen Endzustand des nichtdeterministischen endlichen Automaten enthalten.

Potenzmengenkonstruktion: Gegeben sei ein NEA $\mathcal{A} := (Q, \Sigma, \delta, s, F)$. Wir konstruieren daraus einen DEA $\widetilde{\mathcal{A}} := (\widetilde{Q}, \Sigma, \widetilde{\delta}, \widetilde{s}, \widetilde{F})$:

- $\widetilde{Q} = 2^Q$, d.h. die Zustände des DEA sind Mengen von Zuständen des NEA.
- $\widetilde{\delta}\colon \widetilde{Q} \times \Sigma \to \widetilde{Q}$ mit $\widetilde{\delta}(\widetilde{q}, a) = \widehat{\delta}(\widetilde{q}, a)$ für $a \in \Sigma$. Es ist also $\widetilde{q} \subseteq Q$ und jeder Zustand wird mit seinem ε–Abschluß im NEA identifiziert.
- $\widetilde{s} := E(s)$
- $\widetilde{F} := \left\{ \widetilde{q} \in \widetilde{Q} \mid \widetilde{q} \cap F \neq \emptyset \right\}$

Dadurch kann man von einem Zustand $\widetilde{q}$ aus bei der Eingabe $a \in \Sigma$ alle Zustände erreichen, die im ε–Abschluß einer der möglichen Folgezustände eines der $\delta(q, a)$ (für $q \in \widetilde{q}$) liegen.

$\widetilde{\mathcal{A}}$ ist per Konstruktion ein deterministischer endlicher Automat. Es bleibt zu zeigen, daß $\mathcal{A}$ und $\widetilde{\mathcal{A}}$ dieselbe Sprache akzeptieren.

Wir zeigen per Induktion über die Länge der Wörter w, daß für alle $w \in \Sigma^*$ gilt:

$$\widetilde{\delta}(\widetilde{s}, w) = \widehat{\delta}(s, w)$$

Bemerkung: Wir zeigen damit eine schärfere Aussage als nötig, aber dann gilt auch:

$$w \in L(\widetilde{\mathcal{A}}) \Leftrightarrow \widetilde{\delta}(\widetilde{s}, w) \in \widetilde{F} \Leftrightarrow \widehat{\delta}(s, w) \cap F \neq \emptyset \Leftrightarrow w \in L(\mathcal{A})$$

Induktion über $|w|$

Induktionsanfang $|w| = 0$: D.h. $w = \varepsilon$; dann ist $\widetilde{\delta}(\widetilde{s}, w) = \widetilde{s}$.

Konvention: $\widetilde{\delta}(\widetilde{q}, w) = \widetilde{p}$ bedeutet: bei der Abarbeitung von w aus Zustand $\widetilde{q}$ wird Zustand $\widetilde{p}$ erreicht. Dabei ist $w = \varepsilon$ erlaubt. (Vergleiche Erweiterung von δ auf Wörter, Fall 3) Damit ist:

$$\widetilde{\delta}(\widetilde{s}, w) = \widetilde{s} = E(s) = \widehat{\delta}(s, w)$$

Induktionsschluß $|w| = n + 1$: Induktionsvoraussetzung: für alle Wörter w' mit $|w'| \leq n$ gilt $\widetilde{\delta}(\widetilde{s}, w') = \widehat{\delta}(s, w')$. Sei w so, daß $w = w'a$ mit $|w'| = n$ und $a \in \Sigma$ gilt.

$$\begin{aligned} \widetilde{\delta}(\widetilde{s}, w) &= \widetilde{\delta}(\widetilde{\delta}(\widetilde{s}, w'), a) \\ &\underset{\text{(IV)}}{=} \widetilde{\delta}(\widehat{\delta}(s, w'), a) \\ &\underset{\text{Def } \widetilde{\delta}}{=} \widehat{\delta}(\widehat{\delta}(s, w'), a) = \bigcup_{p \in \widehat{\delta}(s, w')} \widehat{\delta}(p, a) = \widehat{\delta}(s, w) \end{aligned}$$

□

Bemerkung:
Nach Konstruktion gilt $|\widetilde{Q}| = 2^{|Q|}$. D.h. der Nichtdeterminismus (also die Wahlmöglichkeit und die ε–Übergänge) kann mit einem gewissen Zusatzaufwand (#Zustände $\widehat{=}$ Speicherplatz) beseitigt werden. Im allgemeinen verringert sich die Abarbeitungszeit: bei einem DEA ist die Abarbeitung zu w gerade $|w|$, bei einem NEA ist sie dagegen ungewiß.

Beispiel :
Sprache aller Wörter, deren vorletztes Symbol 0 ist: $L = \mathbf{(0 \cup 1)^*0(0 \cup 1)}$

Abbildung 2.6 zeigt einen NEA, der L erkennt, Abbildung 2.7 beschreibt den äquivalenten DEA, der durch die Potenzmengenkonstruktion entstanden ist. Für diesen ergibt sich:

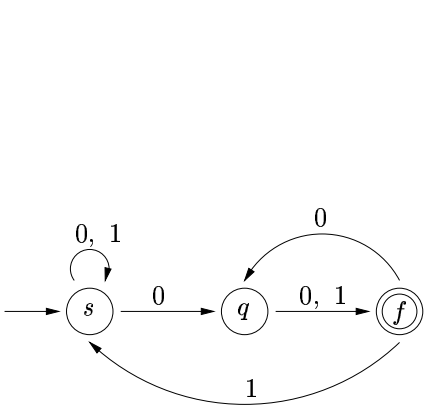

Abbildung 2.6: NEA für L

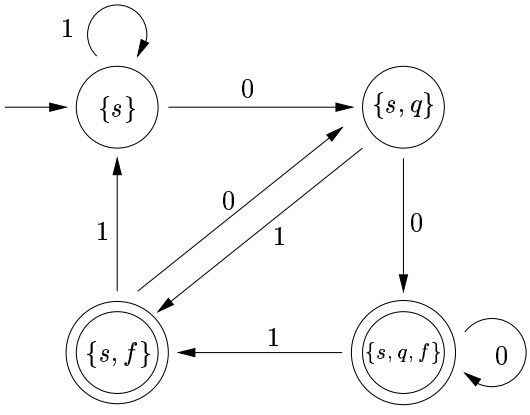

Abbildung 2.7: DEA für L

- Anfangszustand ist $E(s) = \{s\}$
- Zustände sind $\{s\}, \{s,q\}, \{s,f\}, \{s,q,f\}$
- Endzustände sind $\{s,f\}, \{s,q,f\}$
- Alle anderen Zustände aus 2^Q, die nicht vorkommen, werden gestrichen.

■

Wir verwenden nun das Konzept der nichtdeterministischen endlichen Automaten für den noch ausstehenden Beweis von Satz 2.9.

Beweis zu Satz 2.9:
Zu zeigen: Jede reguläre Sprache wird von einem deterministischen endlichen Automaten akzeptiert.

Der Induktionsanfang für $L = \emptyset$ beziehungsweise $L = \boldsymbol{a}$ wurde bereits gezeigt. Wir zeigen hier nur den Induktionsschritt für reguläre Sprachen $L = L_1 \cup L_2$, $L = L_1 \cdot L_2$ und $L = L_1^*$.

Seien also L_1 und L_2 reguläre Sprachen, die von den deterministischen endlichen Automaten $\mathcal{A}_i := (Q_i, \Sigma, \delta_i, s_i, F_i)$ erkannt werden.

- Gesucht ist ein deterministischer endlicher Automat zu $L_1 \cup L_2$.

  Sei $\mathcal{A} := (Q, \Sigma, \delta, s, F)$ ein nichtdeterministischer endlicher Automat mit $Q := Q_1 \cup Q_2 \cup \{s\}$ $(s \notin Q_i)$, $F = F_1 \cup F_2$ und

$$\delta(q,a) := \begin{cases} \{\delta_i(q,a)\} & \text{falls } q \in Q_i,\ a \in \Sigma,\ i \in \{1,2\} \\ \emptyset & \text{falls } q \in Q \backslash \{s\},\ a = \varepsilon \\ \{s_1, s_2\} & \text{falls } q = s,\ a = \varepsilon \\ \emptyset & \text{falls } q = s,\ a \neq \varepsilon \end{cases}$$

  Die Abbildung 2.8 illustriert die Konstruktion. Dann gilt offensichtlich $L(\mathcal{A}) = L$. Zu $\mathcal{A}$ kann ein äquivalenter deterministischer endlicher Automat konstruiert werden.

- Gesucht ist ein deterministischer endlicher Automat zu $L_1 \cdot L_2$.

  Sei $\mathcal{A} := (Q, \Sigma, \delta, s, F)$ ein nichtdeterministischer endlicher Automat mit $Q := Q_1 \cup Q_2$, $s := s_1$, $F := F_2$ und

$$\delta(q,a) := \begin{cases} \{\delta_i(q,a)\} & \text{falls } q \in Q_i,\ a \in \Sigma,\ i \in \{1,2\} \\ \emptyset & \text{falls } q \in Q \backslash F_1,\ a = \varepsilon \\ \{s_2\} & \text{falls } q \in F_1,\ a = \varepsilon \end{cases}$$

  Die Abbildung 2.9 illustriert die Konstruktion. Dann gilt offensichtlich $L(\mathcal{A}) = L$. Zu $\mathcal{A}$ kann ein äquivalenter deterministischer endlicher Automat konstruiert werden.

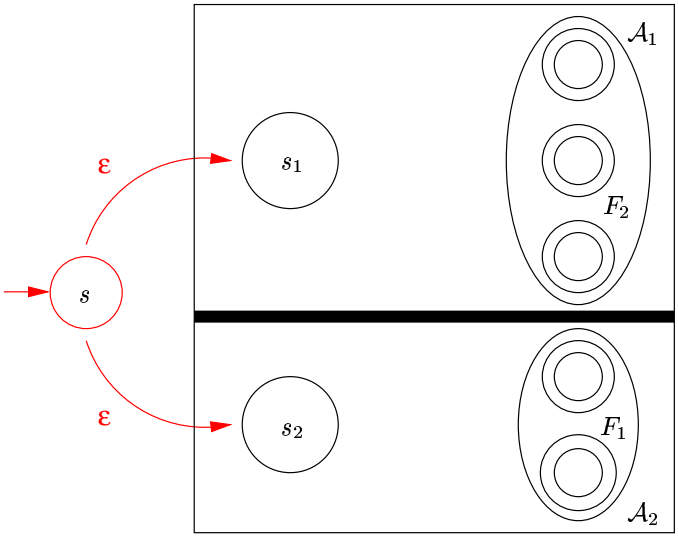

Abbildung 2.8: NEA für $L_1 \cup L_2$

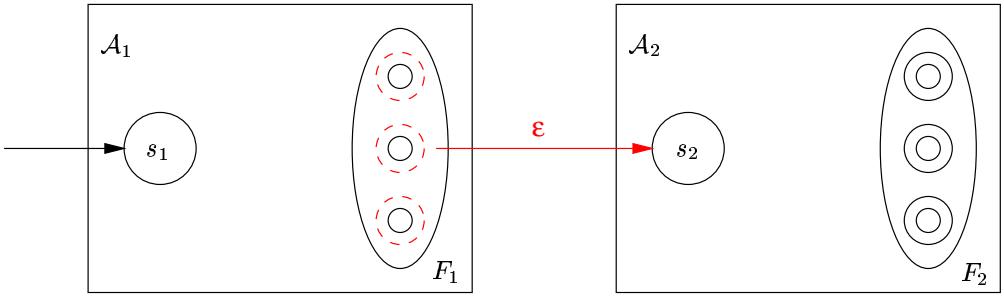

Abbildung 2.9: NEA für $L_1 \cdot L_2$

- Gesucht ist ein deterministischer endlicher Automat zu L_1^*.

  Sei $\mathcal{A} := (Q, \Sigma, \delta, s, F)$ ein nichtdeterministischer endlicher Automat mit $Q := Q_1 \cup \{s, f\}$, wobei s und f neue Zustände sind, $F := \{f\}$ und

$$\delta(q,a) := \begin{cases} \{\delta_1(q,a)\} & \text{falls } q \in Q_1,\ a \in \Sigma \\ \emptyset & \text{falls } q \in Q_1 \backslash F_1,\ a = \varepsilon \\ \{f\} & \text{falls } q \in F_1 \cup \{s\},\ a = \varepsilon \\ \emptyset & \text{falls } q = s,\ a \neq \varepsilon \\ \emptyset & \text{falls } q = f,\ a \neq \varepsilon \\ \{s_1\} & \text{falls } q = f,\ a = \varepsilon \end{cases}$$

  Die Abbildung 2.10 illustriert die Konstruktion. Dann gilt offensichtlich $L(\mathcal{A}) = L$. Zu $\mathcal{A}$ kann ein äquivalenter deterministischer endlicher Automat konstruiert werden.

□

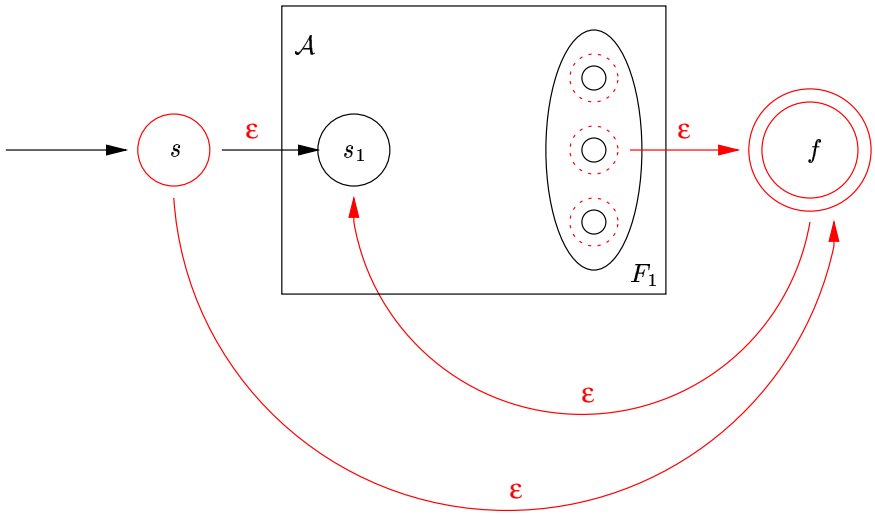

Abbildung 2.10: NEA für L_1^*

Wir haben bei nichtdeterministischen endlichen Automaten explizit ε-Übergänge erlaubt. Diese sind natürlich bei der Konstruktion von Automaten zu Sprachen sehr brauchbar. Kommt man auch ohne ε-Übergänge aus, ohne die Anzahl der Zustände zu vergrößern?

2.13 Satz
Zu jedem nichtdeterministischen endlichen Automat mit ε-Übergängen gibt es einen äquivalenten nichtdeterministischen endlichen Automaten ohne ε-Übergängen, der nicht mehr Zustände hat.

Beweis: Sei $\mathcal{A} := (Q, \Sigma, \delta, s, F)$ ein nichtdeterministischer endlicher Automat mit ε–Übergängen. Wir konstruieren einen äquivalenten nichtdeterministischen endlichen Automaten $\widetilde{\mathcal{A}} := (\widetilde{Q}, \Sigma, \widetilde{\delta}, \widetilde{s}, \widetilde{F})$ ohne ε–Übergänge, der dieselbe Sprache akzeptiert und nicht mehr Zustände hat. Wir wählen:

- $\widetilde{Q} := (Q \backslash F) \cup \widetilde{F}$
- $\widetilde{s} := s$
- Zu $\widetilde{\delta}$: Es soll $\widetilde{\delta}(q, a)$ für $a \in \Sigma$ genau die Zustände enthalten, in die $\mathcal{A}$ von q aus mit einer beliebige Anzahl von ε–Übergängen und dem anschließenden Lesen von a kommen kann, d.h.

$$\widetilde{\delta}(q, a) = \begin{cases} \{q\} & \text{falls } a = \varepsilon \\ \delta(E(q), a) & \text{sonst} \end{cases}$$

- $\widetilde{F} := \{q \mid E(q) \cap F \neq \emptyset\}$

Damit akzeptiert $\widetilde{A}$ dieselbe Sprache wie $\mathcal{A}$, und $|\widetilde{Q}| \leq |Q|$. □

Wir haben mit Satz 2.9 gezeigt, daß es zu jeder regulären Sprache einen (nicht) deterministischen endlichen Automaten gibt, der sie akzeptiert. Es gilt noch mehr, nämlich daß die regulären Sprachen **genau** die Sprachen sind, die durch einen nichtdeterministischen beziehungsweise deterministischen endlichen Automaten akzeptiert werden. Es gilt also auch die Umkehrung von Satz 2.9.

2.14 Satz
Jede Sprache, die von einem endlichen Automaten erkannt wird, ist regulär.

Beweis: Sei ein deterministischer endlicher Automat $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ gegeben, der die Sprache L akzeptiert. Es ist zu zeigen, daß L regulär ist. Sei $Q = \{q_1, \ldots, q_n\}$. Es gilt:

$$L = \{w \in \Sigma^* \mid \mathcal{A} \text{ endet nach Abarbeitung von } w \text{ in einem Zustand aus } F\}$$

Die Abarbeitung eines Wortes $w = a_1 \ldots a_k$ bewirkt das Durchlaufen einer Folge von Zuständen $s, q_1, \ldots, q_k$, wobei nicht notwendig $q_i \neq q_j$ für $i \neq j$ gilt. Wir suchen die Wörter, die eine Folge bewirken, deren letzter Zustand in F ist. Betrachte dazu für jeden Zustand $f \in F$ getrennt die Wörter, deren Abarbeitung in f endet. Zu $f \in F$ definiere:

$$\begin{aligned} L_f &:= \{w \in \Sigma^* \mid \mathcal{A} \text{ endet nach Abarbeitung von } w \text{ in } f\} \\ &= \{w \in \Sigma^* \mid w \text{ überführt } s \text{ in } f \text{ in } \mathcal{A}\} \end{aligned}$$

Damit ist $L = \bigcup_{f \in F} L_f$. Wenn wir zeigen können, daß für alle $f \in F$

L_f regulär ist, so ist auch L regulär.

Wir definieren zu $q_r, q_t \in Q$: $L_{q_r,q_t} := \{w \in \Sigma^* \mid w \text{ überführt } q_r \text{ in } q_t\}$. Insbesondere gilt also: $L_f = L_{s,f}$. Unterteile L_{q_r,q_t}:

$$L_{q_r,i,q_t} := \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} \text{Abarbeitung von } w \text{ aus } q_r \text{ nach } q_t \text{ hat nur} \\ \text{Zwischenzustände } \{q_1, \ldots, q_i\} \end{array} \right\}$$

(also w bewirkt: $q_r \to \underbrace{\ldots\ldots\ldots}_{\in\{q_1,\ldots,q_i\}} \to q_t$.)

Damit gilt $L_{q_r,q_t} = L_{q_r,n,q_t}$.

Wir zeigen, daß L_{q_r,i,q_t} für $q_r, q_t \in Q$ und $1 \leq i \leq n$ regulär sind:

- Zunächst betrachten wir direkte Überführungen, also $i = 0$:

$$L_{q_r,0,q_t} := \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} \text{Abarbeitung von } w \text{ führt von } q_r \text{ nach } q_t \\ \text{ohne Zwischenzustand} \end{array} \right\}$$

  Falls $r = t$ und somit $q_r = q_t$ ist, ist $L_{q_r,0,q_t} = \{a \in \Sigma \mid \delta(q_t, a) = q_t\} \cup \{\varepsilon\}$. Andernfalls betrachten wir alle w mit $q_r \overset{w}{\to} q_t$, ohne Zwischenzustände, also $L_{q_r,0,q_t} = \{a \in \Sigma \mid \delta(q_r, a) = q_t\}$

  Diese Sprachen sind jeweils regulär.

- Betrachte nun $i = 1$:

$$L_{q_r,1,q_t} := \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} w \text{ überführt } q_r \text{ in } q_t \text{ entweder direkt oder} \\ \text{unter Benutzung nur von } q_1 \end{array} \right\}$$

  Es gilt dann:

$$L_{q_r,1,q_t} = L_{q_r,0,q_t} \cup \left(L_{q_r,0,q_1} \cdot L^*_{q_1,0,q_1} \cdot L_{q_1,0,q_t} \right)$$

  Also ist $L_{q_r,1,q_t}$ auch wieder regulär.

- Es gilt allgemein:

$$L_{q_r,i+1,q_t} = L_{q_r,i,q_t} \cup \left(L_{q_r,i,q_{i+1}} \left(L_{q_{i+1},i,q_{i+1}} \right)^* L_{q_{i+1},i,q_t} \right)$$

Da für $L_{\cdot,i+1,\cdot}$ nur die Sprachen $L_{\cdot,i,\cdot}$ und $\cup, \cdot, *$ verwendet werden, ist gezeigt (per Induktion), daß $L_{\cdot,i+1,\cdot}$ regulär ist für beliebiges i $(1 \leq i+1 \leq n)$ und alle Zustandspaare aus Q^2. Damit ist gezeigt, daß insbesondere $L_f = L_{s,n,f}$ regulär ist für jedes $f \in F$. □

Beispiel :
Wir betrachten $(Q, \Sigma, \delta, s, F)$ mit $Q := \{q_1 := s, q_2 := q\}, \Sigma := \{0,1\}, F := \{s\}$ und δ wie in Abbildung 2.11.

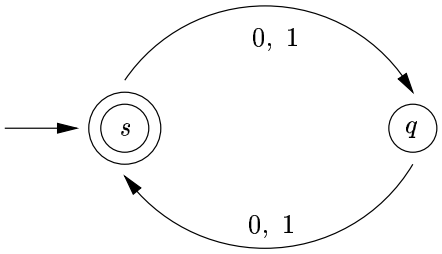

Abbildung 2.11: Beispiel

Es ist die Sprache L gesucht, die durch diesen endlichen Automaten akzeptiert wird. Es gilt $L = L_{q_1,2,q_1}$
Dann ist $L_{q_i,0,q_i} = \varepsilon$ und $L_{q_i,0,q_j} = (0 \cup 1)$ für $i, j \in \{1,2\}, i \neq j$.

$$\begin{aligned} L_{q_1,1,q_1} &= L_{q_1,0,q_1} \cup L_{q_1,0,q_1} \left(L_{q_1,0,q_1} \right)^* L_{q_1,0,q_1} = \varepsilon \\ L_{q_1,1,q_2} &= (0 \cup 1) \cup \varepsilon\varepsilon^*(0 \cup 1) = 0 \cup 1 \\ L_{q_2,1,q_1} &= (0 \cup 1) \cup (0 \cup 1)\varepsilon\varepsilon^* = 0 \cup 1 \\ L_{q_2,1,q_2} &= \varepsilon \cup (0 \cup 1)\varepsilon^*(0 \cup 1) = \varepsilon \cup (0 \cup 1)(0 \cup 1) \end{aligned}$$

$$L = L_{q_1,2,q_1} = \varepsilon \cup (0 \cup 1) \left((0 \cup 1)(0 \cup 1) \right)^* (0 \cup 1) = \left((0 \cup 1)(0 \cup 1) \right)^* \qquad \blacksquare$$

Wir haben gezeigt, daß die von endlichen Automaten akzeptierten Sprachen genau die regulären Sprachen sind. (Satz 2.9, Satz 2.14). Dies wird auch als der **Satz von Kleene** bezeichnet.

Frage: Was können endliche Automaten nicht? Sie können keine nicht-regulären Sprachen erkennen. Aber wie zeigt man die Nichtregularität einer Sprache?

Beispiel :
Die Sprache L der korrekten Klammerausdrücke über $\Sigma = \{(,)\}$.

Zum Beispiel:

$$\Big(()()\Big), \Big(()()\Big(()\Big)\Big) \in L \qquad (((())),((()))()() \notin L$$

Die Klammerung ist genau dann korrekt, wenn w gleich viele (und) enthält, und wenn man w von links nach rechts liest, so gibt es nie mehr) als (bis dahin. (D.h. es werden keine Klammern geschlossen, die nicht vorher geöffnet worden waren.)

Ein Automat, der L erkennen kann, muß in der Lage sein, sich für ein beliebiges Wort $w \in L$ die Anzahl von (gegenüber) zu merken, also die Differenz zwischen #(und #). Diese kann aber beliebig groß werden, und der Automat müßte über unendliche viele Zustände verfügen. Die Sprache der Klammerausdrücke ist also zwar simpel, aber nicht regulär. ■

2.15 Satz (Pumping–Lemma für reguläre Sprachen)
Sei L eine reguläre Sprache. Dann existiert eine Zahl $n \in \mathbb{N}$, so daß sich jedes Wort $w \in L$ mit $|w| > n$ darstellen läßt als:

$$w = uvx \text{ mit } |uv| \leq n\,,\ v \neq \varepsilon,$$

so daß auch $uv^i x \in L$ ist für alle $i \in \mathbb{N}_0$.

Beweis: Sei L eine reguläre Sprache. Dann existiert ein endlicher Automat, der L akzeptiert. Sei Q dessen Zustandsmenge und $n := |Q|$. Sei $w \in L$ mit $|w| > n$, etwa $w = a_1 \dots a_n \dots a_m$ mit $m > n$. Bei der Abarbeitung von w werden dann die Zustände $q_0, \dots, q_m$ durchlaufen mit $q_m \in F$. Dann gibt es i, j mit $0 \leq i, j \leq n$ und $i \neq j$, so daß $q_i = q_j$. Œ gelte $i < j$.

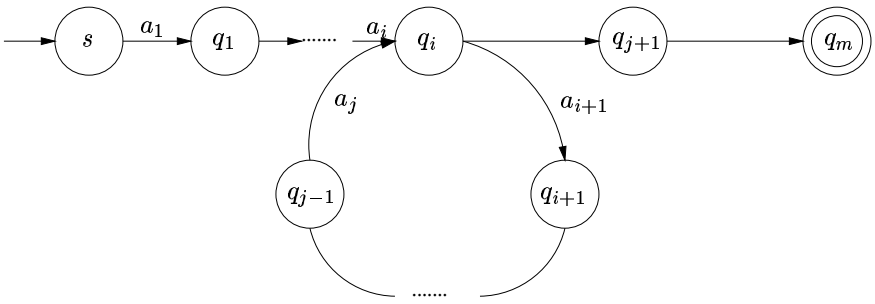

Abbildung 2.12: Abarbeitungs von w im DEA

Dann kann der „Zykel“ $q_i, q_{i+1}, \ldots, q_j = q_i$ auch gar nicht oder beliebig oft bei der Abarbeitung eines Wortes aus L durchlaufen werden so daß der Zustand $q_m \in F$ erreicht wird. Also gibt es eine Zerlegung von w in

$$w = \underbrace{(a_1 \ldots a_i)}_{u} \cdot \underbrace{(a_{i+1} \ldots a_j)}_{v} \cdot \underbrace{(a_{j+1} \ldots a_m)}_{x}$$

mit $|uv| \leq n$ und $v \neq \varepsilon$, so daß auch $uv^i x \in L$ für alle $i \in \mathbb{N}$. □

Bemerkung:
Satz 2.15 liefert nur eine notwendige, aber nicht hinreichende Bedingung für die Regularität einer Sprache.

Beispiele:

(1) Sei $\Sigma = \{0, 1\}$ und

$L = \{w \in \Sigma^* \mid w \text{ enthält } 10 \text{ nicht als Teilwort } \} = 0^*1^*$.

Betrachte $n = 1$ und $w = uvx$ mit $u = \varepsilon$. v entspricht also dem ersten Buchstaben von w. Dann kann $uv^i x$ 10 auch nicht als Teilwort besitzen.

(2) Sei $\Sigma = \{0, 1\}$ und $L = \{0^i 1^i \mid i \geq 0\}$. Wir zeigen, daß L nicht regulär ist.

Für ein n wähle $w = 0^n 1^n$, dann ist $|w| > n$. Für jede Darstellung $w = uvx$ mit $|uv| \leq n$ und $v \neq \varepsilon$ ist aber $uv^0 x = 0^l 1^n$ $(l < n)$ nicht in L.

(3) Bei der Sprache der korrekten Klammerausdrücke wählt man entsprechende zu Beispiel 2 $(^n)^n$.

(4) Sei $\Sigma = \{0\}$ und $L = \left\{0^{k^2} \mid k \in \mathbb{N}\right\}$. L ist die Sprache aller Wörter über Σ mit quadratischer Länge. L ist nicht regulär.

Denn sei $n > 1$ beliebig aus $\mathbb{N}$ und $w = 0^{n^2} \in L$, also $|w| > n$. Weiter sei $w = uvx$ mit $1 \leq |v| \leq n$. Bei jeder Wahl von u, v und x gilt:

$uv^2 x = 0^{n^2+|v|} \notin L$, denn es gilt:

$$n^2 < n^2 + |v| \leq n^2 + n < (n+1)^2.$$

(5) Sei $\Sigma = \{0, 1\}$ und

$$L = \left\{ w \in \Sigma^* \;\middle|\; w = 1^k \ (k > 0) \text{ oder } w = 0^j 1^{k^2} \ (j \geq 1,\ k \geq 0) \right\}.$$

Dann erfüllt L den Satz 2.15: Sei $n = 1$ und $w \in L$ mit $|w| > 1$. w habe eine Darstellung $w = uvx$ mit $|uv| \leq n$ und $|v| \neq \varepsilon$. Setze $u = \varepsilon$ und $|v| = 1$ das erste Symbol von w.

- Falls $w = 1^k$, so ist auch $uv^i x$ vom Typ $1^\ell \in L$.
- Falls $w = 0^j 1^{k^2}$, so ist auch $uv^0 x \in L$ (für $j = 1$ ist $uv^0 x = x = 1^{k^2}$). Für $i \geq 1$ gilt $uv^i x = 0^{j+i} 1^{k^2} \in L$.

Trotzdem legt Beispiel 4 nahe, daß L nicht regulär ist. Dies läßt sich mit folgendem verallgemeinertem Pumping Lemma zeigen.

■

2.16 Satz (Verallgemeinertes Pumping Lemma für reguläre Sprachen)
Sei L eine reguläre Sprache. Dann gibt es eine Zahl $n \in \mathbb{N}$, so daß für jedes Wort $w \in L$ mit $|w| \geq n$ für jede Darstellung $w = tyx$ mit $|y| = n$ gilt:

das Teilwort y läßt sich folgendermaßen darstellen:

$y = uvz$ mit $v \neq \varepsilon$ und $tuv^i zx \in L$ für alle $i \in \mathbb{N}_0$.

Beweis: Sei L eine reguläre Sprache und $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ der deterministische endliche Automat, der L erkennt. Setze $n := |Q|+1$. Sei $tyx \in L$ mit $|y| = n$. Sei $q_0, \ldots, q_n$ die Folge der Zustände, die bei der Abarbeitung von y durchlaufen werden. Da diese Folge mindestens einen Zykel enthält, kann y so aufgespalten werden, daß $y = uvz$ gilt, so daß v der Buchstabenfolge entspricht, die beim Durchlaufen des Zykels abgearbeitet wird. Insbesondere ist v nicht leer. Dieser Zykel kann dann beliebig oft durchlaufen werden, ohne daß sich die Abarbeitung ändert. D.h. auch $tuv^i zx$ ist ein gültiges Wort, und der Automat erkennt es. □

Bemerkung:
Satz 2.16 ist eine Verallgemeinerung von Lemma 2.15, jedoch *keine* äquivalente Bedingung für Regularität. Allerdings kann mit Satz 2.16 für eine große Klasse von Sprachen Nicht-Regularität bewiesen werden.

Beispiel :
Fortsetzung zu Beispiel 5: Zu n sei $w = 0^j 1^{k^2} = tyx$ mit $y = 1^n$. Dann folgt wie im Beispiel 4 die Nicht-Regularität. ■

2.3 Minimierung von Automaten, Äquivalenzklassenautomat

Im Beweis zu Satz 2.12 in der „Potenzmengenkonstruktion“ haben wir zu einem nichtdeterministischen endlichen Automaten einen äquivalenten deterministischen endlichen Automaten konstruiert, der allerdings wesentlich mehr Zustände haben kann. Die Anzahl der Zustände des deterministischen endlichen Automaten kann exponentiell in der Anzahl der Zustände des nichtdeterministischen endlichen Automaten sein. Am Beispiel sieht man allerdings, daß oft viele überflüssige Zustände entstehen.

Frage: Kann man konstruktiv die Anzahl der Zustände eines deterministischen endlichen Automatens erheblich verringern?

2.17 Definition
Zustände eines (deterministischen) endlichen Automatens, die vom Anfangszustand aus nicht erreichbar sind, heißen ***überflüssig****.*

In Beispiel in Abbildung 2.13 sind die Zustände s, q_1, q_2, f erreichbar, q_3 und q_4 sind überflüssig.

Daraus ergibt sich der erste Schritt bei der Zustandsminimierung, nämlich das Streichen aller überflüssigen Zustände. Sind die überflüssigen Zustände leicht zu finden?

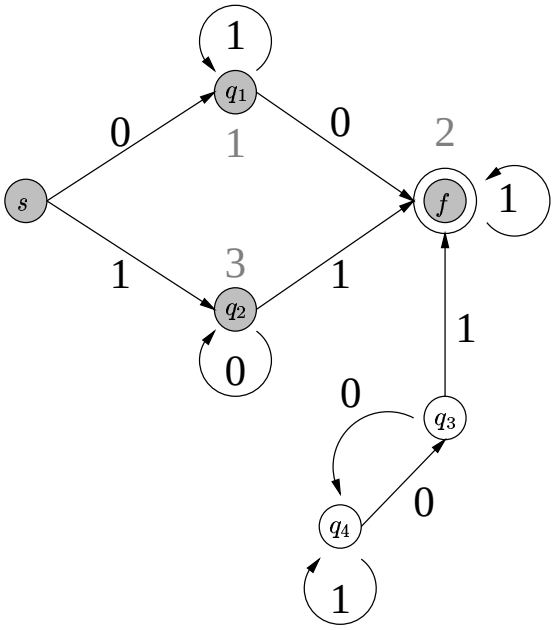

Abbildung 2.13: Beispiel

Wir können endliche Automaten als gerichtete Graphen auffassen. Die überflüssigen Zustände entsprechen dann den Knoten, zu denen es vom Anfangsknoten aus keinen gerichteten Weg gibt. Eine Tiefensuche (**D**epth–**F**irst **S**earch, DFS) in dem Graphen liefert damit alle nicht überflüssigen Zustände.

2.18 Satz
Die Menge aller überflüssigen Zustände eines (deterministischen) endlichen Automaten) kann in der Zeit $\mathcal{O}(|Q| \cdot |\Sigma|)$ berechnet werden.

Beweis: Wende DFS ab dem Startzustand an. Dies erfordert einen Aufwand proportional zu der Anzahl der Kanten in dem Graphen. □

Ein deterministischer endlicher Automat ohne überflüssige Zustände muß jedoch noch nicht minimal sein.

2.19 Beispiel
Sei $L = \{w \in \{0,1\}^* \mid (|w|_0 \mod 2) = (|w|_1 \mod 2) = 0\}$, wobei $|w|_a$ die Anzahl der Vorkommen des Zeichens $a \in \Sigma$ in w bezeichnet. L ist also die Sprache, in der sowohl eine gerade Anzahl von Nullen als auch Einsen vorkommt.

Betrachte den deterministischen endlichen Automaten aus Abbildung 2.14 mit 16 Zuständen q_{ij}, $0 \leq i, j \leq 3$. Sei z_0 die Anzahl der gelesenen Nullen und z_1 die Anzahl der gelesenen Einsen zu einem Zeitpunkt t. Dann wird der Zustand q_{ij} zum Zeitpunkt t genau dann erreicht, wenn gilt:

$$i \equiv z_0 \mod 4 \qquad \text{und} \qquad j \equiv z_1 \mod 4.$$

Der Automat in Abbildung 2.15 akzeptiert ebenfalls L, besitzt jedoch nur vier Zustände. ■

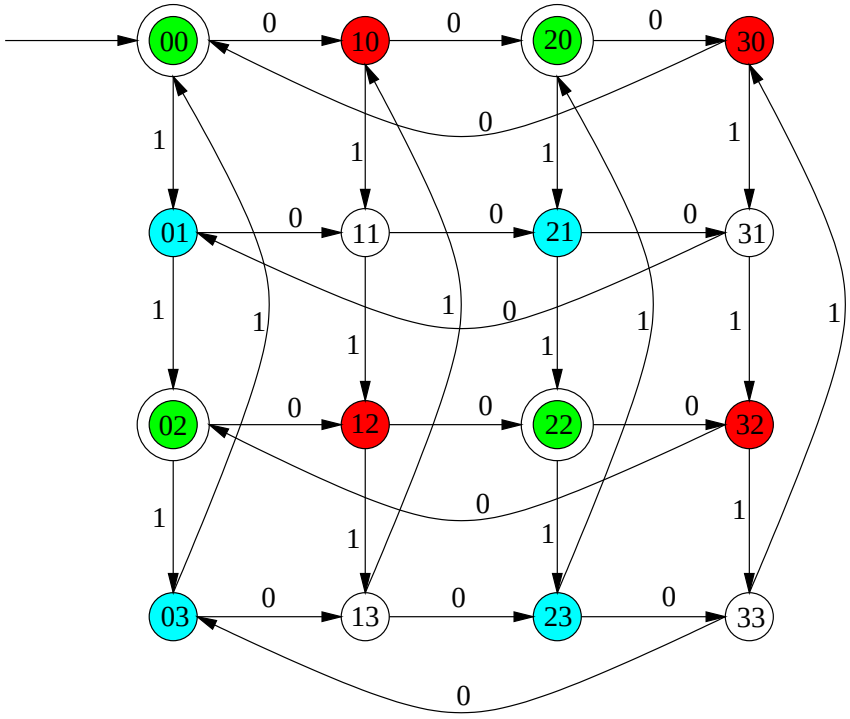

Abbildung 2.14: Gitterautomat

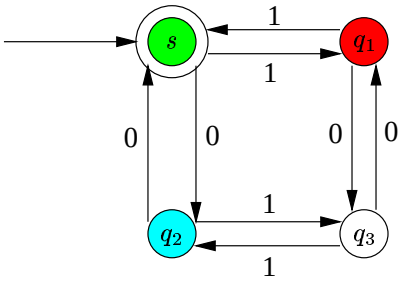

Abbildung 2.15: Äquivalenter Automat mit weniger Zuständen

Zwei Zustände haben dasselbe „Akzeptanzverhalten“, wenn es für das Erreichen eines Endzustandes durch Abarbeiten eines Wortes w unerheblich ist, aus welchem der beiden Zustände wir starten. Man kann die Anzahl der Zustände nun reduzieren, indem man Zustände, deren Akzeptanzverhalten gleich ist, „zusammenlegt“. Im obigen Beispiel ist dies durch Färbung der Zustände mit gleichem Verhalten durch gleiche Farben veranschaulicht.

2.20 Definition
Zwei Zustände p und q eines deterministischen endlichen Automaten heißen ***äquivalent*** *($p \equiv q$), wenn für alle Wörter $w \in \Sigma^*$ gilt:*

$$\delta(p, w) \in F \quad \Longleftrightarrow \quad \delta(q, w) \in F$$

Offensichtlich ist $\equiv$ eine Äquivalenzrelation. Mit $[p]$ bezeichnen wir die Äquivalenzklasse der zu p äquivalenten Zustände.

2.21 Definition
Zu einem deterministischen endlichen Automaten $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ definieren wir den Äquivalenzklassenautomaten $\mathcal{A}^{\equiv} = (Q^{\equiv}, \Sigma^{\equiv}, \delta^{\equiv}, s^{\equiv}, F^{\equiv})$ durch:

- $Q^{\equiv} := \{[q] \mid q \in Q\}$
- $\Sigma^{\equiv} := \Sigma$
- $\delta^{\equiv}([q], a) := [\delta(q, a)]$
- $s^{\equiv} := [s]$
- $F^{\equiv} := \{[f] \mid f \in F\}$

2.22 Satz
Der Äquivalenzklassenautomat $\mathcal{A}^{\equiv}$ zu einem deterministischen endlichen Automaten $\mathcal{A}$ ist wohldefiniert.

Beweis: Wir müssen zeigen, daß $F^{\equiv}$ und $\delta^{\equiv}$ wohldefiniert sind, der Rest ist klar. Dazu zeigen wir:

- ein Endzustand kann nur zu einem Endzustand äquivalent sein,
- δ führt äquivalente Zustände beim Lesen desselben Symbols wieder in äquivalente Zustände über.

Für ε gilt: $\delta(p, \varepsilon) \in F \Leftrightarrow \delta(q, \varepsilon) \in F$. Dies gilt genau für $p, q \in F$. D.h. falls $p \equiv q$ dann gilt $p, q \in F$ oder $p, q \notin F$. Also ist $F^{\equiv}$ wohldefiniert.

Sei $p \equiv q$. Dann gilt für alle $w \in \Sigma^*$ $\delta(q, w) \in F \Leftrightarrow \delta(p, w) \in F$. Somit gilt auch für alle $a \in \Sigma$:

$$\delta(q, aw) \in F \Leftrightarrow \delta(p, aw) \in F.$$

Damit folgt $\delta(q, a) \equiv \delta(p, a)$, also ist auch $\delta^{\equiv}$ wohldefiniert. □

2.23 Satz
Der Äquivalenzklassenautomat $\mathcal{A}^{\equiv}$ zu $\mathcal{A}$ akzeptiert dieselbe Sprache wie $\mathcal{A}$.

Beweis: Sei $w \in \Sigma^*$, $q_0 := s, q_1, \ldots, q_n$ die Folge der Zustände, die von $\mathcal{A}$ bei der Abarbeitung von w durchlaufen werden. Bei Abarbeitung von w in $\mathcal{A}^{\equiv}$ werden dann die Zustände $[q_0], [q_1], \ldots, [q_n]$ durchlaufen. $\mathcal{A}$ akzeptiert w genau dann, wenn $q_n \in F$ gilt. $\mathcal{A}^{\equiv}$ akzeptiert w genau dann, wenn $[q_n] \in F^{\equiv}$ gilt. Nach Definition von $\mathcal{A}^{\equiv}$ ist $q_n \in F$ genau dann, wenn $[q_n] \in F^{\equiv}$ gilt. □

Frage: Wie konstruiert man $\mathcal{A}^{\equiv}$ zu $\mathcal{A}$? D.h. wie berechnet man alle Äquivalenzklassen zu den Zuständen von $\mathcal{A}$?

Zu beweisen, daß zwei Zustände p und q äquivalent sind, erscheint aufwendig, da nach Definition nachgewiesen werden müßte, daß für alle $w \in \Sigma^*$ gilt:

$$\delta(p, w) \in F \Longleftrightarrow \delta(q, w) \in F.$$

Es gibt jedoch unendlich viele $w \in \Sigma^*$.

Es ist einfacher für p und q zu zeigen, daß p nicht äquivalent zu q ist. Da benötigen wir *nur ein* Wort $w \in \Sigma^*$ mit $\delta(p, w) \in F$ aber $\delta(q, w) \notin F$, beziehungsweise $\delta(p, w) \notin F$ aber $\delta(q, w) \in F$.

Notation
Wir bezeichnen ein solches Wort w als **Zeuge** für die Nichtäquivalenz von p und q und sagen w trennt p und q.

Frage: Können Zeugen beliebig lang werden?

Sei $w = aw'$ ein *kürzester* Zeuge für $p \not\equiv q$. Dann ist w' Zeuge für $\delta(p, a) \not\equiv \delta(q, a)$. Wenn es für $p' \not\equiv q'$ einen kürzeren Zeugen w'' gäbe, so wäre aw'' ein kürzerer Zeuge für $p \not\equiv q$ als w. Dies ist aber ein Widerspruch dazu, daß w ein kürzester Zeuge ist. D.h. wenn wir alle Wörter aus Σ^* in der Reihenfolge ihrer Länge darauf testen, ob sie Zeuge sind, und für eine bestimmte Länge kein Zeuge mehr für eine Nichtäquivalenz auftritt, so kann das Verfahren abgebrochen werden.

Vorgehensweise

Betrachte alle Zustandspaare und zunächst ε, dann alle Elemente aus Σ, dann alle Wörter der Länge 2 aus Σ^*, und so weiter.

Zunächst betrachte alle Zustände als eine Klasse. Dann trennt ε die Zustände aus F von denen aus $Q \backslash F$. Danach testen wir nur noch Paare von Zuständen aus F beziehungsweise $Q \backslash F$. Durch mindestens ein Wort der Länge 1 wird entweder F oder $Q \backslash F$ weiter getrennt, oder das Verfahren ist beendet. Dies wird iterativ so weitergeführt mit Wörtern wachsender Länge.

Beispiel :
Betrachte den Gitterautomat aus Beispiel 2.19, Abbildung 2.14

- ε trennt $\underbrace{\{00, 02, 20, 22\}}_{\text{grün}}$ von $\{01, 03, 10, 11, 12, 13, 21, 23, 30, 31, 32, 33\}$
- 0 trennt $\underbrace{\{10, 30, 12, 32\}}_{\text{rot}}$ von $\{01, 03, 11, 13, 21, 23, 31, 33\}$

- 1 trennt $\underbrace{\{01, 03, 21, 23\}}_{\text{blau}}$ von $\underbrace{\{11, 13, 31, 33\}}_{\text{weiß}}$
- die Wörter $00, 01, 10, 11$ trennen keine Zustandspaare mehr.

D.h. die Äquivalenzklassen der Zustände sind: $s = [00], q_1 = [01], q_2 = [10]$ und $q_3 = [11]$. ■

Frage: Ist der Äquivalenzklassenautomat zu einem deterministischen endlichen Automaten schon der äquivalente Automat mit der minimalen Anzahl von Zuständen?

Um zu beweisen, daß $\mathcal{A}^{\equiv}$ zu einem deterministischen endlichen Automaten $\mathcal{A}$ minimal ist, konstruieren wir einen minimalen Automaten zu der zugehörigen Sprache L des DEAs. Dieser Automat ist der „Automat der Nerode-Relation". Anschließend zeigen wir, daß $\mathcal{A}^{\equiv}$ höchstens soviele Zustände hat wie jener.

2.24 Definition
*Eine Äquivalenzrelation R über Σ^* heißt **rechtsinvariant**, wenn:*

für alle $x, y \in \Sigma^*$ gilt: falls $x\ R\ y$ so gilt auch $xz\ R\ yz$ für alle $z \in \Sigma^*$.

*Den **Index** von R bezeichnen wir mit **ind(R)**; er ist die Anzahl der Äquivalenzklassen von Σ^* bezüglich R.*

2.25 Definition
*Für eine Sprache $L \subseteq \Sigma^*$ ist die **Nerode–Relation** R_L definiert für $x, y \in \Sigma^*$ durch:*

$$x\ R_L\ y \text{ genau dann wenn } (xz \in L \Leftrightarrow yz \in L) \text{ für alle } z \in \Sigma^* \text{ gilt.}$$

Bemerkung:
Die Nerode–Relation R_L zu einer Sprache $L \subseteq \Sigma^*$ ist eine rechtsinvariante Äquivalenzrelation, denn R_L ist offensichtlich Äquivalenzrelation. Es gilt:

$$\begin{aligned} x\ R_L\ y &\Rightarrow (xw \in L \Leftrightarrow yw \in L) \text{ für alle } w \in \Sigma^* \\ &\Rightarrow (xzw \in L \Leftrightarrow yzw \in L) \text{ für alle } w,\ z \in \Sigma^* \\ &\Rightarrow (xz\ R_L\ yz) \text{ für alle } z \in \Sigma^*. \end{aligned}$$

2.26 Satz (von Nerode)
Die folgenden Aussagen sind äquivalent:

1. *$L \subseteq \Sigma^*$ wird von einem deterministischen endlichen Automaten erkannt bzw. akzeptiert.*
2. *L ist die Vereinigung von (einigen) Äquivalenzklassen einer rechtsinvarianten Äquivalenzrelation mit endlichem Index.*
3. *Die Nerode–Relation hat endlichen Index.*

Beweis:

$1 \Rightarrow 2$: Sei $\mathcal{A} := (Q, \Sigma, \delta, s, F)$ der deterministische endliche Automat, der L akzeptiert, und $R_{\mathcal{A}}$ wie folgt definiert:

$$\forall\, x, y \in \Sigma^* : x\ R_{\mathcal{A}}\ y \iff \delta(s, x) = \delta(s, y).$$

$R_{\mathcal{A}}$ ist eine rechtsinvariante Äquivalenzrelation. Der Index von $R_{\mathcal{A}}$ ist gerade die Anzahl der nicht überflüssigen Zustände von $\mathcal{A}$, also endlich. Dann ist L natürlich die Vereinigung der Äquivalenzklassen von $R_{\mathcal{A}}$, die zu den Endzuständen von $\mathcal{A}$ gehören.

$2 \Rightarrow 3$: Sei R die nach Voraussetzung existierende rechtsinvariante Äquivalenzrelation mit endlichem Index. Wir zeigen, daß die Nerode–Relation R_L eine Vergröberung von R ist, d.h. $x\ R\ y$ impliziert $x\ R_L\ y$. Dann ist natürlich $\mathrm{ind}(R_L) \leq \mathrm{ind}(R) < \infty$.

Sei also $x\ R\ y$. Da R rechtsinvariant ist, gilt für alle $z \in \Sigma^*$: $xz\ R\ yz$. Da nach Voraussetzung jede Äquivalenzklasse von R entweder ganz oder gar nicht zu L gehört, ist $xz, yz \in L$ oder $xz, yz \notin L$. Damit folgt $x\ R_L\ y$.

$3 \Rightarrow 1$: Wir konstruieren zu R_L einen deterministischen endlichen Automaten, der L akzeptiert. Sei $\mathcal{A} := (Q, \Sigma, \delta, s, F)$ mit:

- $Q := \{[x]_{R_L} \mid x \in \Sigma^*\}$, Menge aller Äquivalenzklassen bezüglich R_L. Es ist also $|Q| = \mathrm{ind}(R_L) < \infty$.
- $s := [\varepsilon]_{R_L}$,
- $F := \{[w]_{R_L} \mid w \in L\}$ (wohldefiniert)
- $\delta([x]_{R_L}, a) := [xa]_{R_L}$
  δ ist wohldefiniert, denn falls $[w]_{R_L} = [w']_{R_L}$ dann gilt $w\ R_L\ w'$ und wegen Rechtsinvarianz von R_L auch $wa\ R_L\ w'a$.
  Also ist $[wa]_{R_L} = [w'a]_{R_L}$.

Es bleibt zu zeigen, daß $\mathcal{A}$ genau L akzeptiert. Nach Konstruktion ist $\delta(s, w) = \delta([\varepsilon], w) = [\varepsilon w]_{R_L} = [w]_{R_L}$. Also wird w von $\mathcal{A}$ akzeptiert genau dann, wenn $[w] \in F$ gilt, d.h. wenn $w \in L$.

□

2.27 Korollar
*Der im dritten Beweisteil zu Satz 2.26 konstruierte Automat $\mathcal{A}$ zu R_L — der **Automat der Nerode–Relation** — ist minimal.*

Beweis: Sei $\mathcal{A}' := (Q', \Sigma, \delta', s', F')$ ein deterministischer endlicher Automat, der L akzeptiert. Aus $1 \Rightarrow 2$ folgt, daß eine rechtsinvariante Äquivalenzrelation $R_{\mathcal{A}'}$ mit $\mathrm{ind}(R_{\mathcal{A}'}) \leq |Q'|$ existiert. Wegen $2 \Rightarrow 3$ gilt: $\mathrm{ind}(R_L) \leq \mathrm{ind}(R_{\mathcal{A}'})$. Mit $3 \Rightarrow 1$ folgt

$$|Q| = \mathrm{ind}(R_L) \leq \mathrm{ind}(R_{\mathcal{A}'}) \leq |Q'|,$$

für den Nerode–Automat $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ (siehe oben). □

2.28 Satz
Der Äquivalenzklassenautomat $A^{\equiv}$ zu einem deterministischen endlichen Automaten $\mathcal{A}$ ohne überflüssige Zustände ist minimal.

Beweis: $A^{\equiv}$ hat natürlich auch keine überflüssigen Zustände. Nach obigem Korollar genügt es zu zeigen, daß $|Q^{\equiv}| = \mathrm{ind}(\mathrm{R}_L)$, wobei L die vom Automaten $\mathcal{A}$ bzw. $\mathcal{A}^{\equiv}$ akzeptierte Sprache ist.

Es bleibt zu zeigen, daß für alle $x, y \in \Sigma^*$ gilt: $x \; R_L \; y \Rightarrow \delta(s,x) \equiv \delta(s,y)$

$$
\begin{aligned}
x \; R_L \; y &\Rightarrow \forall\, z \in \Sigma^* \colon \; (xz \in L \Leftrightarrow yz \in L) \\
&\Rightarrow \forall\, z \in \Sigma^* \colon \; (\delta(s,xz) \in F \Leftrightarrow \delta(s,yz) \in F) \\
&\Rightarrow \forall\, z \in \Sigma^* \colon \; (\delta(\delta(s,x),z) \in F \Leftrightarrow \delta(\delta(s,y),z) \in F) \\
&\Rightarrow \delta(s,x) \equiv \delta(s,y)
\end{aligned}
$$

□

Kapitel 3

Turing-Maschine, Berechenbarkeit

Wie wir im vorigen Kapitel gesehen haben, sind endliche Automaten als Berechnungsmodell nicht mächtig genug.

Frage: Gibt es ein mächtigeres, realistisches Rechnermodell, das als Grundlage für „allgemeine“ theoretische Aussagen über „Berechenbarkeit“ beziehungsweise „Entscheidbarkeit“ und die „Komplexität“ geeignet ist?

Wir werden einerseits als „realistisches“ Rechnermodell die **Registermaschine** (**R**andom**A**ccess**M**aschine) einführen. Andererseits wird uns die **Turing-Maschine** (TM) als ein Rechnermodell dienen, das sich für allgemeine Aussagen hervorragend eignet.

Zwar scheint die Turing-Maschine nicht besonders „realistisch“ zu sein, d.h. sie entspricht nicht unserer Vorstellung eines wirklichen Rechners; man kann allerdings zeigen (hier nicht), daß die Turing-Maschine „gleichwertig“ zur Registermaschine ist, die wiederum einem wirklichen Rechner modelliert.

Hauptfrage in diesem Kapitel: Welche Probleme sind berechenbar?

3.1 Die Registermaschine

Die RAM besteht aus einem **Befehlszähler**, einem **Akkumulator**, **Registern** und einem **Programm**. Die Inhalte von Befehlszähler, Akkumulator und Registern sind natürliche Zahlen. Die Register bilden den (unendlichen) Speicher der Registermaschine und haben alle jeweils eine eindeutige Adresse (siehe Abbildung 3.1).

Ein Programm besteht aus einer Folge von Befehlen, wobei die Programmzeilen durchnumeriert sind. Der Befehlszähler b startet bei Eins und enthält jeweils die Nummer des nächsten auszuführenden Befehls. In den ersten Registern des Speichers steht zu Beginn der Berechnung die Eingabe. In den übrigen Registern und im Akkumulator steht zu Beginn Null. Am Ende der Berechnung stehen die Ausgabedaten in vorher festgelegten Registern. Den Inhalt des Registers i

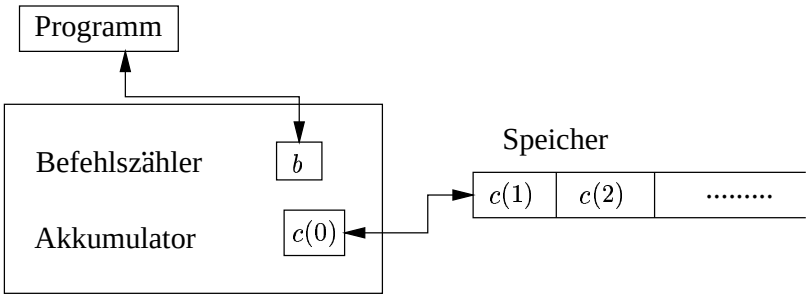

Abbildung 3.1: Schematische Darstellung der RAM

bezeichnet man mit $c(i)$.

Die RAM kann Befehle entsprechend der folgenden Liste ausführen:

Befehl	Wirkung
LOAD i	$c(0) := c(i); \quad b := b+1$
STORE i	$c(i) := c(0); \quad b := b+1$
ADD i	$c(0) := c(0) + c(i); \quad b := b+1$
SUB i	$c(0) := \max\{0, c(0) - c(i)\}; \quad b := b+1$
MULT i	$c(0) := c(0) \cdot c(i); \quad b := b+1$
DIV i	$c(0) := \left\lfloor \frac{c(0)}{c(i)} \right\rfloor; \quad b := b+1$
GOTO j	$b := j$
IF $c(0) \# \ell$ GOTO j	$\begin{cases} b := j & \text{falls } c(0) \# \ell \\ b := b+1 & \text{sonst} \end{cases}$ wobei $\# \in \{\leq, \geq, <, >, \neq, =\}$
END	$b := b$

Die Befehle können modifiziert werden zu: CLOAD, CSTORE, CADD, CSUB, CMULT, CDIV, wobei in diesem Fall $c(i)$ durch die Konstante i ersetzt wird. Daneben gibt es noch die Befehle INDLOAD, INDSTORE, INDADD, INDSUB, INDMULT, INDDIV, INDGOTO, bei denen $c(i)$ durch $c(c(i))$ ersetzt wird (indirekte Adressierung).

Üblicherweise wird das „uniforme“ Kostenmodell verwendet. Dabei „kostet“ jede Programmzeile, bis auf END, eine „Einheit“. Dieses Kostenmodell ist gerechtfertigt, solange keine großen Zahlen auftreten. Ansonsten ist das „logarithmische“ Kostenmodell realistischer. Kosten entsprechen dann der Länge der benutzten Zahlen.

3.2 Die Turing-Maschine

3.2.1 Der Aufbau der Turing-Maschine

Erfinder: Alan Turing (1936)

Die Turing-Maschine besteht aus einem beidseitig unendlichen Eingabe– und Rechenband mit einem freibeweglichen Lese-/Schreibkopf, der von einer endli-

chen Kontrolle gesteuert wird. Das Eingabe– und Rechenband enthält eine Folge von Symbolen. Die Kontrolle ist in einem von endlich vielen Zuständen. Dies entspricht dem Befehlszähler der Registermaschine. Die Zellen des Bandes entsprechen den Registern der Registermaschine und enthalten jeweils höchstens ein Symbol aus dem Bandalphabet. Ist die Turing-Maschine in einem bestimmten Zustand und liest ein Symbol, so geht sie in einen Folgezustand über, überschreibt eventuell das Symbol und bewegt den Lese-/Schreibkopf eine Stelle nach rechts, nach links oder überhaupt nicht.

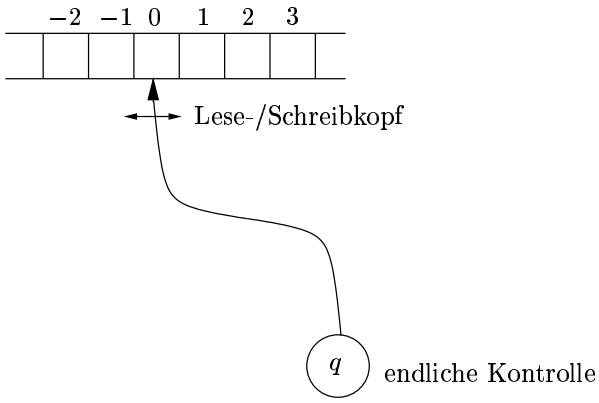

Abbildung 3.2: Schematische Darstellung der Turing-Maschine

3.1 Definition
Eine deterministische Turing-Maschine ((D)TM) besteht aus:

- Q, *einer endlicher Zustandsmenge,*
- Σ, *einem endlichen Eingabealphabet,*
- $\sqcup$, *einem Blanksymbol mit* $\sqcup \notin \Sigma$,
- Γ, *einem endlichen Bandalphabet mit* $\Sigma \cup \{\sqcup\} \subseteq \Gamma$,
- $s \in Q$, *einem Startzustand*
- $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$, *einer Übergangsfunktion.*

  Dabei bedeutet L eine Bewegung des Lese-/Schreibkopfes nach links, R eine Bewegung nach rechts und N ein Stehenbleiben. Die Übergangsfunktion beschreibt, wie das aktuell eingelesene Zeichen verarbeitet werden soll.

- $F \subseteq Q$, *einer Menge von Endzuständen.*

  Die Menge der Endzustände kann auch entfallen.

Bemerkung:
Für $q \in F$ gilt: $\forall\, a \in \Gamma\colon \delta(q, a) = (q, a, N)$, d.h. die Berechnung der Turing-Maschine stoppt.

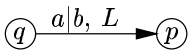

Abbildung 3.3: Übergang von Zustand q nach p

3.2 Bemerkung

(1) Der Übergang $\delta(q, a) = (p, b, L)$ wird graphisch dargestellt wie in Abbildung 3.3.

Bedeutung: Ist die Turing-Maschine im Zustand q und liest das Symbol a, so überschreibt sie dieses a mit b, geht auf dem Band eine Stelle nach links und wechselt in den Zustand p.

(2) Die Turing-Maschine startet im Zustand s, wobei der Lese-/Schreibkopf an der linkesten Stelle des Bandes, in der ein Eingabesymbol steht, positioniert ist. (Konvention)

(3) Die Turing-Maschine stoppt, wenn sie zum ersten Mal in einen Endzustand kommt oder in einem Zustand q ein Symbol a liest und $\delta(q, a) = (q, a, N)$ ist. Das bedeutet insbesondere, daß Übergänge, die aus Endzuständen herausführen, sinnlos sind.

Beispiel :
Die folgende Turing-Maschine (Abbildung 3.4) erkennt alle Wörter aus $\{0, 1\}^*$, die mit einer Eins beginnen.

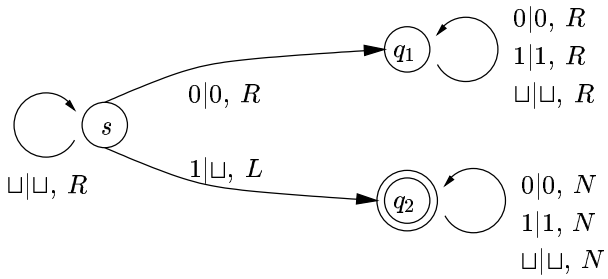

Abbildung 3.4: Turing-Maschine für die Sprache aller Wörter aus $\{0, 1\}^*$, die mit einer Eins beginnen

Außerdem löscht die Turing-Maschine die führende Eins, falls vorhanden, und läßt alles andere auf dem Band unverändert. Der Lese-/Schreibkopf steht nach dem Stop links neben der Stelle an der die führende Eins gelesen wurde. Der Zustand q_1 ist unwesentlich. ■

Bemerkungen:

(1) Es gibt Eingaben, für die eine Turing-Maschine unter Umständen niemals stoppt. Im obigen Beispiel sind das alle Folgen, die nicht mit Eins beginnen.

(2) Eine Turing-Maschine erkennt nicht nur Mengen von Wörtern (≡ Sprachen), sondern sie verändert auch die Eingabe und hat insofern auch eine Ausgabe (= Inhalt des Bandes nach der Bearbeitung). Die Turing-Maschine realisiert also eine partielle Funktion $f\colon \Sigma^* \to \Gamma^*$. Im obigen

Beispiel ist

$$f(w) = \begin{cases} v & \text{falls } w = 1v \\ \text{undefiniert} & \text{sonst} \end{cases}$$

(3) Oft werden wir die Turing-Maschine beziehungsweise deren Übergangsfunktion nur unvollständig beschreiben, zum Beispiel durch

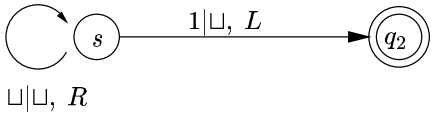

Abbildung 3.5: unvollständige Übergangsfunktion

Dabei ist eine Vervollständigung immer möglich. Wenn für eine bestimmte Kombination q, a kein Übergang $\delta(q, a)$ definiert ist, dann stoppt die Turing-Maschine im Zustand q.

3.3 Definition
Eine Turing-Maschine ***akzeptiert*** *eine Eingabe $w \in \Sigma^*$, wenn sie nach Lesen von w in einem Zustand aus F stoppt. Sie* ***akzeptiert*** *eine Sprache L genau dann, wenn sie ausschließlich Wörter aus $w \in L$ als Eingabe akzeptiert.*

3.4 Definition

1. *Eine Sprache $L \subseteq \Sigma^*$ heißt* ***rekursiv*** *oder* ***entscheidbar****, wenn es eine Turing-Maschine gibt, die auf allen Eingaben stoppt und eine Eingabe w genau dann akzeptiert, wenn $w \in L$ gilt.*

2. *Eine Sprache $L \subseteq \Sigma^*$ heißt* ***rekursiv-aufzählbar*** *oder* ***semi-entscheidbar****, wenn es eine Turing-Maschine gibt, die genau die Eingaben w akzeptiert für die $w \in L$. Das Verhalten der Turing-Maschine für Eingaben $w \notin L$ ist damit nicht definiert. D.h., die Turing-Maschine stoppt entweder nicht in einem Endzustand oder aber stoppt gar nicht.*

Notation
Oft wird die Situation, in der sich eine Turing-Maschine $\mathcal{M} := (Q, \Sigma, \Gamma, \delta, s, F)$ befindet, durch die Angabe der **Konfiguration** in der Form $w(q)av$ codiert, wobei $w, v \in \Gamma^*$, $a \in \Gamma$ und $q \in Q$ sind. Dies bedeutet, daß sich $\mathcal{M}$ gerade im Zustand q befindet; der Lesekopf steht auf dem Zeichen a; links davon steht das Wort w auf dem Rechenband und rechts davon das Wort v.

Beispiel :
Abbildung 3.6 zeigt eine Turing-Maschine, die $L^= := \{0^n 1^n : n \geq 1\}$ akzeptiert. Die Überprüfung der Eingabe 0011 ergibt die folgenden Konfigurationen:

$\sqcup(s)0011$, $0(s)011$, $00(s)11$, $001(q_1)1$, $0011(q_1)\sqcup$, $001(q_2)1$, $00(q_3)1$, $0(q_3)01$, $\sqcup(q_3)001$, $\sqcup(q_3)\sqcup 001$, $\sqcup(q_4)001$, $\sqcup(q_5)01$, $0(q_6)1$, $01(q_6)\sqcup$, $0(q_2)1$, $\sqcup(q_3)0$, $\sqcup(q_3)\sqcup 0$, $\sqcup(q_4)0$, $\sqcup(q_5)\sqcup$, $\sqcup(q_7)\sqcup$.

■

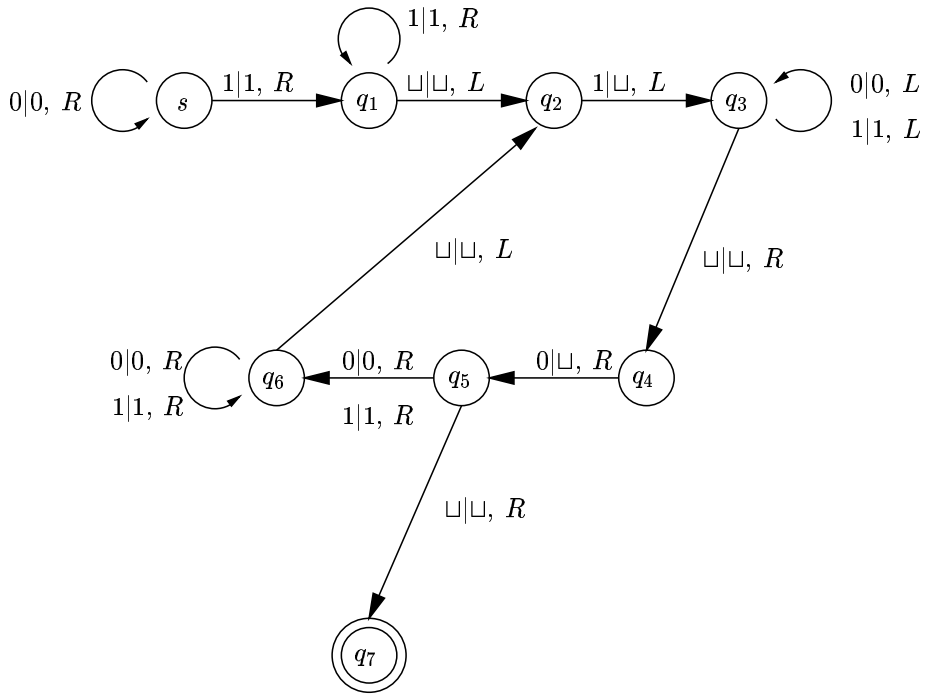

Abbildung 3.6: Turing-Maschine für $L^{=}$

Wie bereits erwähnt, kann eine Turing-Maschine auch eine Funktion berechnen.

3.5 Definition

1. *Eine Funktion $f\colon \Sigma^* \to \Gamma^*$ heißt* ***(Turing-)berechenbar*** *oder* ***totalrekursiv****, wenn es eine Turing-Maschine gibt, die bei Eingabe von $w \in \Sigma^*$ den Funktionswert $f(w) \in \Gamma^*$ ausgibt.*

2. *Eine Turing-Maschine* ***realisiert*** *eine Funktion $f\colon \Sigma^* \to \Gamma^*$, falls gilt:*

$$f(w) = \begin{cases} \textit{Ausgabe der Turing-Maschine, wenn sie bei Eingabe } w \textit{ stoppt} \\ \textit{undefiniert sonst} \end{cases}$$

Beispiel :
Eine Turing-Maschine, die zur Eingabe $w \in (0 \cup 1)^*$ eine Eins addiert, wobei w als binäre Zahl aufgefaßt wird. Es sollen nur Eingaben ohne führende Nullen akzeptiert werden beziehungsweise Null.

Dabei sind die Zustände jeweils für die folgenden Aufgaben verantwortlich:

q_1: Bewegung des Lese-/Schreibkopfes nach rechts bis zum Eingabeende

q_2: Bildung des Übertrages, der durch die Addition von Eins zu einer bereits vorhandenen Eins entsteht

q_3: Bewegung des Lese-/Schreibkopfes nach links, nachdem die Aufsummierung abgeschlossen ist (kein Übertrag mehr)

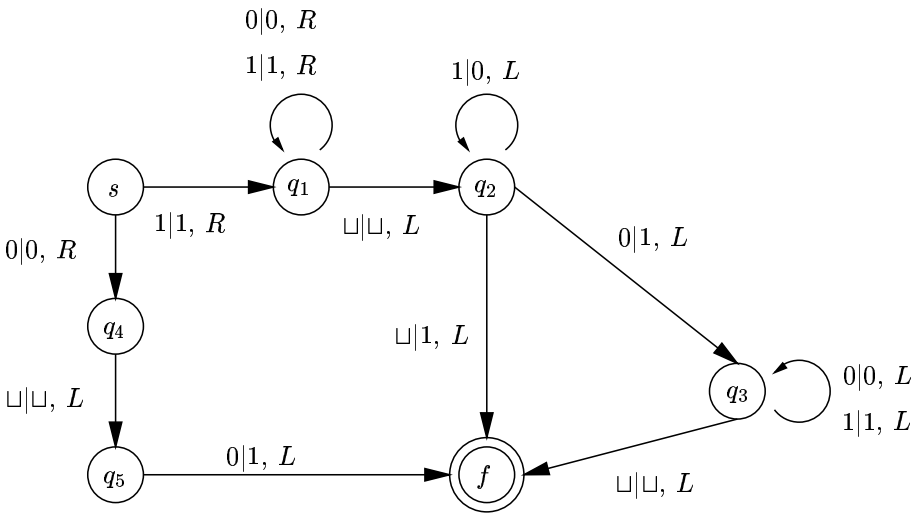

Abbildung 3.7: Turing-Maschine für Addition

q_4, q_5: Sonderbehandlung für den Fall der Eingabe 0

f: Endzustand

Es gilt: $f(w) = \begin{cases} w+1 & \text{falls } w \in 0 \cup 1(0 \cup 1)^*, \\ & w \text{ interpretiert als Binärzahl} \\ \text{undefiniert} & \text{sonst} \end{cases}$ ■

Wir wollen die Konzepte der Entscheidbarkeit von Sprachen und der Berechenbarkeit von Funktionen zusammenbringen:

- Eine Turing-Maschine akzeptiert eine Sprache L, wenn sie genau auf den Eingaben $w \in L$ in einem ausgezeichneten Endzustand stoppt. L ist entscheidbar, wenn es eine Turing-Maschine gibt, die auf allen Eingaben stoppt und L akzeptiert.
- Die Funktion f heißt berechenbar, wenn eine Turing-Maschine existiert, die f realisiert.

Man kann eine Turing-Maschine $\mathcal{M}$, die auf allen Eingaben stoppt, so modifizieren, daß es zwei ausgezeichnete Zustände q_J und q_N gibt und daß $\mathcal{M}$ stets in einem der Zustände q_J oder q_N hält. Dabei stoppt sie bei der Eingabe w genau dann in q_J, wenn sie w akzeptiert, ansonsten in q_N. Damit ist die Sprache L genau dann entscheidbar, wenn es eine Turing-Maschine gibt, die immer in einem der Zustände $\{q_J, q_N\}$ stoppt, wobei sie gerade für $w \in L$ in q_J hält.

3.6 Korollar

- *Eine Sprache $L \subseteq \Sigma^*$ ist **entscheidbar** genau dann, wenn ihre **charakteristische Funktion** χ_L berechenbar ist, wobei gilt:*

$$\chi_L : \Sigma^* \to \{0,1\} \quad \textit{mit} \qquad \chi_L(w) = \begin{cases} 1 & \textit{falls } w \in L \\ 0 & \textit{sonst} \end{cases}$$

- *Eine Sprache L ist* ***semientscheidbar*** *genau dann, wenn die Funktion χ_L^* berechenbar ist, wobei gilt:*

$$\chi_L^*(w) = \begin{cases} 1 & \text{falls } w \in L \\ \text{undefiniert} & \text{sonst} \end{cases}$$

3.2.2 Die Church'sche These

Die Church'sche These besagt, daß die Menge der (Turing-)berechenbaren Funktionen genau die Menge der im intuitiven Sinne überhaupt berechenbaren Funktion ist.

Interpretation

Turing-Maschinen sind formale Modelle für Algorithmen. Kein Berechnungsverfahren kann „algorithmisch" genannt werden, wenn es nicht von einer Turing-Maschine ausführbar ist.

Bemerkung:
Die Church'sche These ist „nur" eine These, kann also nicht bewiesen werden. Sie ist aber in der Informatik allgemein akzeptiert.

Begründung

- Es existieren keine Beispiele von Funktionen, die als intuitiv berechenbar angesehen werden, aber nicht Turing-berechenbar sind.
- Alle Versuche, realistische Modelle aufzustellen, die mächtiger sind als Turing-Maschinen, schlugen fehl.
- Eine Reihe von völlig andersartigen Ansätzen, den Begriff der Berechenbarkeit formal zu fassen, wie zum Beispiel die Registermaschine, haben sich als „äquivalent" erwiesen.

3.2.3 Erweiterungen der Turing-Maschine

1. **Mehrere Lese-/Schreibköpfe**

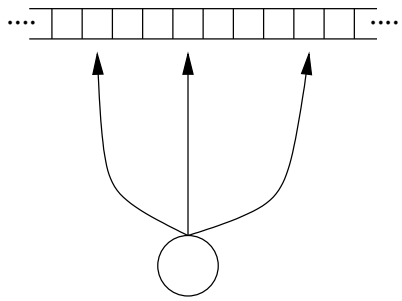

Abbildung 3.8: TM mit mehreren Lese-/Schreibköpfen

Siehe Abbildung 3.8: Mehrere Lese-/Schreibköpfe ($n \in \mathbb{N}$) greifen auf das eine Eingabeband zu und werden von der endlichen Kontrolle gesteuert. Die Übergangsfunktion ist dann vom Typ $\delta\colon Q \times \Gamma^n \to Q \times \Gamma^n \times \{L, N, R\}^n$, und die Zustände $q \in Q$ kann man als n–Tupel auffassen. Außerdem ist es nötig eine Prioritätenregel für die einzelnen Köpfe anzugeben, falls mehrere auf einem Feld des Eingabebandes stehen.

2. **Mehrere Bänder**

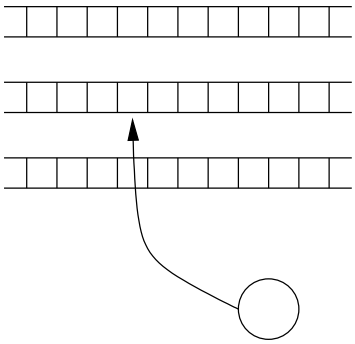

Abbildung 3.9: TM mit mehreren Bändern

Siehe Abbildung 3.9: Ein Lese-/Schreibkopf kann auf mehrere Eingabebänder ($n \in \mathbb{N}$) zugreifen. Die Übergangsfunktion ist dann vom Typ

$$\delta\colon Q \times \Gamma \times \{1, \ldots, n\} \to Q \times \Gamma \times \{L, N, R\} \times \{1, \ldots, n\}.$$

3. **Mehrere Lese-/Schreibköpfe für mehrere Bänder**

Wir haben jetzt für jedes der n Bänder einen Lese-/Schreibkopf. Die Übergangsfunktion ist dann vom Typ

$$\delta\colon Q \times \Gamma^n \times \{1, \ldots, m\}^n \to Q \times \Gamma^n \times \{L, N, R\}^n \times \{1, \ldots, m\}^n.$$

4. **Mehrdimensionale Bänder**

Das Eingabeband ist nun mehrdimensional, es hat zum Beispiel die Dimension Zwei, (siehe Abbildung 3.10). Wir sprechen dann von einem Arbeitsfeld. Dabei ist

$$\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L(eft), U(p), R(ight), D(own), N(othing)\}$$

Fragestellungen der Art: „Wann stoppt eine Mehrkopf Maschine?“ oder „Welcher Kopf ‚gewinnt‘, wenn mehrere Köpfe (verschiedene) Symbole an dieselbe Stelle schreiben wollen?“ müssen bei solchen Modifikationen noch geklärt werden. Es hat sich allerdings gezeigt, daß keine dieser „Erweiterungen“ mehr leistet, als eine „normale“ Turing-Maschine. Man kann zum Beispiel beweisen, daß jede k–Band Turing-Maschine durch eine 1–Band Turing-Maschine simuliert werden kann. Das gleiche gilt auch für alle anderen angegebenen Modifikationen.

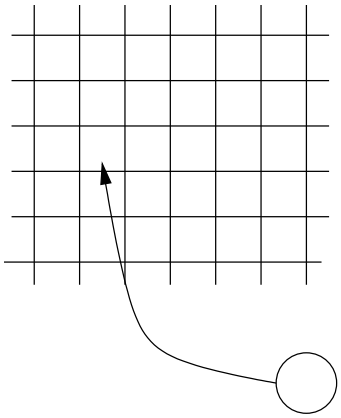

Abbildung 3.10: TM mit zweidimensionalem Band

3.3 Die universelle Turing-Maschine und unentscheidbare Probleme

Wir haben bisher nur Turing-Maschinen betrachtet, die eine bestimmte Aufgabe erfüllen. Wir wollen nun eine „universelle“ Turing-Maschine angeben, die als Eingabe ein Programm und eine spezielle Eingabe erhält. Die Aufgabe der universellen Turing-Maschine besteht darin, das gegebene Programm auf der gegebenen Eingabe auszuführen.

Wir betrachten dazu eine „normierte“ Turing-Maschine, d.h.

- $Q := \{q_1, \ldots, q_n\}$
- $\Sigma := \{a_1, \ldots, a_k\}$
- $\Gamma := \{a_1, \ldots, a_k, a_{k+1}, \ldots, a_l\}$
- $s := q_1$
- $F := \{q_2\}$

Dies bedeutet keine Einschränkung in der Mächtigkeit der Turing-Maschinen; d.h. jede beliebige Turing-Maschine kann durch eine derart normiert Turing-Maschine der obigen Form simuliert werden. Jede normierte Turing-Maschine $\mathcal{M}$ läßt sich eindeutig kodieren durch folgende Kodierungsvorschrift:

1. Kodiere $\delta(q_i, a_j) = (q_r, a_s, d_t)$ durch $0^i\,1\,0^j\,1\,0^r\,1\,0^s\,1\,0^t$,
   wobei $d_t \in \{d_1, d_2, d_3\}$ und d_1 für L, d_2 für R und d_3 für N steht.
2. Die Turing-Maschine wird dann kodiert durch:

   $$111\text{code}_1 11\text{code}_2 11 \ldots 11\text{code}_z 111,$$

   wobei code_i für $i = 1, \ldots, z$ alle Funktionswerte von δ in beliebiger Reihenfolge beschreibt.

Die eigentlichen Werte der Turing-Maschine werden also (unär) durch Nullen beschrieben und die Einsen dienen als Begrenzung der Eingabewerte.

Diese Kodierung nennt man die **Gödelnummer** einer Turing-Maschine und wird mit $\langle \mathcal{M} \rangle$ bezeichnet (zur Turing-Maschine $\mathcal{M}$). Jede Turing-Maschine kann also durch ein Wort aus $(0 \cup 1)^*$ kodiert werden. Umgekehrt beschreibt jedes Wort aus $\{0 \cup 1\}^*$ (höchstens) eine Turing-Maschine. Wir vereinbaren, daß ein Wort, das keine Turing-Maschine in diesem Sinne beschreibt, (zum Beispiel ε, 0, 000) eine Turing-Maschine kodiert, die $\emptyset$ akzeptiert.

Eine *universelle Turing-Maschine* erhält als Eingabe ein Paar $(\langle \mathcal{M} \rangle, w)$, wobei $w \in \{0,1\}^*$ ist, und sie simuliert $\mathcal{M}$ auf w.

Beispiel :
Sei $\mathcal{M} = (\{q_1, q_2, q_3\}, \{0,1\}, \sqcup, \{0,1,\sqcup\}, \delta, q_1, \{q_2\})$, mit

$$\delta(q_1, 1) = (q_3, 0, R)$$

$$\delta(q_3, 0) = (q_1, 1, R)$$

$$\delta(q_3, 1) = (q_2, 0, R)$$

$$\delta(q_3, \sqcup) = (q_3, 1, L)$$

$\mathcal{M}$ zusammen mit der Eingabe 1011 ist dann:

1110100100010100110001010100100110001001001010011000100010001001011110 11

■

3.7 Definition
Zu $w \in \{0,1\}^*$ sei $\boldsymbol{T_w}$ die Turing-Maschine mit der Gödelnummer (GN) w, beziehungsweise die Turing-Maschine, die $\emptyset$ akzeptiert (d.h. T_w stoppt nie, wenn w keine Gödelnummer ist). $\boldsymbol{L(T_w)}$ *ist die Sprache, die von T_w akzeptiert wird.*

Wir konstruieren eine Sprache L_d, die sogenannte **Diagonalsprache**, wie folgt: Betrachte die Wörter aus $\{0,1\}^*$ in *kanonischer* Reihenfolge, d.h. w_i steht vor w_j $(i < j)$, falls $|w_i| < |w_j|$ oder $|w_i| = |w_j|$ und w_i lexikographisch vor w_j steht. $\mathcal{M}_j$ sei die Turing-Maschine, deren Gödelnummer die j-te Nummer bei kanonischer Reihenfolge aller Gödelnummern ist. Wie konstruieren eine unendliche Tabelle, an deren Position (i,j) für $1 \leq i,j < \infty$ eine Null oder eine Eins steht und welche beinhaltet, ob w_i in $L(\mathcal{M}_j)$ ist. Damit gilt für die Einträge

$$(i,j) = \begin{cases} 1 & \text{falls } \mathcal{M}_j \ w_i \text{ akzeptiert} \\ 0 & \text{sonst} \end{cases}$$

Definiere dazu

$$L_d := \{w_i \mid \mathcal{M}_i \text{ akzeptiert } w_i \text{ nicht}\}.$$

L_d enthält also alle w_i, für die auf der Diagonalen an der Stelle (i,i) eine Null steht. (Dies führt später zu einem Diagonalbeweis (Cantor).)

Beispiel:

$w \in \{0,1\}^*$	Gödelnummer i	j	k					
$\vdots$	$\vdots$							
w_i	1	**0**	1	0	1	0	0	$w_i \in L_d$
w_j	0	0	**1**	0	0	1	1	$w_j \notin L_d$
w_k	1	0	0	**1**	1	0	1	$w_k \notin L_d$
$\vdots$	$\vdots$							

■

3.8 Satz
Die Sprache L_d ist nicht entscheidbar.

Beweis: Falls L_d entscheidbar ist, gibt es eine Turing-Maschine $\mathcal{M}_i$, die

(i) stets hält und

(ii) genau die $w \in L_d$ akzeptiert.

Wende nun $\mathcal{M}_i$ auf w_i an:

1. Falls $w_i \in L_d$, dann wird w_i, wegen (ii) von $\mathcal{M}_i$ akzeptiert. Dies ist ein Widerspruch zur Definition von L_d.
2. Falls $w_i \notin L_d$, dann akzeptiert $\mathcal{M}_i$ das Wort w_i wegen (ii) nicht. Dies ist auch ein Widerspruch zur Definition von L_d.

□

3.9 Korollar
Die Sprache $L_d^c := \{0,1\}^* \backslash L_d$ ist nicht entscheidbar.

Beweis: Falls L_d^c entscheidbar ist, gibt es eine Turing-Maschine, die L_d^c entscheidet. Diese kann aber leicht zu einer Turing-Maschine modifiziert werden, die L_d entscheidet. Dies ist ein Widerspruch zu Satz 3.8. □

Bemerkung:
Korollar 3.9 kann folgendermaßen interpretiert werden: Das Problem, ob eine Turing-Maschine auf einer Eingabe w stoppt, ist nicht entscheidbar.

3.10 Definition (Halteproblem)
*Das **Halteproblem** definiert folgende Sprache*

$$\mathcal{H} := \{wv \mid T_w \text{ hält auf der Eingabe } v\}\,.$$

3.11 Satz
$\mathcal{H}$ ist nicht entscheidbar.

Beweis: Angenommen es existiert eine stets haltende Turing-Maschine, die $\mathcal{H}$ entscheidet. Wir konstruieren daraus eine stets haltende Turing-Maschine, die L_d^c entscheidet, was im Widerspruch zu Korollar 3.9 steht.

Sei w eine Eingabe, für die wir entscheiden wollen, ob $w \in L_d^c$. Wir können wie folgt vorgehen:

1. Berechne das i, so daß $w = w_i$ ist.
2. Berechne daraus die Codierung von $\mathcal{M}_i$.
3. Wende die Turing-Maschine für $\mathcal{H}$ auf $\langle \mathcal{M}_i \rangle \cdot w_i$ an.

Falls $\langle \mathcal{M}_i \rangle \cdot w_i$ nicht akzeptiert wird, dann hält $\mathcal{M}_i$ nicht auf w_i. Nach Definition von $\mathcal{H}$ ist also $w_i \in L_d$ und damit $w_i \notin L_d^c$. Falls $\langle \mathcal{M}_i \rangle \cdot w_i$ akzeptiert wird, dann hält $\mathcal{M}_i$ auf w_i. Dann können wir auf der universellen Turing-Maschine die Berechnung von $\mathcal{M}_i$ auf w_i simulieren. □

3.12 Definition
*Die **universelle Sprache** L_u über $\{0,1\}$ ist definiert durch*

$$L_u := \{wv \mid v \in L(T_w)\} .$$

L_u ist also die Menge aller Wörter wv für die T_w bei der Eingabe v hält und v akzeptiert.

3.13 Satz
Die Sprache L_u ist nicht entscheidbar.

Beweis: Wir zeigen, daß L_u eine Verallgemeinerung von L_d^c ist. Das heißt wir nehmen wieder an, daß es eine Turing-Maschine zur Entscheidung von L_u gibt und zeigen, daß wir damit auch L_d^c entscheiden können. Wir können dazu wie folgt vorgehen:

1. Berechne das i, für das $w = w_i$.
2. Berechne die Gödelnummer $\langle \mathcal{M}_i \rangle$.
3. Wende die Turing-Maschine für L_u auf $\langle \mathcal{M}_i \rangle \, w_i$ an.

Wäre L_u entscheidbar, so auch L_d^c. Dies ist ein Widerspruch zu Korollar 3.9. □

3.14 Satz
Die Sprache L_u ist semi-entscheidbar.

Beweis: Wir benutzen die universelle Turing-Maschine, mit der Eingabe wv:

- Falls T_w die Eingabe v akzeptiert, geschieht dies nach endlich vielen Schritten und die universelle Turing-Maschine akzeptiert wv.
- Falls T_w die Eingabe v nicht akzeptiert, wird wv von der universellen Turing-Maschine ebenfalls nicht akzeptiert. Dies ist unabhängig davon, ob die Simulation stoppt oder nicht.

□

Bemerkung:
Die Begriffe entscheidbar und semi-entscheidbar unterscheiden sich tatsächlich.

Wir haben bisher gezeigt, daß wir kein Programm schreiben können, das für ein Turing-Maschinen-Programm $\langle \mathcal{M} \rangle$ und eine Eingabe w entscheidet, ob $\mathcal{M}$ auf der Eingabe w hält. Wir werden im folgenden sehen, daß wir aus einem Programm im allgemeinen keine „nicht-trivialen" Eigenschaften der von dem Programm realisierten Funktion ableiten können.

3.15 Satz (Satz von Rice)
Sei R die Menge der von Turing-Maschinen berechenbaren Funktionen und S eine nicht-triviale Teilmenge von R ($\emptyset \neq S \neq R$). Dann ist die Sprache

$$L(S) := \{\langle \mathcal{M} \rangle \mid \mathcal{M} \text{ berechnet eine Funktion aus } S\}$$

nicht entscheidbar.

Beweis (Skizze):

- Zeige, daß $\mathcal{H}_\varepsilon := \{\langle \mathcal{M} \rangle \mid \mathcal{M}$ hält auf der Eingabe $\varepsilon\}$ unentscheidbar ist
- Zeige, daß $\mathcal{H}_\varepsilon^c$ nicht entscheidbar ist.
- Führe den Widerspruchsbeweis für die Nicht-Entscheidbarkeit von $L(S)$, indem ausgehend von einer Turing-Maschine $\mathcal{M}'$ für $L(S)$ eine Turing-Maschine $\mathcal{M}''$ für $\mathcal{H}_\varepsilon^c$ konstruiert wird.

□

Bemerkungen:

(1) Der Satz von Rice hat weitreichende Konsquenzen. Es ist danach zum Beispiel für Programme nicht entscheidbar, ob die durch sie definierte Sprache endlich, leer, unendlich oder ganz Σ^* ist.

(2) Wir haben hier nur die Unentscheidbarkeit von L_d direkt bewiesen. Die anderen Beweise folgten dem folgenden Schema: Um zu zeigen, daß ein Problem A unentscheidbar ist, zeigen wir, wie man mit einem Entscheidungsverfahren für A ein bekanntermaßen unentscheidbares Problem B entscheiden kann. Dies liefert den gewünschten Widerspruch.

Das Post'sche Korrespondenzproblem: Ein weiteres unentscheidbares Problem ist das Post'sche Korrespondezproblem (PKP). Dabei ist eine endliche Folge von Wortpaaren

$$K = ((x_1, y_1), \ldots, (x_n, y_n))$$

über einem endlichen Alphabet Σ gegeben. Es gilt weiter $x_i \neq \varepsilon$ und $y_i \neq \varepsilon$. Gefragt ist nun, ob es eine endliche Folge von Indizes $i_1, \ldots, i_k \in \{1, \ldots, n\}$ gibt, so daß $x_{i_1} \ldots x_{i_k} = y_{i_1} \ldots y_{i_k}$ gilt.

Beispiele:

(1) $K = ((10, 101), (10111, 10), (10, 0))$ hat die Lösung $(2, 1, 1, 3)$, denn es gilt:

$$x_2 x_1 x_1 x_3 = 101111110 = y_2 y_1 y_1 y_3$$

(2) $K = ((10, 101), (011, 11), (101, 011))$ hat keine Lösung

(3) $K = ((001, 0), (01, 011), (01, 101), (10, 001))$ hat eine Lösung der Länge 66.

■

3.16 Satz
Das Post'sche Korrespondezproblem ist nicht entscheidbar

Beweis: Beweis über Nicht-Entscheidbarkeit des Halteproblem. □

Bemerkung:
Eigenschaften von (semi-)entscheidbaren Sprachen:

1. Die entscheidbaren Sprachen sind abgeschlossen unter Komplementbildung, Schnitt und Vereinigung.
2. Die semi-entscheidbaren Sprachen sind abgeschlossen unter Schnitt und Vereinigung, aber nicht unter Komplementbildung.

Kapitel 4

Komplexitätsklassen

Wir haben bisher nur die Frage erörtert: „Ist eine Sprache L entscheidbar oder nicht? “ beziehungsweise „Ist eine Funktion berechenbar oder nicht?“ Dies führte zu einer Aussage, ob ein Problem lösbar ist oder nicht. Darüberhinaus interessiert uns die Frage: „Wie effizient kann ein Problem gelöst werden?“. Die Effizienz bezieht sich dabei immer auf die Laufzeit, die benötigt wird, um das Problem zu lösen. Wir gehen dazu auch im folgenden (entsprechend der Church'schen These) davon aus, daß die Turing-Maschine ein aussagekräftiges Berechnungsmodell ist.

Bisher haben wir nur deterministische Turing-Maschinen betrachtet. Wir werden nun auch nichtdeterministische Turing-Maschinen einführen.

Frage: Gibt es einen „wesentlichen Effizienzgewinn“ beim Übergang der deterministischen Turing-Maschine zur nichtdeterministischen Turing-Maschine? (Vergleiche dazu die Äquivalenz bei endlichen Automaten). Dahinter verbirgt sich eine der wichtigsten offenen Fragen der theoretischen Informatik: $\mathcal{P} \neq \mathcal{NP}$?

4.1 Sprachen, Probleme, Zeitkomplexität

Fragen: Wie stehen Sprache und Probleme im Zusammenhang? Wie sieht ein typisches Problem aus?

Beispiel Traveling Salesman Problem (TSP):
Gegeben sei ein vollständiger Graph $G = (V, E)$, d.h.

$$V := \{1, \ldots, n\}, \quad E := \{\{u, v\} \mid u, v \in V, u \neq v\},$$

sowie eine Längenfunktion $c\colon E \to \mathbb{Z}^+$.

① Gesucht ist eine Tour (Rundreise), die alle Elemente aus V enthält und minimale Gesamtlänge unter allen solchen Touren hat. Gesucht ist also eine Permutation π auf V, so daß

$$\sum_{i=1}^{n-1} c(\pi(i), \pi(i+1)) + c(\pi(n), \pi(1)) \qquad \text{minimal ist}$$

Dies ist ein **Optimierungsproblem**.

Eine „etwas schwächere“ Variante wäre:

② Gesucht ist die Länge einer minimalen Tour.

Diese Variante ist insofern „schwächer“, als mit ① auch ② lösbar ist. Noch „schwächer“ ist:

③ Gegeben sei zusätzlich zu G und c auch ein Parameter $k \in \mathbb{Z}^+$. Die Frage ist nun: „Gibt es eine Tour, deren Länge höchstens k ist?“

Offensichtlich können wir mit ① beziehungsweise ② auch ③ lösen. ③ ist das zugehörige **Entscheidungsproblem** zu ①.

Wir werden hier zunächst nur Entscheidungsprobleme, also Probleme, bei denen die Lösung aus einer Antwort „Ja“ oder „Nein“ besteht, betrachten, da diese mit Sprachen korrespondieren.

Um zu motivieren, daß es „zulässig“ ist, sich auf Entscheidungsprobleme zu beschränken, werden wir am Beispiel des Traveling Salesman Problems feststellen, daß man „leicht“ mit ③ auch ② und ① lösen kann. ■

Zunächst betrachten wir ein formales Konzept für die Korrespondenz zwischen Sprachen und Entscheidungsproblemen:

Ein **Problem** Π ist gegeben durch:

1. eine allgemeine Beschreibung aller vorkommenden Parameter;
2. eine genaue Beschreibung der Eigenschaften, die die Lösung haben soll.

Ein **Problembeispiel** I (**Instanz**) von Π erhalten wir, indem wir die Parameter von Π festlegen.

Wir interessieren uns für die **Laufzeit** von Algorithmen beziehungsweise Berechnungen. Diese wird in der „Größe des Problems“ gemessen. Die Größe des Problems ist abhängig von der **Beschreibung** oder **Kodierung** aller Problembeispiele.

4.1 Definition
*Ein **Kodierungsschema** s ordnet jedem Problembeispiel eines Problems eine Zeichenkette oder* Kodierung *über einem Alphabet Σ zu. Die **Inputlänge** eines Problembeispiels ist die Anzahl des Symbole seiner Kodierung. Dabei gibt es natürlich verschiedene Kodierungsschemata für ein bestimmtes Problem.*

Beispiel :
Zahlen können dezimal, binär, unär, usw. kodiert werden. Die Inputlänge von 5127 beträgt dann 4 für dezimal, 13 für binär und 5127 für unär. ■

Wir werden uns auf „vernünftige“ Schemata festlegen:

1. Die Kodierung eines Problembeispiels sollte keine überflüssigen Informationen enthalten.
2. Zahlen sollen binär (oder k-är für $k \neq 1$) kodiert sein.

Dies bedeutet, die Kodierungslänge

- einer ganzen Zahl n ist $\lceil \log_k |n| + 1 \rceil + 1 =: \langle n \rangle$;
- einer rationalen Zahl $r = \frac{p}{q}$ ist $\langle r \rangle = \langle p \rangle + \langle q \rangle$;
- eines Vektors $X = (x_1, \ldots, x_n)$ ist $\langle X \rangle := \sum_{i=1}^{n} \langle x_i \rangle$;
- einer Matrix $A \in \mathbb{Q}^{m \times n}$ ist $\langle A \rangle := \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle$.
- Ein Graph $G = (V, E)$ kann zum Beispiel durch die Kodierung seiner *Adjazenzmatrix*, ein „gewichteter" Graph durch die Kodierung der *Gewichtsmatrix* beschrieben werden.

4.2 Definition
Zwei Kodierungsschemata s_1, s_2 heißen ***äquivalent*** *bezüglich eines Problems Π, falls es Polynome p_1, p_2 gibt, so daß gilt:*

$$(|s_1(I)| = n \Rightarrow |s_2(I)| \leq p_2(n)) \text{ und } (|s_2(I)| = m \Rightarrow |s_1(I)| \leq p_1(m))$$

für alle Problembeispiele I von Π.

Ein Entscheidungsproblem Π können wir als Klasse von Problembeispielen D_Π *auffassen. Eine Teilmenge dieser Klasse ist $J_\Pi \subseteq D_\Pi$, die Klasse der* ***Ja–Beispiele****, d.h. die Problembeispiele deren Antwort „Ja" ist. Der Rest der Klasse $N_\Pi \subseteq D_\Pi$ ist die Klasse der* ***Nein–Beispiele****.*

Die Korrespondenz zwischen Entscheidungsproblemen und Sprachen ist festgelegt durch das Kodierungsschema. Ein Problem Π und ein Kodierungsschema $s : D_\Pi \to \Sigma$ zerlegen Σ^* in drei Klassen:

1. Wörter aus Σ^*, die *nicht* Kodierung eines Beispiels aus D_Π sind,
2. Wörter aus Σ^*, die Kodierung eines Beispiels $I \in N_\Pi$ sind,
3. Wörter aus Σ^*, die Kodierung eines Beispiels $I \in J_\Pi$ sind.

Die dritte Klasse ist die Sprache, die zu Π im Kodierungsschema s korrespondiert.

4.3 Definition
Die zu einem Problem Π und einem Kodierungsschema s ***zugehörige Sprache*** *ist*

$$L[\Pi, s] := \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} \Sigma \text{ ist das Alphabet zu } s \text{ und } x \text{ ist Kodierung} \\ \text{eines Ja–Beispiels } I \text{ von } \Pi \text{ unter } s, \text{ d.h. } I \in J_\Pi \end{array} \right\}$$

Wir betrachten im folgenden deterministische Turing-Maschinen mit zwei Endzuständen q_J, q_N, wobei q_J akzeptierender Endzustand ist. Dann wird die Sprache $L_\mathcal{M}$ akzeptiert von der Turing-Maschine $\mathcal{M}$, falls

$$L_\mathcal{M} = \{x \in \Sigma^* \mid \mathcal{M} \text{ akzepiert } x\}$$

4.4 Definition
Eine deterministische Turing-Maschine $\mathcal{M}$ ***löst*** *ein Entscheidungsproblem Π unter einem Kodierungsschema s, falls $\mathcal{M}$ bei jeder Eingabe über dem Eingabe-Alphabet in einem Endzustand endet und $L_{\mathcal{M}} = L[\Pi, s]$ ist.*

4.5 Definition
Für eine deterministische Turing-Maschine $\mathcal{M}$, die für alle Eingaben über dem Eingabe-Alphabet Σ hält, ist die ***Zeitkomplexitätsfunktion*** *$T_{\mathcal{M}} : \mathbb{Z}^+ \to \mathbb{Z}^+$ definiert durch*

$$T_{\mathcal{M}}(n) = \max \left\{ m \;\middle|\; \begin{array}{l} \text{es gibt eine Eingabe } x \in \Sigma^* \text{ mit } |x| = n, \text{ so} \\ \text{daß die Berechnung von } \mathcal{M} \text{ bei Eingabe } x \\ m \text{ Berechnungsschritte (Übergänge) benötigt,} \\ \text{bis ein Endzustand erreicht wird} \end{array} \right\}$$

4.6 Definition
Die Klasse ***$\mathcal{P}$*** *ist die Menge aller Sprachen L (Probleme), für die eine deterministische Turing-Maschine existiert, deren Zeitkomplexitätsfunktion polynomial ist, d.h. es existiert ein Polynom p mit*

$$T_{\mathcal{M}}(n) \leq p(n).$$

Bemerkung:
Da wir nur Kodierungsschemata, die äquivalent zur Binärkodierung sind, betrachten, brauchen wir das gewählte Kodierungsschema s nicht explizit angeben.

Zurück zu Entscheidungs– und Optimierungsproblemen. Wir zeigen am Beispiel des TSP, daß die Beschränkung auf das Entscheidungsproblem keine wirkliche Einschränkung ist. Wir zeigen, daß mit der Lösung für das Entscheidungsproblem auch das Optimierungsproblem „leicht“ gelöst werden kann.

4.7 Satz
Falls es einen Algorithmus gibt, der das Entscheidungsproblem des TSP in polynomialer Zeit löst, so gibt es auch einen Algorithmus, der das Optimierungsproblem in polynomialer Zeit löst.

Beweis: Sei $\mathcal{A}(n, c, k)$ ein polynomialer Algorithmus zur Lösung des Entscheidungsproblems des TSP bei der Eingabe eines Graphen $G = (V, E)$ mit $|V| = n$, Längenfunktion c, sowie Parameter $k \in \mathbb{Z}^+$.

Betrachte folgenden Algorithmus.

Algorithmus OPT–TOUR

Input: $G = (V, E)$, $c_{ij} = c(\{i, j\})$ für $i, j \in V := \{1, \ldots, n\}$

Output: d_{ij} $(1 \leq i, j \leq n)$, so daß alle bis auf n der d_{ij}-Werte den Wert $\left(\max\limits_{i,j} c_{ij}\right) + 1$ haben. Die restlichen n d_{ij}-Werte haben den Wert c_{ij} und geben genau die Kanten einer optimalen Tour an.

Algorithmus:

1. berechne $m := \max_{1 \le i,j \le n} c_{ij}$;
   setze L(ow) := 0 und H(igh) := $n \cdot m$;
2. solange $H \neq L$ gilt, führe aus:

$$\left.\begin{array}{l} \text{falls } \mathcal{A}\left(n, c, \left\lceil \frac{1}{2}(H+L) \right\rceil\right) = \text{„nein“ ist,} \\ \qquad \text{setze } L := \left\lceil \frac{1}{2}(H+L) \right\rceil + 1; \\ \text{sonst} \\ \qquad \text{setze } H := \left\lceil \frac{1}{2}(H+L) \right\rceil + 1; \end{array}\right\} (* \text{ binäre Suche } *)$$

   ($*$ Wenn $H = L$ gilt, so ist H die Länge einer optimalen Tour. $*$)
3. Setze OPT := H;
4. für $i = 1 \ldots n$ führe aus
5. für $j = 1 \ldots n$ führe aus
6. setze $R := c_{ij}$; $c_{ij} := m + 1$;
   falls $A(n, c, OPT)$ „nein“ ist, setze $c_{ij} := R$;
7. setze $d_{ij} = c_{ij}$;

Laufzeit:

In 2. wird $\mathcal{A}(n, c, k)$ etwa $\log(n \cdot m)$–mal aufgerufen, in 4. wird $\mathcal{A}(n, c, opt)$ etwa n^2–mal aufgerufen. Es finden also $\mathcal{O}(n^2 + \log(nm))$ Aufrufe von $\mathcal{A}$ statt. Die Inputlänge ist $\mathcal{O}(n + \log(n \max c_{ij}))$. Da $\mathcal{A}$ polynomial ist, ist dies also auch OPT-TOUR.

□

4.2 Nichtdeterministische Turing-Maschinen und die Klasse $\mathcal{NP}$

Wir führen eine weitere wichtige Klasse von Sprachen beziehungsweise Entscheidungsproblemen ein. Zugrunde liegt dazu die nichtdeterministische Turing-Maschine (NTM). Bei der nichtdeterministischen Turing-Maschine wird die Übergangsfunktion δ zu einer Relation erweitert, die Wahlmöglichkeiten und ε–Übergänge (vergleiche endliche Automaten) ermöglicht. Wir betrachten hier ein äquivalentes Modell einer nichtdeterministischen Turing-Maschine, die auf einem **Orakel** basiert, und der Intuition näher kommt. Diese nichtdeterministische Turing-Maschine enthält zusätzlich zu der endlichen Kontrolle mit dem Lese-/Schreibkopf noch ein **Orakelmodul** mit einem eigenen Schreibkopf (siehe Abbildung 4.1).

Nichtdeterministische Turing-Maschinen haben wieder genau zwei Endzustände q_J und q_N, wobei q_J der akzeptierende Endzustand ist. Sie werden analog zu DTM durch das Oktupel $(Q, \Sigma, \sqcup, \Gamma, s, \delta, q_J, q_N)$ beschrieben.

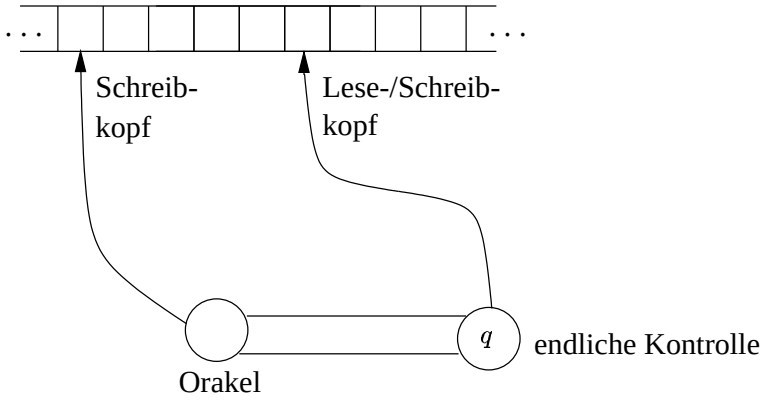

Abbildung 4.1: nichtdeterministische Turing-Maschine

Berechnung bei einer nichtdeterministischen Turing-Maschine

1. **Stufe:** Das Orakelmodul weist seinen Schreibkopf an, Schritt–für–Schritt entweder ein Symbol zu schreiben und nach links zu gehen oder anzuhalten. Falls der Schreibkopf anhält, wird das Orakelmodul inaktiv, und die endliche Zustandskontrolle wird aktiv.

2. **Stufe:** Ab diesem Zeitpunkt läuft die nichtdeterministische Turing-Maschine genauso ab wie eine deterministische Turing-Maschinen-Berechnung. Das Orakelmodul und sein Schreibkopf sind nicht weiter beteiligt.

Damit **akzeptiert** eine nichtdeterministische Turing-Maschine $\mathcal{M}$ ein Wort $x \in \Sigma^*$ genau dann, wenn es eine Berechnung gibt, die in q_J endet. $\mathcal{M}$ akzeptiert die Sprache $L \subseteq \Sigma^*$ genau dann, wenn sie gerade die Wörter aus L akzeptiert.

Übertragung auf Entscheidungsprobleme Π

Die Eingabe ist ein Wort aus Σ^*, zum Beispiel eine Kodierung eines Problembeispiels $I \in D_\Pi$.

1. **Stufe:** Es wird ein Orakel aus Γ^* berechnet, zum Beispiel ein „Lösungsbeispiel“ für I, also ein Indikator, ob $I \in J_\Pi$ oder $I \in N_\Pi$ gilt.

2. **Stufe:** Hier wird nun dieser Lösungsvorschlag überprüft, d.h. es wird geprüft ob $I \in J_\Pi$.

Beispiel TSP:

1. **Stufe:** Es wird zum Beispiel eine Permutation σ auf der Knotenmenge V vorgeschlagen. D.h. $(\sigma(1), \ldots, \sigma(n))$, $G = (V, E)$, c und k bilden die Eingabe.

2. **Stufe:** Es wird nun überprüft, ob $\sigma(V)$ eine Tour in G enthält, deren Länge bezüglich c nicht größer als k ist.

■

4.8 Definition
Die ***Zeitkomplexitätsfunktion*** $T_{\mathcal{M}}\colon \mathbb{Z}^+ \to \mathbb{Z}^+$ *einer nichtdeterministischen Turing-Maschine* $\mathcal{M}$ *ist definiert durch*

$$T_{\mathcal{M}}(n) := \max\left(\{1\} \cup \left\{ m \;\middle|\; \begin{array}{l} \text{es gibt ein } x \in L_{\mathcal{M}} \text{ mit } |x| = n, \text{ so daß} \\ \text{die minimale Anzahl von Schritten, die} \\ \mathcal{M} \text{ benötigt, um } x \text{ zu akzeptieren, } m \text{ ist} \end{array} \right\}\right)$$

D.h. zu $x \in L_{\mathcal{M}}$ *wird die kürzeste akzeptierende Berechnung für alle Problembeispiele der Länge* $|x|$ *betrachtet. Somit ergibt sich eine* worst–case *Abschätzung.*

Bemerkung:
Die Zeitkomplexität hängt nur von der Anzahl der Schritte ab, die bei einer akzeptierenden Berechnung auftreten. Per Konvention wird $T_{\mathcal{M}}(n) = 1$, falls es keine Eingabe x der Länge n gibt, die von $\mathcal{M}$ akzeptiert wird.

4.9 Definition
Die Klasse $\boldsymbol{\mathcal{NP}}$ *ist die Menge aller Sprachen* L*, für die es eine nichtdeterministische Turing-Maschine gibt, deren Zeitkomplexitätsfunktion polynomial beschränkt ist. (*$\mathcal{NP}$ *steht für* ***nichtdeterministisch polynomial****.)*

Entsprechung für Entscheidungsprobleme

Informell kann man sagen: Ein Entscheidungsproblem Π gehört zu $\mathcal{NP}$, falls

1. es für jedes Ja-Beispiel $I \in J_\Pi$ eine Struktur s gibt, mit deren Hilfe die Korrektheit der Antwort „Ja“ überprüft werden kann;

2. es einen Algorithmus gibt, der ein Problembeispiel $I \in D_\Pi$ und die zugehörige Struktur s als Input akzeptiert (annimmt) und in einer Laufzeit, die polynomial in der Kodierungslänge von I ist, überprüft, ob es eine Struktur ist, aufgrund deren Existenz die Antwort „Ja“ für I gegeben werden muß.

„Lasch“ ausgedrückt: Π gehört zu $\mathcal{NP}$, falls Π folgende Eigenschaft hat: Ist die Antwort bei Eingabe eines Beispiels I von Π „Ja“, dann kann die Korrektheit der Antwort in polynomialer Zeit überprüft werden.

Beispiel :
TSP$\in \mathcal{NP}$: Denn zu gegebenem $G = (V, E), c, k$ und einer festen Permutation σ auf V kann man in $O(|V| \cdot \log C)$, wobei C die größte vorkommende Zahl ist, überprüft werden, ob

$$\sum_{i=1}^{n-1} c(\{\sigma(i), \sigma(i+1)\}) + c(\{\sigma(n), \sigma(i)\}) \leq k$$

gilt. ■

4.3 $\mathcal{NP}$–vollständige Probleme

Trivialerweise gilt: $\mathcal{P} \subseteq \mathcal{NP}$.

Frage: Gilt $\mathcal{P} \subset \mathcal{NP}$ oder $\mathcal{P} = \mathcal{NP}$?

Die Vermutung ist, daß $\mathcal{P} \neq \mathcal{NP}$ gilt, d.h. daß es Probleme in $\mathcal{NP}$ gibt, die nicht in $\mathcal{P}$ sind. Dazu betrachten wir Probleme, die zu den „schwersten Problemen" in $\mathcal{NP}$ gehören. Dabei ist „am schwersten" im folgenden Sinne gemeint: wenn ein solches Problem trotzdem in $\mathcal{P}$ liegt, so kann man folgern, daß alle Probleme aus $\mathcal{NP}$ in $\mathcal{P}$ liegen, d.h. $\mathcal{P} = \mathcal{NP}$. Diese Probleme sind also Kandidaten, um $\mathcal{P}$ und $\mathcal{NP}$ zu trennen.

Es wird sich zeigen, daß alle diese „schwersten" Probleme im wesentlichen „gleich schwer" sind:

4.10 Definition
Eine ***polynomiale Transformation*** *einer Sprache $L_1 \subseteq \Sigma_1^*$ in eine Sprache $L_2 \subseteq \Sigma_2^*$ ist eine Funktion $f\colon \Sigma_1^* \to \Sigma_2^*$ mit den Eigenschaften:*

1. *es existiert eine polynomiale deterministische Turing-Maschine, die f berechnet;*
2. *für alle $x \in \Sigma_1^*$ gilt: $x \in L_1 \Leftrightarrow f(x) \in L_2$.*

Wir schreiben dann $L_1 \propto L_2$ (L_1 ist polynomial transformierbar in L_2).

4.11 Definition
Eine Sprache L heißt ***$\mathcal{NP}$–vollständig*** *, falls gilt:*

1. *$L \in \mathcal{NP}$ und*
2. *für alle $L' \in \mathcal{NP}$ gilt $L' \propto L$.*

Bemerkung:
Wir formulieren nun die Begriffe polynomial transformierbar und $\mathcal{NP}$-vollständig für Entscheidungsprobleme.

- Ein Entscheidungsproblem Π_1 ist **polynomial transformierbar** in das Entscheidungsproblem Π_2, wenn eine Funktion $f\colon D_{\Pi_1} \to D_{\Pi_2}$ existiert mit folgenden Eigenschaften:
  1. f ist durch einen polynomialen Algorithmus berechenbar;
  2. $\forall\, I \in D_{\Pi_1} : I \in J_{\Pi_1} \iff f(I) \in J_{\Pi_2}$.

  Wir schreiben dann $\Pi_1 \propto \Pi_2$.
- Ein Entscheindungsproblem Π heißt **$\mathcal{NP}$-vollständig**, falls gilt:
  1. $\Pi \in \mathcal{NP}$ und
  2. für alle $\Pi' \in \mathcal{NP}$ gilt $\Pi' \propto \Pi$.

4.12 Lemma
$\propto$ ist transitiv, d.h. aus $L_1 \propto L_2$ und $L_2 \propto L_3$ folgt $L_1 \propto L_3$.

Beweis: Die Hintereinanderausführung zweier polynomialer Transformationen ist wieder eine polynomiale Transformation. □

4.13 Korollar
Falls $L_1, L_2 \in \mathcal{NP}$, $L_1 \propto L_2$ und L_1 $\mathcal{NP}$-vollständig, dann ist auch L_2 $\mathcal{NP}$-vollständig.

Um also zu zeigen, daß ein Entscheidungsproblem Π $\mathcal{NP}$-vollständig ist, gehen wir folgendermaßen vor. Wir beweisen:

- $\Pi \in \mathcal{NP}$
- für ein „bekanntes“ $\mathcal{NP}$-vollständiges Problem Π' gilt: $\Pi' \propto \Pi$.

Zunächst müssen wir natürlich für ein erstes Problem zeigen, daß alle anderen Probleme aus $\mathcal{NP}$ sich auf dieses polynomial transformieren lassen. Das „erste“ $\mathcal{NP}$-vollständige Problem ist das **Erfüllbarkeitsproblem SAT** (satisfiability).

Definition von SAT

Sei $U = \{u_1, \ldots, u_m\}$ eine Menge von booleschen Variablen ($u_i, \overline{u_i}$ heißen Literale). Eine Wahrheitsbelegung für U ist eine Funktion $t\colon U \to \{\texttt{wahr}, \texttt{falsch}\}$. Eine **Klausel** ist ein Boole'scher Ausdruck der Form

$$y_1 \vee \ldots \vee y_s \quad \text{mit} \quad y_i \in \underbrace{\{u_1, \ldots, u_n\} \cup \{\overline{u_1}, \ldots, \overline{u_m}\}}_{\text{Literalmenge}} \cup \{\texttt{wahr}, \texttt{falsch}\}$$

Dann ist SAT wie folgt definiert:

Gegeben: Menge U von Variablen, Menge C von Klauseln über U

Frage: Existiert eine Wahrheitsbelegung von U, so daß C erfüllt wird, d.h. daß alle Klauseln aus C den Wahrheitswert `wahr` annehmen?

Beispiel :
$U = \{u_1, u_2\}$ mit $C = \{u_1 \vee \overline{u_2},\ \overline{u_1} \vee u_2\}$ ist Ja-Beispiel von SAT. Mit der Wahrheitsbelegung $t(u_1) = t(u_2) = \texttt{wahr}$ wird C erfüllt. ■

4.14 Satz (von Steven Cook 1971)
SAT ist $\mathcal{NP}$-vollständig.

Beweis:

1. SAT $\in \mathcal{NP}$ ist offensichtlich erfüllt, denn für ein Beispiel I von SAT (mit n Klauseln und m Variablen) und einer Wahrheitsbelegung t kann in $O(m + n)$ überprüft werden, ob t alle Klauseln erfüllt, d.h. ob I ein Ja–Beispiel ist.

2. Wir müssen zeigen, daß für jede Sprache $L \in \mathcal{NP}$ gilt: $L \propto L_{\mathrm{SAT}}$, wobei $L_{\mathrm{SAT}} = L[\mathrm{SAT}, s]$ für ein geeignetes Kodierungsschema s ist. Dazu muß für alle Sprachen $L \in \mathcal{NP}$ eine polynomiale Transformation f_L angegeben werden, für die gilt, daß für alle $x \in \Sigma^*$ (Σ Alphabet zu L) gilt

$$x \in L \Longleftrightarrow f_L(x) \in L_{\mathrm{SAT}}.$$

Wir benötigen eine Konstruktion von f_L, die zeigt, wie man die Überprüfung, ob eine nicht deterministische Turing-Maschine $\mathcal{M}$ zu L ein Wort $x \in \Sigma^*$ akzeptiert, durch die Angabe von Klauseln simulieren kann. Dazu benutzen wir, daß es eine nichtdeterministische Turing-Maschine $\mathcal{M}$ zu L gibt, die L in polynomialer Laufzeit erkennt.

$\mathcal{M}$ sei gegeben durch $(Q, \Sigma, \sqcup, \Gamma, q_0, \delta, q_J, q_N)$ und akzeptiere die Sprache $L = L_{\mathcal{M}}$ in der Laufzeit $T_{\mathcal{M}} \leq p(n)$, wobei p ein Polynom ist. O.B.d.A. gilt $p(n) \geq n$.

Bei einer akzeptierenden Berechnung von $\mathcal{M}$ für $x \in \Sigma^*$ ist die Anzahl der Berechnungsschritte von

- Orakelphase und
- Überprüfungsphase

jeweils beschränkt durch $p(n)$, wenn $|x| = n$ gilt. An einer so beschränkten Berechnung können höchstens die Zellen $-p(n)$ bis $p(n)+1$ des Bandes beteiligt sein, denn zu Anfang steht der Lese-/Schreibkopf der Überprüfungsphase an der Stelle 1. Der Zustand der deterministischen Stufe der Berechnung, d.h. der Überprüfungsphase, ist zu jedem Zeitpunkt eindeutig festgelegt, und zwar durch:

- den jeweiligen Bandinhalt dieser $-p(n)$ bis $p(n)+1$ Plätze,
- den Zustand der endlichen Kontrolle
- und der Position des Lese-/Schreibkopfs.

Dadurch ist es möglich, eine Berechnung vollständig durch eine begrenzte Anzahl von boole'schen Variablen und eine Belegung dieser Variablen zu beschreiben.

Bezeichne die Zustände aus Q durch $q_0, q_1 = q_J, q_2 = q_N, q_3, \ldots, q_r$ mit $|Q| = r+1$ und die Symbole aus Γ durch $s_0 = \sqcup, s_1, \ldots, s_\ell$ mit $|\Gamma| = \ell+1$. Es gibt drei Typen von Variablen in dem zugehörigen Problem SAT, die jeweils eine bestimmte Bedeutung haben:

Variable	Gültigkeitsbereich	Bedeutung
$Q[i,k]$	$0 \leq i \leq p(n)$ $0 \leq k \leq r$	„zum Zeitpunkt“ i der Über–prüfungsphase ist $\mathcal{M}$ in Zustand q_k
$H[i,j]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n)+1$	„zum Zeitpunkt“ i der Über–prüfungsphase ist der Lese–/ Schreibkopf an Position j des Bandes
$S[i,j,k]$	$0 \leq i \leq p(n)$ $-p(n) \leq j \leq p(n)+1$ $0 \leq k \leq \ell$	„zum Zeitpunkt“ i der Über–prüfungsphase ist der Bandinhalt an Position j das Symbol s_k

Eine Berechnung von $\mathcal{M}$ induziert in kanonischer Weise eine Wahrheitsbelegung dieser Variablen. Dabei legen wir „per Konvention“ fest, daß falls $\mathcal{M}$ vor dem Zeitpunkt $p(n)$ stoppt, die Berechnung weitergeht, aber „statisch“ bleibt; d.h. sie bleibt in demselben Zustand, an derselben Position und der Bandinhalt bleibt unverändert. Der Bandinhalt zum Zeitpunkt 0 der Überprüfungsphase sei: Eingabe x auf Platz 1 bis n und das „Orakel“ w auf Platz -1 bis $-|w|$; ansonsten Blanks. Umgekehrt muß eine beliebige Wahrheitsbelegung nicht notwendigerweise eine Berechnung induzieren. (zum Beispiel $Q[i,k] = Q[i,\ell]$ für $k \neq \ell$).

Die Transformation f_L bildet nun ein Problembeispiel x aus Π auf ein Problembeispiel von SAT ab, indem sie eine Menge von Klauseln über diesen Variablen konstruiert. Diese Klauselmenge wird genau dann durch eine Wahrheitsbelegung der Variablen erfüllt, wenn die Wahrheitsbelegung induziert wird durch eine akzeptierende Berechnung für die Eingabe x, deren Überprüfungsphase höchstens $p(n)$ Zeit benötigt, und deren Orakel höchstens Länge $p(n)$ hat. Damit können wir dann schließend:

$x \in L \Leftrightarrow$ es existiert eine akzeptierende Berechnung von $\mathcal{M}$ bei Eingabe x

$\Leftrightarrow$ es existiert eine akzeptierende Berechnung von $\mathcal{M}$ bei Eingabe x mit höchstens $p(n)$ Schritten in der Überprüfungsphase und einem Orakel w der Länge $|w| = p(n)$

$\Leftrightarrow$ es existiert eine erfüllende Wahrheitsbelegung für die Klauselmenge $f_L(x)$

Dann muß noch gezeigt werden, daß f_L polynomial berechenbar ist. Zunächst geben wir an, wie die Klauseln konstruiert werden.

Konstruktion der Klauseln: Die Bewegungsrichtung des Kopfes sei $d \in \{-1, 0, 1\}$. Es gibt sechs Klauselgruppen, die jeweils eine bestimmte Einschränkung der Wahrheitsbelegung bewirken:

Klauselgruppe	Einschränkung / Bedeutung
G_1	Zum Zeitpunkt i ist $\mathcal{M}$ in genau einem Zustand.
G_2	Zum Zeitpunkt i hat der Lese-/Schreibkopf genau eine Position.
G_3	Zum Zeitpunkt i enthält jede Bandstelle genau ein Symbol aus Γ.
G_4	Festlegung der Anfangskonfiguration zum Zeitpunkt 0.
G_5	Bis zum Zeitpunkt $p(n)$ hat $\mathcal{M}$ den Zustand q_J erreicht.
G_6	Zu jedem Zeitpunkt i folgt die Konfiguration von $\mathcal{M}$ zum Zeitpunkt $i+1$ aus einer einzigen Anwendung von δ aus der Konfiguration von $\mathcal{M}$ zum Zeitpunkt i.

Wenn die Klauselgruppen diese Einschränkungen bewirken, so korrespondiert eine erfüllende Wahrheitsbelegung gerade mit einer akzeptierenden Berechnung von x.

Konstruktion:

G_1: $Q[i,0] \vee \ldots \vee Q[i,r]$ für $0 \leq i \leq p(n)$,
(zu jedem Zeitpunkt i ist $\mathcal{M}$ in mindestens einem Zustand)
$\overline{Q[i,j]} \vee \overline{Q[i,j']}$ für $0 \leq i \leq p(n),\ 0 \leq j < j' \leq r$,
(zu jedem Zeitpunkt i ist $\mathcal{M}$ in nicht mehr als einem Zustand)

G_2: $H[i,-p(n)] \vee \ldots \vee H[i,p(n)+1]$ für $0 \leq i \leq p(n)$
$\overline{H[i,j]} \vee \overline{H[i,i']}$ für $0 \leq i \leq p(n)$ und $-p(n) \leq i < i' \leq p(n)+1$
(analog zu G_1)

G_3: $S[i,j,0] \vee S[i,j,1] \vee \ldots \vee S[i,j,\ell]$ für $0 \leq i \leq p(n)$
und $-p(n) \leq j \leq p(n)+1$
$\overline{S[i,j,k]} \vee \overline{S[i,j,k']}$ für $0 \leq i \leq p(n),\ -p(n) \leq j \leq p(n)$
und $0 \leq k < k' \leq \ell$
(analog zu G_1)

G_4: $Q[0,0]$, $H[0,1]$,
$S[0,0,0], S[0,1,k_1], \ldots, S[0,n,k_n]$, falls $x = s_{k_1} \ldots s_{k_n}$ ist
$S[0,n+1,0], \ldots, S[0,p(n)+1,0]$
$S[0,-1,\ell_1], \ldots, S[0,-|w|,\ell_{|w|}]$ für $w = s_{\ell_1} \ldots s_{\ell_{|w|}}$
$S[0,-|w|-1,0], \ldots, S[0,-p(n),0]$

G_5: $Q[p(n),1]$

G_6: besteht aus zwei Teilgruppen $G_{6,1}$ und $G_{6,2}$

$G_{6,1}$ bewirkt folgendes: falls $\mathcal{M}$ zum Zeitpunkt i an der Position j das Symbol s_k hat und der Lese-/Schreibkopf nicht an der Position j steht, dann hat $\mathcal{M}$ auch zum Zeitpunkt $i+1$ an Position j das Symbol s_k für $0 \leq i \leq p(n)$, $0 \leq k \leq \ell$ und $-p(n) \leq j \leq p(n)+1$:

$$\left(\left(S[i,j,k] \wedge \overline{H[i,j]}\right) \Longrightarrow S[i+1,j,k]\right)$$

Dies ergibt die Klausel

$$\left(\overline{S[i,j,k]} \vee H[i,j] \vee S[i+1,j,k]\right)$$

$G_{6,2}$ bewirkt, daß der Wechsel von einer Konfiguration zur nächsten tatsächlich δ entspricht. Sei $\delta(q_k, s_m) = (q_\kappa, s_\mu, d)$. Œ sei q_k aus $Q \backslash \{q_J, q_N\}$ sonst gilt $q_\kappa = q_k$, $s_\mu = s_m$ und $d = 0$. Falls $\mathcal{M}$ zum Zeitpunkt i den Lese-/Schreibkopf an Position j hat, im Zustand q_k ist und an der Stelle j das Zeichen s_m steht, dann ist zum Zeitpunkt $i+1$ der Lese-/Schreibkopf an Position $j+d$, an Position j steht s_μ und $\mathcal{M}$ geht in den Zustand q_κ über:

$$\begin{aligned} &(H[i,j] \wedge Q[i,k] \wedge S[i,j,m]) \Rightarrow H[i+1,j+d] \\ \text{und } &(H[i,j] \wedge Q[i,k] \wedge S[i,j,m]) \Rightarrow Q[i+1,\kappa] \\ \text{und } &(H[i,j] \wedge Q[i,k] \wedge S[i,j,m]) \Rightarrow S[i+1,j,\mu] \end{aligned}$$

Dies ergibt folgende Klauseln

$$\overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,m]} \vee H[i+1, j+d]$$
$$\overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,m]} \vee Q[i+1, \kappa]$$
$$\overline{H[i,j]} \vee \overline{Q[i,k]} \vee \overline{S[i,j,m]} \vee S[i+1, j, \mu]$$
$$\text{für } 0 \leq i \leq p(n),\ -p(n) \leq j \leq p(n)+1,\ 0 \leq k \leq r,\ 0 \leq m \leq l$$

Durch f_L wird nun ein Input $(\mathcal{M}, x)$ auf die Klauselmenge

$$G := G_1 \wedge G_2 \wedge \ldots \wedge G_6$$

abgebildet. Wenn $x \in L$, dann ist G erfüllbar. Andererseits induziert eine erfüllende Wahrheitsbelegung der Variablen aus G eine akzeptierende Berechnung von $\mathcal{M}$ für die Eingabe $x \in$ L.

Es bleibt zu zeigen, daß $G_1, \ldots, G_6$ in polynomialer Zeit (in $|x| = n$) gebildet werden können. Dazu müsssen wir die Größe der Klauselgruppen, also die Anzahl der Literale, abschätzen.

G_1: $(p(n)+1)(r+1) + (p(n)+1)r(r+1)$

G_2: $(p(n)+1)(2p(n)+1) + (2p(n)(2p(n)+1))$

G_3: $(p(n)+1)(2p(n)+1)(\ell+1) + (p(n)+1)(2p(n)+1)(\ell+1)$

G_4: $n + 2 + p(n) - n + p(n)$

G_5: 1

G_6: $\underbrace{2(p(n)+1)^2(\ell+1)\cdot 3}_{G_{6,1}} + \underbrace{6\cdot(p(n)(p(n)+1)\cdot(r+1)(\ell+1)\cdot 4}_{G_{6,2}}$

r und ℓ sind Konstanten, die durch $\mathcal{M}$ (und damit durch L) induziert werden, $p(n)$ ist ein Polynom in n. Also sind alle Größen polynomial in n. Die angegebene Funktion f_L ist also eine polynomiale Transformation von L nach L_{SAT}.

□

Wir betrachten nun eine eingeschränkte Version des Erfüllbarkeitsproblems, die uns später dabei hilfreich sein wird, die $\mathcal{NP}$–Vollständigkeit weiterer Probleme zu zeigen.

Problem 3SAT

Gegeben: Menge U von Variablen, Menge C von Klauseln über U , wobei jede Klausel genau *drei* Literale enthält.

Frage: Existiert eine erfüllende Wahrheitsbelegung für C?

4.15 Satz

Das Problem 3SAT ist $\mathcal{NP}$-vollständig.

Beweis:

1. 3SAT$\in \mathcal{NP}$, denn für eine feste Wahrheitsbelegung t kann in polynomialer Zeit ($O(n)$, falls $|C| =: n$) überprüft werden, ob t alle Klauseln aus C erfüllt.

2. Wir zeigen SAT $\propto$ 3SAT durch Angabe einer polynomialen Transformation, die jeder Instanz von SAT eine Instanz von 3SAT zuordnet, und für die Ja-Beispiele von 3SAT genau die Bilder von Ja-Beispielen von SAT sind.

   Seien $u_1, \ldots, u_m, \overline{u_1}, \ldots, \overline{u_m}$ die Literale und $c_1, \ldots, c_n$ die Eingabeklauseln für SAT, also $|C| = n, |U| = m$. Die polynomiale Transformation von SAT nach 3SAT wird wie folgt konstruiert:

   - Besteht die Klausel c aus einem Literal x so wird c auf $x \vee x \vee x$ abgebildet.
   - Besteht die Klausel c aus zwei Literalen $x \vee y$ so wird c auf $x \vee y \vee x$ abgebildet.
   - Klauseln mit drei Literalen werden auf sich selbst abgebildet.
   - Enthält eine Klausel $k > 3$ Literale, wird sie durch $k-2$ neue Klauseln mit jeweils genau drei Literalen ersetzt. Dazu werden für jede solche Klausel $k-3$ neue Variablen eingeführt. Sei also $c = x_1 \vee \ldots \vee x_k$ mit $k > 3$. Die neuen Variablen seien $y_{c,1}, \ldots, y_{c,k-3}$. Dann wird c ersetzt durch die folgenden $k-2$ Klauseln:

$$\begin{aligned} x_1 \vee x_2 &\vee y_{c,1} \\ \overline{y_{c,1}} \vee x_3 &\vee y_{c,2} \\ &\vdots \\ \overline{y_{c,k-4}} \vee x_{k-2} &\vee y_{c,k-3} \\ \overline{y_{c,k-3}} \vee x_{k-1} &\vee x_k \end{aligned}$$

   Diese Klauseln lassen sich in polynomialer Zeit konstruieren ($\mathcal{O}(|C| \cdot |U|) = \mathcal{O}(nm)$). Wir müssen noch zeigen, daß die Klauselmenge von SAT genau dann erfüllbar ist, wenn die so konstruierte Klauselmenge von 3SAT erfüllbar ist.

   Sei zunächst die Klauselmenge zu SAT erfüllbar. Dann wird in jeder Klausel $c = x_1 \vee \ldots \vee x_k$ mindestens ein x_i `wahr` gesetzt von einer erfüllenden Wahrheitsbelegung. Der Fall für $k \leq 3$ ist damit klar. Sei nun $k > 3$.

   (a) Falls $x_1 =$ `wahr` oder $x_2 =$ `wahr` gesetzt wird, so erfüllt die Erweiterung dieser Wahrheitsbelegung, die alle $y_{c,j}$ `falsch` setzt, alle neuen Klauseln zu C.

   (b) Falls $x_i =$ `wahr` ist für $i > 2$, so erfüllt die Erweiterung

$$y_{c,j} = \begin{cases} \texttt{wahr} & \text{falls } 1 \leq j \leq i-2 \\ \texttt{falsch} & \text{falls } i-1 \leq j \leq k-3 \end{cases}$$

   alle Klauseln zu C.

   Um die Umkehrung zu zeigen (d.h. SAT-Instanz ist erfüllbar, falls die dazu konstruierte 3SAT–Instanz erfüllbar ist), beweisen wir, daß, wenn die SAT-Instanz nicht erfüllbar ist, so ist die 3SAT-Instanz auch nicht erfüllbar. Falls die SAT-Instanz nicht erfüllbar ist, so existiert für jede

Wahrheitsbelegung eine Klausel $C = x_1 \vee \ldots \vee x_k$ bei der alle x_i auf `falsch` gesetzt sind. Um die zugehörige 3SAT-Instanz zu erfüllen, müßten alle $y_{c,j}$ $(1 \leq j \leq k-3)$ `wahr` gesetzt werden. Dann ist allerdings die letzte Klausel $\overline{y_{c_{k-3}}} \vee x_{k-1} \vee x_k$ nicht erfüllt. Also ist auch die 3SAT–Instanz nicht erfüllbar.

□

Wir betrachten nun einige weiter Probleme.

4.16 Definition
*Eine **Clique** ist eine Menge von Knoten, deren knoteninduzierter Subgraph vollständig ist.*

Problem CLIQUE

Gegeben: Graph $G = (V, E)$ und ein Parameter $K \leq |V|$

Frage: Gibt es in G eine Clique der Größe mindestens K?

4.17 Satz
Das CLIQUE–Problem ist $\mathcal{NP}$–vollständig.

Beweis:

1. CLIQUE $\in \mathcal{NP}$: Übung.

2. Wir zeigen die $\mathcal{NP}$–Vollständigkeit durch Angabe einer Transformation von 3SAT zu CLIQUE, also 3SAT $\propto$ CLIQUE. Sei $C = \{c_1, \ldots, c_n\}$ mit $c_i = x_{i1} \vee x_{i2} \vee x_{i3}$ und $x_{ij} \in \{u_1, \ldots, u_m, \overline{u_1}, \ldots, \overline{u_m}\}$ ein Problembeispiel für 3SAT. Wir transformieren dieses Problembeispiel in einen Graphen $G = (V, E)$ und $K \in \mathbb{Z}^+$ für CLIQUE wie folgt: V enthält $3n$ Knoten v_{ij} für $1 \leq i \leq n,\ 1 \leq j \leq 3$, entsprechend allen Vorkommen von Literalen.

   v_{ij} und $v_{k\ell}$ sind durch Kanten aus E verbunden genau dann, wenn:

   (a) $i \neq k$, d.h. wenn die entsprechenden Literale in verschiedenen Klauseln vorkommen

   (b) $x_{ij} \neq \overline{x_{kl}}$, d.h. wenn die entsprechenden Literale gleichzeitig erfüllbar sind.

   Außerdem wird $K = n$ gesetzt.

   Beispiel:
   Sei $C = \{u_1 \vee u_2 \vee \overline{u_3},\ u_1 \vee \overline{u_2} \vee u_3,\ \overline{u_1} \vee u_2 \vee \overline{u_3}\}$. Dann entspricht

Knotennummer	v_{11}	v_{12}	v_{13}	v_{21}	v_{22}	v_{23}	v_{31}	v_{32}	v_{33}
Literal	u_1	u_2	$\overline{u_3}$	u_1	$\overline{u_2}$	u_3	$\overline{u_1}$	u_2	$\overline{u_3}$.

   So ergibt sich G wie in Abbildung 4.2.

■

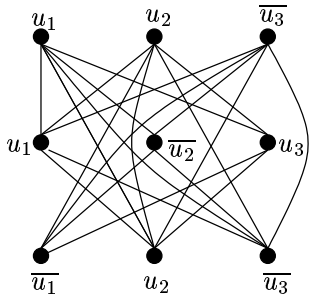

Abbildung 4.2: Graph G aus 3SAT$\propto$CLIQUE

Wenn das Problembeispiel von 3SAT erfüllbar ist, kann in jeder der n Klauseln mindestens ein Literal mit `wahr` belegt werden. D.h. n verschiedene Vorkommen von Literalen können gleichzeitig erfüllt werden, und damit bilden die zugehörigen n Knoten aus G eine Clique.

Gibt es umgekehrt in G eine Clique der Größe mindestens n, so existieren n verschiedene Vorkommen von Literalen, die gleichzeitig erfüllbar sind und die wegen (a) in verschiedenen Klauseln liegen. Also ist die gesamte Klauselmenge erfüllbar. □

Problem 2SAT
Gegeben: Menge U von Variablen, Menge C von Klauseln über U, wobei jede Klausel genau *zwei* Literale enthält.

Frage: Existiert eine erfüllende Wahrheitsbelegung für C?

Im Gegensatz zu 3SAT kann man zu 2SAT direkt einen polynomialen Algorithmus angegeben. Also ist 2SAT$\in \mathcal{P}$. (ohne Beweis — vgl. Übungsblatt)

Problem Max2SAT
Gegeben: Menge U von Variablen, Menge C von Klauseln über U, wobei jede Klausel genau *zwei* Literale enthält und eine Zahl $K \in \mathbb{N}$.

Frage: Existiert eine Wahrheitsbelegung, die mindestens K Klauseln erfüllt?

Im Gegensatz zu 2SAT ist Max2SAT wieder $\mathcal{NP}$-vollständig (Beweis durch Transformation von 3SAT — vgl. Übungsblatt).

Ein weiteres $\mathcal{NP}$–vollständiges Problem ist das folgende Problem:

Problem Subgraphisomorphie
Gegeben: Graphen $G = (V, E)$ und $H = (V', E')$ mit $|V'| < |V|$

Frage: Gibt es eine Menge $U \subseteq V$ mit $|U| = |V'|$ und eine bijektive Abbildung Iso: $V' \to U$, so daß für alle $x, y \in V'$ gilt:

$$\{x, y\} \in E' \Longleftrightarrow \{\text{Iso}(x), \text{Iso}(y)\} \in E$$

Die Frage ist also, ob H isomorph zu einem Subgraphen von G ist. Es gilt Subgraphisomorphie ist $\mathcal{NP}$-vollständig (ohne Beweis).

Wir haben nun gesehen, daß alle $\mathcal{NP}$–vollständigen Probleme im wesentlichen gleich „schwer“ sind, da es immer eine polynomiale Transformation von einem zum anderen Problem gibt. Dies hat aber auch Auswirkungen auf die Frage, ob $\mathcal{P} = \mathcal{NP}$ ist.

4.18 Lemma
Sei L $\mathcal{NP}$-vollständig, dann gilt:

1. $L \in \mathcal{P} \Longrightarrow \mathcal{P} = \mathcal{NP}$
2. $L \notin \mathcal{P}$, *so gilt für alle $\mathcal{NP}$-vollständigen Sprachen L', daß $L' \notin \mathcal{P}$ gilt.*

Beweis:

1. Wenn $L \in \mathcal{P}$ ist, so existiert eine polynomiale deterministische Turing-Maschine für L. Dann liefert die Hintereinanderausführung der polynomialen Transformation zu $L' \propto L$ und dieser polynomialen Berechnung für L wieder eine polynomiale deterministische Turing-Maschinen-Berechnung für L'. Damit ist für alle $L' \in \mathcal{NP}$ auch $L' \in \mathcal{P}$.
2. Sei $L \notin \mathcal{P}$, aber angenommen für eine $\mathcal{NP}$-vollständige Sprache L' gilt: $L' \in \mathcal{P}$, so folgt aus (1) $\mathcal{P} = \mathcal{NP}$. Dies ist aber ein Widerspruch zur Voraussetzung $L \notin \mathcal{P}$, da dann $\mathcal{NP} \backslash \mathcal{P} \neq \emptyset$.

□

4.4 Komplementsprachen

Wir betrachten nun weitere Sprachklassen, die im Zusammenhang mit den Klassen $\mathcal{P}$ und $\mathcal{NP}$ auftreten.

4.19 Definition
Die Klasse $\boldsymbol{\mathcal{NPC}}$ sei die Klasse der $\mathcal{NP}$-vollständigen Sprachen/Probleme. ($\mathcal{NP}$-complete). Die Klasse $\boldsymbol{\mathcal{NPI}}$ ist definiert durch $\mathcal{NPI} := \mathcal{NP} \backslash (\mathcal{P} \cup \mathcal{NPC})$. Die Klasse $\boldsymbol{co - \mathcal{P}}$ ist die Klasse aller Sprachen $\Sigma^* \backslash L$ für $L \subseteq \Sigma^*$ und $L \in \mathcal{P}$ (die Klasse der Komplementsprachen). Die Klasse $\boldsymbol{co - \mathcal{NP}}$ ist die Klasse aller Sprachen $\Sigma^* \backslash L$ für $L \subseteq \Sigma^*$ und $L \in \mathcal{NP}$.

4.20 Satz (Ladner (1975))
Falls $\mathcal{P} \neq \mathcal{NP}$, so folgt $\mathcal{NPI} \neq \emptyset$.

Es liegt vermutlich eine Situation wie in Abbildung 4.3 vor. Es ist $\mathcal{P} = \text{co}-\mathcal{P}$, da man für eine Sprache L^c zu $L \in \mathcal{P}$ nur die Endzustände q_J und q_N in der entsprechenden deterministischen Turing-Maschinen-Berechnung vertauschen muss.

Frage: Gilt auch $\mathcal{NP} = \text{co} - \mathcal{NP}$?

Eine nichtdeterministische Berechnung muß noch nicht einmal enden. Natürlich folgt aus $\mathcal{NP} \neq \text{co}-\mathcal{NP}$, daß $\mathcal{P} \neq \mathcal{NP}$ gilt. Aber was folgt aus $\mathcal{NP} = \text{co}-\mathcal{NP}$? Vermutlich ist $\mathcal{NP} \neq \text{co} - \mathcal{NP}$ (Verschärfung der „$\mathcal{P} \neq \mathcal{NP}$“–Vermutung).

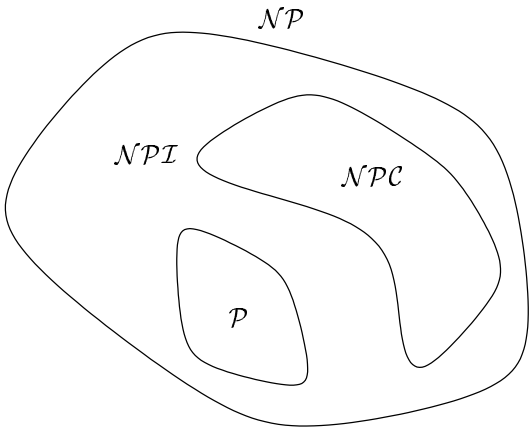

Abbildung 4.3: Komplexitätsklassen

Ein Beispiel für ein Problem in $\text{co} - \mathcal{NP}$ ist das TSP–Komplement–Problem:

Problem co–TSP
Gegeben: Graph $G = (V, E)$, c: $E \to \mathbb{Z}^+$ und ein Parameter K.

Frage: Gibt es *keine* Tour der Länge $\leq K$?

Es ist klar, daß co–TSP in $\text{co} - \mathcal{NP}$ liegt, da TSP in $\mathcal{NP}$ liegt. Bei TSP ist es leicht nachzuweisen, ob eine Instanz ein „Ja"–Beispiel ist, wenn eine geeignete Tour bekannt ist. Für co–TSP ist dies jedoch nicht so leicht, auch wenn eine Tour gegeben ist. Die Frage, ob co–TSP auch in $\mathcal{NP}$ liegt, ist daher nicht so leicht zu beantworten. Die Vermutung ist „nein".

4.21 Lemma
Falls L $\mathcal{NP}$-vollständig ist und $L \in co - \mathcal{NP}$, so ist $\mathcal{NP} = co - \mathcal{NP}$.

Beweis: Sei $L \in \text{co} - \mathcal{NP}$, dann existiert eine polynomiale nichtdeterministische Berechnung für L^c. Da für alle $L' \in \mathcal{NP}$ gilt: $L' \propto L$, so existiert auch eine deterministische polynomiale Transformation $L'^c \propto L^c$. Also existiert eine polynomiale nichtdeterministische Berechnung für L'^c, also $L' \in \text{co} - \mathcal{NP}$. □

Bemerkung:
Mit der Vermutung $\mathcal{NP} \neq \mathcal{P}$ folgt auch $\mathcal{NPC} \cap \text{co} - \mathcal{NP} = \emptyset$. Wenn ein Problem in $\mathcal{NP}$ und $\text{co} - \mathcal{NP}$ ist, vermutlich aber nicht in $\mathcal{P}$, so ist es in $\mathcal{NPI}$.

Ein Kandidat für ein Problem aus $\mathcal{NPI}$ ist die Graphenisomorphie:

Problem Graphenisomorphie
Gegeben: Graphen $G = (V, E)$ und $H = (V', E')$ mit $|V| = |V'|$.

Frage: Sind G und H isomorph, d.h. existiert eine bijektive Abbildung

$$\text{Iso}: V' \to V \text{ mit } \quad \{x, y\} \in E' \iff \{\text{Iso}(x), \text{Iso}(y)\} \in E?$$

Für Graphenisomorphie weiss man, daß es sowohl in $\mathcal{NP}$ als auch in $\text{co} - \mathcal{NP}$ liegt.

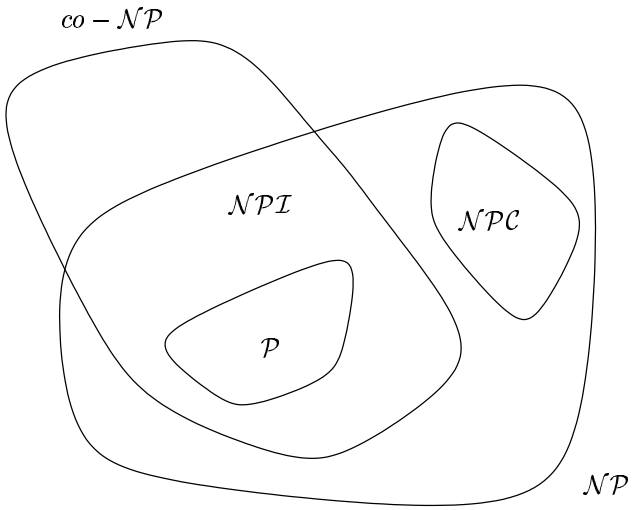

Abbildung 4.4: Komplexitätsklassen

4.5 Weitere Komplexitätsklassen über $\mathcal{NP}$ hinaus

Wir betrachten nun Probleme, die allein aufgrund der Problemformulierung nicht in $\mathcal{NP}$ liegen, zum Beispiel Optimierungs-, Such- und Aufzählungsprobleme.

4.22 Definition
Ein ***Suchproblem*** Π *wird beschrieben durch*

1. *die Menge der Problembeispiele* D_Π *und*
2. *für* $I \in D_\Pi$ *die Menge* $S_\Pi(I)$ aller *Lösungen von* I.

Die „Lösung" eines Suchproblems besteht in der Angabe einer Lösung aus $S_\Pi(I)$ *für ein Problembeispiel* $I \in D_\Pi$ *mit* $S_\Pi(I) \neq \emptyset$ *und „Nein" sonst.*

Problem TSP–Suchproblem
Gegeben: Graph $G = (V, E)$ vollständig und gewichtet mit $c\colon E \to \mathbb{R}$.

Frage: Gib eine optimale Tour zu G bezüglich c an.

$S_\Pi(I)$ ist die Menge aller optimalen Touren zu I. Die Angabe einer optimalen Tour löst also das Problem.

Problem Hamilton–Kreis (Suchproblem)
Gegeben: Ein ungerichteter, ungewichteter Graph $G = (V, E)$.

Frage: Gib einen Hamilton–Kreis in G an, falls einer existiert. Ein Hamilton–Kreis ist dabei eine Permutation π auf V, so daß

$$\{\pi(n), \pi(1)\} \in E \text{ und } \{\pi(i), \pi(i+1)\} \in E \text{ für } 1 \leq i \leq n-1 \text{ ist.}$$

4.23 Definition
Ein ***Aufzählungsproblem*** Π *ist gegeben durch*

1. *die Menge der Problembeispiele* D_Π *und*
2. *für* $I \in D_\Pi$ *die Menge* $S_\Pi(I)$ *aller Lösungen von* I.

Die Lösung *eines Aufzählungsproblem* Π *besteht in der Angabe der Kardinalität von* $S_\Pi(I)$, *d.h.* $|S_\Pi(I)|$.

Problem Hamilton–Kreis (Aufzählungsproblem)
Gegeben: ein ungerichteter, ungewichteter Graph $G = (V, E)$.

Frage: Wieviele Hamilton–Kreise gibt es in G?

Um Suchprobleme oder Aufzählungsprobleme untereinander oder im Vergleich zu $\mathcal{NP}$-vollständigen Problemen bezüglich ihrer Komplexität zu vergleichen, benötigt man ein Konzept dafür, wann ein Problem Π mindestens so schwer ist, wie ein Problem Π'. Wir verwenden dazu, analog zur polynomialen Transformation bei Entscheidungsproblemen, das Konzept der Turingreduzierbarkeit. Die entsprechenden Sprachklassen heißen:

- „stark-$\mathcal{NP}$-vollständig“ bei Suchproblemen
- „Anzahl-$\mathcal{P}$-vollständig“ bei Aufzählungsproblemen

Wir betrachten hier Suchprobleme genauer. Dazu assoziieren wir mit einem Suchproblem Π folgende Relation:

$$R_\Pi = \{(x, s) \mid x \in D_\Pi, s \in S_\Pi(x)\}$$

4.24 Definition
Eine Funktion $f\colon \Sigma^* \to \Sigma^*$ ***realisiert*** *eine Relation* R, *wenn für alle* $x \in \Sigma^*$ *gilt:*

$$f(x) = \begin{cases} y & \text{falls es ein } y \in \Sigma^* \backslash \{\varepsilon\} \text{ gibt mit } (x, y) \in R \\ \varepsilon & \text{sonst} \end{cases}$$

Ein Algorithmus ***löst*** *das durch* R_Π *beschriebene Suchproblem* Π, *wenn er eine Funktion berechnet, die* R_Π *realisiert.*

Der Begriff der „Turing-Reduzierbarkeit“ ist über die **Orakel-Turing-Maschine** definiert.

4.25 Definition
Eine ***Orakel-Turing-Maschine*** *zum Orakel* $G\colon \Sigma^* \to \Sigma^*$ *ist eine deterministische Turing-Maschine mit einem ausgezeichnetem* ***Orakelband*** *und zwei zusätzlichen Zuständen* q_f *und* q_a. *Dabei ist* q_f *der* ***Fragezustand*** *und* q_a *der* ***Antwortzustand*** *des Orakels. Die Arbeitsweise ist in allen Zuständen* $q \neq q_f$ *wie bei der normalen Turing-Maschine. Wenn der Zustand* q_f *angenommen wird, der Kopf sich auf Position* i *der Orakelbandes befindet und der Inhalt des Orakelbandes auf Position* $1, \ldots, i$ *das Wort* $y = y_1 \ldots y_i$ *ist, so ist der Übergang:*

1. *Fehlermeldung, falls* $y \notin \Sigma^*$
2. *in einem Schritt wird* y *auf dem Orakelband gelöscht und* $g(y)$ *auf die Positionen* $1, \ldots, |g(y)|$ *des Orakelbandes geschrieben. Der Kopf des Orakelbandes springt auf Position 1 und der Folgezustand ist* q_a.

4.26 Definition
Seien R, R' *Relationen über* Σ^*. *Eine* ***Turing-Reduktion*** $\propto_T$ *von* R *auf* R' *(*$R \propto_T R'$*), ist eine Orakel-Turing-Maschine* $\mathcal{M}$, *deren Orakel die Relation* R' *realisiert und die selber in polynomialer Zeit die Funktion* f *berechnet, die* R *realisiert.*

Bemerkung:

- Falls R' in polynomialer Zeit realisierbar ist und $R \propto_T R'$, so ist auch R in polynomialer Zeit realisierbar.
- Falls $R \propto_T R'$ und $R' \propto_T R''$ so auch $R \propto_T R''$.

4.27 Definition
Ein Suchproblem Π *heißt* ***$\mathcal{NP}$–schwer*** , *falls es eine* $\mathcal{NP}$*-vollständige Sprache* L *gibt mit* $L \propto_T \Pi$.

Bemerkung:
Ein Problem das $\mathcal{NP}$–schwer ist, muß nicht notwendigerweise in $\mathcal{NP}$ sein.

Wir nennen ein Problem $\mathcal{NP}$–schwer, wenn es mindestens so schwer ist, wie alle $\mathcal{NP}$–vollständigen Probleme. Darunter fallen auch

- Optimierungsprobleme für die das zugehörige Entscheidungsproblem $\mathcal{NP}$–vollständig ist.
- Entscheidungsprobleme Π für die gilt, daß für alle Probleme $\Pi' \in \mathcal{NP}$ gilt $\Pi' \propto \Pi$, aber für die nicht klar ist, ob $\Pi \in \mathcal{NP}$.

Klar ist, daß ein $\mathcal{NP}$–vollständiges Problem auch $\mathcal{NP}$–schwer ist.

4.28 Lemma
Falls L $\mathcal{NP}$*–schwer ist, so ist auch* L^c $\mathcal{NP}$*–schwer. D.h. die Klasse der* $\mathcal{NP}$*–schweren Probleme ist bezüglich Komplementbildung abgeschlossen.*

Beweis: Es gilt: $L \propto_T L^c$, denn man kann L mit einem L^c-Orakel realisieren, indem man die Eingabe auf das Orakelband kopiert und die Antwort des Orakels negiert. Da L $\mathcal{NP}$–schwer ist, gibt es mindestens eine $\mathcal{NP}$-vollständige Sprache L' mit $L' \propto_T L$. Damit folgt $L' \propto_T L^c$. □

Bisher haben wir Komplexität nur bezüglich der Laufzeit betrachtet. Interessant ist aber auch der Speicherplatz. Betrachte dazu eine Turing-Maschine mit drei Bändern (Input–, Arbeits– und Output–Band). Gemessen wird die maximale Anzahl an Positionen des **Arbeitsbandes**, die während der Berechnung benötigt wird.

- Die Klasse der Probleme, die mit polynomialem Speicherplatz gelöst werden können heißt $\mathcal{PSPACE}$. Es gilt:

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PSPACE}$$

- Die Klasse der Probleme, die mit polylogarithmischem Speicherplatz gelöst werden können heißt $\mathcal{SC}$ (Steven's Class, nach Steven Cook).

4.6 Pseudopolynomiale Algorithmen

Es gibt $\mathcal{NP}$–vollständige Probleme, die von einer deterministischen Turing-Maschine mit einer Laufzeit, die polynomial in der Inputlänge ist, gelöst werden können, wenn man die Eingabe unär anstatt zum Beispiel binär kodiert. Dann gehen „Zahlen" nicht logarithmisch sondern direkt in die Inputlänge ein, d.h. die Inputlänge ist also größer. Dies ist natürlich nur für Probleme relevant, in denen überhaupt Zahlen vorkommen, wie zum Beispiel beim Traveling Salesman Problem (die Kostenfunktion c). Einen solchen Algorithmus, der polynomial in der Inputlänge bei Unärkodierung ist, nennt man **pseudopolynomialen** Algorithmus.

Es gilt allerdings: Falls $\mathcal{P} \neq \mathcal{NP}$ ist, so gibt es für TSP keinen pseudopolynomialen Lösungsalgorithmus. TSP wird daher als **stark $\mathcal{NP}$–vollständig** bezeichnet.

4.7 Approximationsalgorithmen für Optimierungsprobleme

Für Optimierungsprobleme, für die das zugehörige Entscheidungsproblem $\mathcal{NP}$–vollständig ist, kann man versuchen, polynomiale Algorithmen anzugeben, die zwar keine Optimallösung liefern, aber immerhin eine „beweisbar gute Lösung".

Wie wird die Güte einer Lösung gemessen?

Wir betrachten die Güte einer Lösung, die ein Algorithmus im *worst-case* für ein Problem Π liefert, und zwar im Vergleich zur Optimallösung.

Bezeichne $\mathbf{OPT}(\boldsymbol{I})$ für $I \in D_\Pi$ den Wert der (beziehungsweise einer) Optimallösung. Zu einem Algorithmus $\mathcal{A}$ zur Lösung von Π bezeichnet $\boldsymbol{\mathcal{A}(I)}$ den Wert der Lösung, die $\mathcal{A}$ bei Eingabe $I \in D_\Pi$ liefert.

4.7.1 Approximation mit Differenzengarantie, absolute Approximation

4.29 Definition
Sei Π ein Optimierungsproblem. Ein polynomialer Algorithmus $\mathcal{A}$, der für jedes $I \in D_\Pi$ einen Wert $\mathcal{A}(I)$ liefert, mit

$$|\,\mathrm{OPT}(I) - \mathcal{A}(I)| \leq K$$

und $K \in \mathbb{N}_0$ konstant, heißt ***Approximationsalgorithmus mit Differenzengarantie*** *oder* ***absoluter Approximationsalgorithmus****.*

Es gibt nur wenige $\mathcal{NP}$–schwere Optimierungsprobleme, für die ein absoluter Approximationsalgorithmus existiert, aber viele „Negativ–Resultate".

Das Rucksack–Problem, KNAPSACK

Problem

Gegeben: Eine Menge von „Teilen" $M = \{1, \ldots, n\}$, Kosten $c_1, \ldots, c_n \in \mathbb{N}_0$ und Gewichten $w_1, \ldots, w_n \in \mathbb{N}$ sowie ein Gesamtgewicht $W \in \mathbb{N}$.

Frage: Gib $x_1, \ldots, x_n \in \mathbb{N}_0$ an, so daß

$$\sum_{i=0}^{n} x_i w_i \leq W \quad \text{und} \quad \sum_{i=1}^{n} x_i c_i \text{ maximal ist.}$$

Das KNAPSACK–Problem ist $\mathcal{NP}$–schwer und es läßt sich (vermutlich) kein absoluter Approximationsalgorithmus angeben:

4.30 Satz
Falls $\mathcal{P} \neq \mathcal{NP}$, so gibt es keinen absoluten Approximationsalgorithmus $\mathcal{A}$ für KNAPSACK.

Beweis: Wir zeigen, daß aus einem absoluten Approximationsalgorithmus für KNAPSACK auch ein optimaler polynomialer Algorithmus für KNAPSACK abgeleitet werden kann, im Widerspruch zu $\mathcal{P} \neq \mathcal{NP}$.

Sei $\mathcal{A}$ ein absoluter Approximationsalgorithmus mit $|\operatorname{OPT}(I) - \mathcal{A}(I)| \leq K$ für alle I. Betrachte nun zum Problembeispiel I für KNAPSACK mit M, w_i, c_i, W ein Problembeispiel I' mit

$$M' := M, w_i' := w_i, W' := W \text{ und } c_i' := c_i \cdot (K+1).$$

Damit ist $\operatorname{OPT}(I') = (K+1)\operatorname{OPT}(I)$. Dann liefert $\mathcal{A}$ zu I' eine Lösung $x_1, \ldots, x_k$ mit Wert $\sum_{i=1}^{k} x_i c_i' = \mathcal{A}(I')$, für den gilt:

$$|\operatorname{OPT}(I') - \mathcal{A}(I')| \leq K.$$

$\mathcal{A}(I')$ induziert damit eine Lösung $x_1, \ldots, x_k$ für I mit dem Wert

$$\mathcal{L}(I) := \sum_{i=1}^{k} x_i c_i,$$

für den gilt:

$$|(K+1)\operatorname{OPT}(I) - (K+1)\mathcal{L}(I)| \leq K$$

also $|\operatorname{OPT}(I) - \mathcal{L}(I)| \leq \frac{K}{K+1} < 1$. Da $\operatorname{OPT}(I)$ und $\mathcal{L}(I) \in \mathbb{N}_0$ für alle I, ist also $\operatorname{OPT}(I) = L(I)$. Der entsprechende Algorithmus ist natürlich polynomial und liefert einen Optimalwert für das KNAPSACK–Problem. Dies steht im Widerspruch zu $\mathcal{P} \neq \mathcal{NP}$. □

4.31 Satz
Falls $\mathcal{P} \neq \mathcal{NP}$, so gibt es keinen absoluten Approximationsalgorithmus $\mathcal{A}$ für CLIQUE.

Beweis: Übung. □

4.7.2 Approximation mit relativer Gütegarantie

4.32 Definition
Sei Π ein Optimierungsproblem. Ein polynomialer Algorithmus $\mathcal{A}$, der für jedes $I \in D_\Pi$ einen Wert $\mathcal{A}(I)$ liefert mit $R_\mathcal{A}(I) \leq K$, wobei $K \geq 1$ eine Konstante und,

$$\mathcal{R}_\mathcal{A}(I) := \begin{cases} \frac{\mathcal{A}(I)}{\mathrm{OPT}(I)} & \text{falls } \Pi \text{ Minimierungsproblem} \\ \frac{\mathrm{OPT}(I)}{\mathcal{A}(I)} & \text{falls } \Pi \text{ Maximierungsproblem} \end{cases}$$

*heißt **Approximationsalgorithmus mit relativer Gütegarantie**. $\mathcal{A}$ heißt ε–**approximativ** , falls $\mathcal{R}_\mathcal{A}(I) \leq 1 + \varepsilon$ für alle $I \in D_\Pi$.*

Bei einem Approximationsalgorithmus mit relativer Gütegarantie wird die Approximationsgüte also immer im Verhältnis zum Optimalwert betrachtet und nicht absolut wie oben.

Beispiel :
Wir betrachten folgenden **„Greedy-Algorithmus“** für KNAPSACK:

1. Berechne die „Gewichtsdichten“ $p_i := \frac{c_i}{w_i}$ für $i = 1, \ldots, n$ und indiziere so, daß $p_1 \geq p_2 \geq \ldots \geq p_n$, d.h. sortiere nach Gewichtsdichten. Dies kann in Zeit $\mathcal{O}(n \log n)$ geschehen.
2. Für $i = 1$ bis n setze $x_i := \left\lfloor \frac{W}{w_i} \right\rfloor$ und $W := W - \left\lfloor \frac{W}{w_i} \right\rfloor \cdot w_i$.

Es werden also der Reihe nach so viele Elemente wie möglich von der aktuellen Gewichtsdichte in die Lösung aufgenommen. Die Laufzeit dieses Algorithmus ist in $\mathcal{O}(n \log n)$. ■

4.33 Satz
Der Greedy–Algorithmus $\mathcal{A}$ für KNAPSACK erfüllt $\mathcal{R}_\mathcal{A}(I) \leq 2$ für alle Instanzen I.

Beweis: O.B.d.A. sei $w_1 \leq W$. Offensichtlich gilt:

$$\mathcal{A}(I) \geq c_1 \cdot x_1 = c_1 \cdot \left\lfloor \frac{W}{w_1} \right\rfloor \text{ für alle } I,$$

und

$$\mathrm{OPT}(I) \leq c_1 \cdot \frac{W}{w_1} \leq c_1 \cdot \left(\left\lfloor \frac{W}{w_1} \right\rfloor + 1 \right) \leq 2 \cdot c_1 \cdot \left\lfloor \frac{W}{w_1} \right\rfloor \leq 2 \cdot \mathcal{A}(I)$$

Also $\mathcal{R}_\mathcal{A}(I) \leq 2$. □

Bemerkung:
Die Schranke $\mathcal{R}_{\mathcal{A}}(I)$ ist in gewissem Sinne scharf. Betrachte dazu folgendes Problembeispiel I: Sei $n = 2,\ w_2 = w_1 - 1,\ c_1 = 2 \cdot w_1,\ c_2 = 2 \cdot w_2 - 1,\ W = 2 \cdot w_2$.

Dann ist

$$\frac{c_1}{w_1} = 2 > \frac{c_2}{w_2} = 2 - \frac{1}{w_2}$$

und $\mathcal{A}(I) = 2w_1$ und $\mathrm{OPT}(I) = 4w_2 - 2$, also

$$\frac{\mathrm{OPT}(I)}{\mathcal{A}(I)} = \frac{4w_2 - 2}{2w_1} = \frac{2w_1 - 3}{w_1} \longrightarrow 2 \qquad \text{für } w_1 \to \infty$$

4.34 Definition
Zu einem polynomialen Approximationsalgorithmus $\mathcal{A}$ sei

$$\mathcal{R}_{\mathcal{A}}^{\infty} := \inf \left\{ r \geq 1 \;\middle|\; \begin{array}{l} \text{es gibt ein } N_0 > 0, \text{ so daß } \mathcal{R}_{\mathcal{A}}(I) \leq r \\ \text{für alle } I \text{ mit } \mathrm{OPT}(I) \geq N_0 \end{array} \right\}$$

Problem COLOR
Gegeben: Graph $G = (V, E)$ und ein Parameter $K \in \mathbb{N}$.

Frage: Gibt es eine Knotenfärbung von G mit höchstens K Farben, so daß je zwei adjazente Knoten verschiedene Farben besitzen?

3COLOR bezeichnet das Problem COLOR mit festem Parameter $K = 3$.

COLOR entspricht also dem Optimierungproblem, bei dem eine minimale Anzahl benötigter Farben gesucht ist. Dagegen ist 3COLOR das Entscheidungsproblem für einen Graphen zu entscheiden, ob er dreifärbbar ist.

4.35 Satz
COLOR und 3COLOR sind $\mathcal{NP}$–vollständig (ohne Beweis).

4.36 Satz
Falls $\mathcal{P} \neq \mathcal{NP}$, dann existiert kein relativer Approximationsalgorithmus $\mathcal{A}$ für COLOR mit $\mathcal{R}_{\mathcal{A}}^{\infty} \leq \frac{4}{3}$.

Beweis: Wir zeigen, daß ein solcher Algorithmus verwendet werden kann, um einen polynomialen Algorithmus zum Lösen von 3COLOR anzugeben, im Widerspruch zu $\mathcal{P} \neq \mathcal{NP}$.

Zu zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ sei $G := (V, E) := G_1[G_2]$ definiert durch $V := V_1 \times V_2$ und

$$E := \left\{ \{(u_1, u_2), (v_1, v_2)\} \;\middle|\; \begin{array}{l} \text{entweder } \{u_1, v_1\} \in E_1, \text{ oder} \\ u_1 = v_1 \text{ und } \{u_2, v_2\} \in E_2 \end{array} \right\}$$

D.h. jeder Knoten aus G_1 wird durch eine Kopie von G_2 ersetzt, und jede Kante aus E_1 durch einen „vollständig bipartiten Graphen“ zwischen den entsprechenden Kopien. Betrachte Abbildung 4.5 für ein Beispiel.

Angenommen, es existiert ein relativer Approximationsalgorithmus $\mathcal{A}$ mit $\mathcal{R}_{\mathcal{A}}^{\infty} < \frac{4}{3}$. Dann existiert ein $N \in \mathbb{N}$ so, daß $\mathcal{A}(G) < \frac{4}{3}\,\mathrm{OPT}(G)$ für alle Graphen

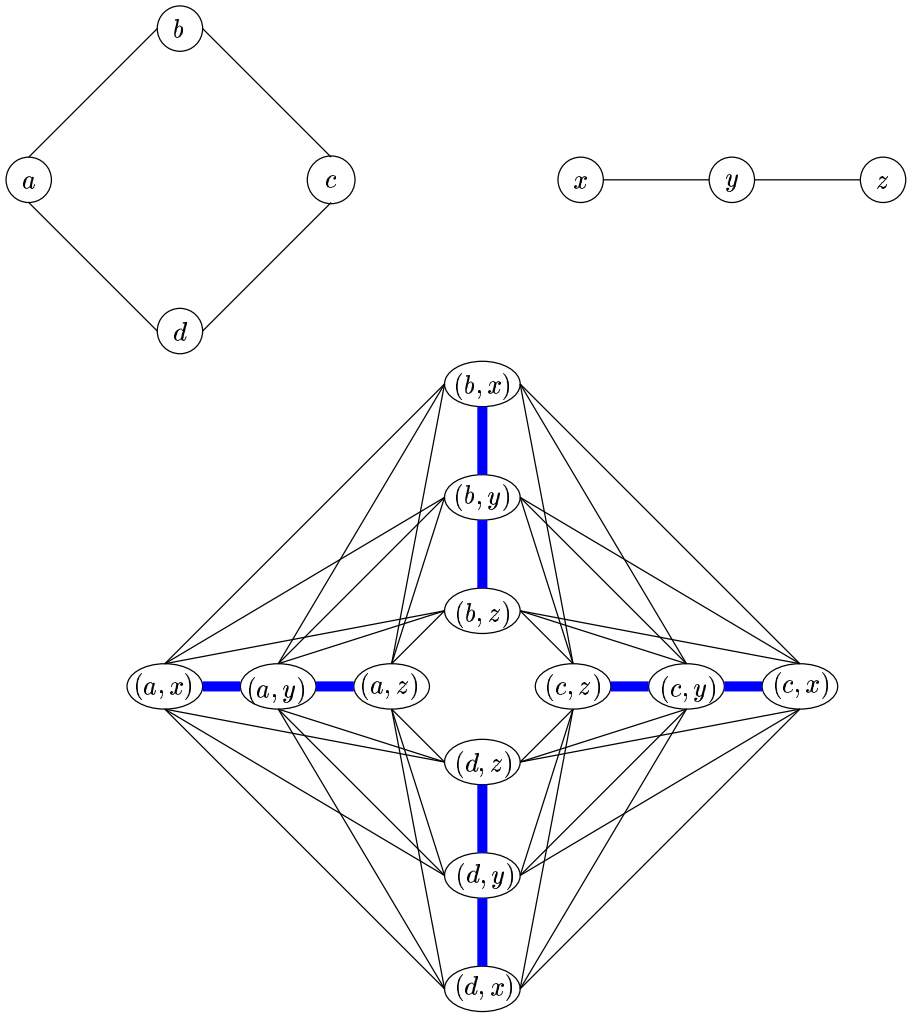

Abbildung 4.5: Graphen G_1, G_2 und $G_1[G_2]$ (von links nach rechts)

G mit $\mathrm{OPT}(G) \geq N$. Sei also $G = (V, E)$ ein beliebiges Beispiel für 3COLOR. Dann definiere $G^* := K_N[G]$, wobei K_N der vollständige Graph über N Knoten ist. G^* besteht also aus N Kopien von G, die vollständig miteinander verbunden sind. Dann gilt:

$$\mathrm{OPT}(G^*) = N \cdot \mathrm{OPT}(G) \geq N.$$

Da die Größe von G^* polynomial in der Größe von G ist, kann G^* in polynomialer Zeit konstruiert werden. Damit ist die Anwendung von $\mathcal{A}$ auf G^* polynomial in der Größe von G. Falls G dreifärbbar ist, dann gilt:

$$\mathcal{A}(G^*) < \frac{4}{3}\mathrm{OPT}(G^*) \leq \frac{4}{3} \cdot N \cdot 3 = 4N.$$

Andererseits gilt, falls G nicht dreifärbbar ist, daß

$$\mathcal{A}(G^*) \geq \mathrm{OPT}(G^*) = N \cdot \mathrm{OPT}(G^*) \geq 4N.$$

D.h. G ist dreifärbbar genau dann, wenn $\mathcal{A}(G^*) < 4N$. Dies ist ein polynomialer Algorithmus zur Lösung von 3COLOR im Widerspruch zu $\mathcal{P} \neq \mathcal{NP}$. □

Dieses Resultat kann verschärft werden zu $\mathcal{R}_{\mathcal{A}}^{\infty} \geq 2$ beziehungsweise zu $\mathcal{R}_{\mathcal{A}} \geq n^{\frac{1}{7}-\varepsilon}$ für jedes $\varepsilon > 0$, wobei $n = |V|$ für alle $\mathcal{A}$ unter Voraussetzung $\mathcal{P} \neq \mathcal{NP}$.

4.7.3 Approximationsschemata

Kann es für $\mathcal{NP}$–schwere Optimierungsprobleme noch bessere Approximierbarkeitsresultate geben als Approximationsalgorithmen mit relativer Gütegarantie K, wobei K konstant ist?

4.37 Definition
Ein (polynomiales) ***Approximationsschema (PAS)*** *für ein Optimierungsproblem Π ist eine Familie von Algorithmen $\{\mathcal{A}_\varepsilon \mid \varepsilon > 0\}$, so daß $\mathcal{A}_\varepsilon$ ein ε–approximierender Algorithmus ist (d.h. $\mathcal{R}_{\mathcal{A}_\varepsilon} \leq 1 + \varepsilon$) für alle $\varepsilon > 0$. Dabei bedeutet polynomial wie üblich polynomial in der Größe des Inputs I.*

Ein Approximationsschema $\{\mathcal{A}_\varepsilon \mid \varepsilon > 0\}$ heißt ***vollpolynomial (FPAS)*** *falls seine Laufzeit zudem polynomial in $\frac{1}{\varepsilon}$ ist.*

4.38 Satz
Sei Π ein $\mathcal{NP}$–schweres Optimierungsproblem mit:

1. *$\mathrm{OPT}(I) \in \mathbb{N}$ für alle $I \in D_\Pi$, und*
2. *es existiert ein Polynom q mit $\mathrm{OPT}(I) < q(\langle I \rangle)$ für alle $I \in D_\Pi$ wobei $\langle I \rangle$ die Inputlänge von I ist.*

Falls $\mathcal{P} \neq \mathcal{NP}$, so gibt es kein FPAS $\{\mathcal{A}_\varepsilon \mid \varepsilon > 0\}$ für Π.

Beweis: Sei $\{\mathcal{A}_\varepsilon \mid \varepsilon > 0\}$ ein FPAS für Π und o.B.d.A. sei Π ein Maximierungsproblem.

Für $I \in D_\Pi$ ist $\mathcal{A}_{\varepsilon_0}$ mit $\varepsilon_0 = \frac{1}{q(\langle I \rangle)}$ polynomial in $\langle I \rangle$ und in $\frac{1}{\varepsilon_0} = q(\langle I \rangle)$, also insgesamt polynomial in $\langle I \rangle$. Dann gilt:

$$\begin{aligned} OPT(I) &\leq (1+\varepsilon_0)\mathcal{A}_{\varepsilon_0}(I) \text{ und} \\ OPT(I) &\leq q(\langle I \rangle) = \frac{1}{\varepsilon_0} \end{aligned}$$

Also gilt

$$\mathrm{OPT}(I) - \mathcal{A}_{\varepsilon_0}(I) \leq \varepsilon_0 \cdot \mathcal{A}_{\varepsilon_0}(I) \leq \varepsilon_0 \cdot \mathrm{OPT}(I) < 1$$

Da $\mathrm{OPT}(I) \in \mathbb{N}$, ist also $\mathrm{OPT}(I) = \mathcal{A}_{\varepsilon_0}(I)$, im Widerspruch zu $\mathcal{P} \neq \mathcal{NP}$. □

Ein FPAS für spezielles $\mathcal{NP}$–vollständiges Problem (KNAPSACK)

Problem Spezielles KNAPSACK

Gegeben: Eine Menge von „Teilen“ $M = \{1, \ldots, n\}$, Kosten $c_1, \ldots, c_n \in \mathbb{N}_0$ und Gewichten $w_1, \ldots, w_n \in \mathbb{N}$ sowie ein Gesamtgewicht $W \in \mathbb{N}$.

Frage: Gib eine Teilmenge M' von M an, so daß

$$\sum_{i \in M'} w_i \leq W \text{ und } \sum_{i \in M'} c_i \text{ maximal ist.}$$

Bezeichne

$$w_r^j := \min_{M' \subseteq M, |M'|=j} \left\{ \sum_{i \in M'} w_i \;\middle|\; \sum_{i \in M'} c_i = r \right\}$$

für $r \in \mathbb{N}_0$. Dann erhalten wir folgenden Algorithmus zur Bestimmung einer maximalen Befüllung des Knapsacks.

1. für $1 \leq j \leq n$ setze $w_0^j := 0$
2. setze $j = 2,\ r = 0$
3. solange $w_r^j \leq W$

   berechne für $2 \leq j \leq n$ und $1 \leq r \leq \sum_{i=1}^n c_i =: c$ den Wert

   $$w_r^j = \min\{w_{r-c_j}^{j-1} + w_j, w_r^{j-1}\}$$

4. gib $C^* := \max\limits_{1 \leq j \leq n} \{r \mid w_r^j \leq W\}$ und die entsprechende Menge $M' \subset M$ mit $C^* = \sum_{i \in M'} c_i$ aus.

Die Laufzeit dieses Algorithmus $\mathcal{A}$ ist in $\mathcal{O}(n \cdot c)$ und die Lösung ist optimal. Dies ist also ein pseudopolynomialer optimaler Algorithmus für das spezielle KNAPSACK.

Betrachte nun das „skalierte“ Problem Π_k zu konstantem k mit $c_i' := \lfloor \frac{c_i}{k} \rfloor$ für alle $i \in M$. Dann liefert Algorithmus $\mathcal{A}$ für jedes $I_k \in \Pi_k$ eine Menge $M' \subseteq M$

mit $\sum_{i\in M'} c'_i = \mathrm{OPT}(I_k)$. Setze nun $c_{\max} := \max_{i\in M} c'_i$. Zu $\varepsilon > 0$ sei $\mathcal{A}_\varepsilon$ Algorithmus $\mathcal{A}$ angewendet auf I_k, wobei

$$k := \frac{c_{\max}}{\left(\frac{1}{\varepsilon}+1\right)\cdot n}$$

4.39 Satz
$\mathcal{R}_{\mathcal{A}_\varepsilon}(I) \le 1+\varepsilon$ für alle $I \in D_\Pi$ und die Laufzeit von $\mathcal{A}_\varepsilon$ ist in $\mathcal{O}(n^3 \cdot \frac{1}{\varepsilon})$ für alle $\varepsilon > 0$, d.h. $\{\mathcal{A}_\varepsilon \mid \varepsilon > 0\}$ ist ein FPAS für das spezielle KNAPSACK.

Beweis: Die Laufzeit von $\mathcal{A}_\varepsilon$ ist in $\mathcal{O}(n \cdot \sum_{i=1}^n c'_i)$ und

$$\sum_{i=1}^n c'_i = \sum_{i=1}^n \frac{c_i}{k} \le n \cdot \frac{c_{\max}}{k} = \left(\frac{1}{\varepsilon}+1\right) n^2.$$

Also ist die Laufzeit von $\mathcal{A}_\varepsilon$ in $\mathcal{O}(n^3 \cdot \frac{1}{\varepsilon})$ für alle $\varepsilon > 0$.

Für die Abschätzung von $\mathcal{R}_{\mathcal{A}_\varepsilon}$ betrachte M' mit $\mathrm{OPT}(I) = \sum_{i\in M'} c_i$. Dann gilt also

$$\mathrm{OPT}(I_k) \ge \sum_{i\in M'} \left\lfloor \frac{c_i}{k} \right\rfloor \ge \sum_{i\in M'} \left(\frac{c_i}{k} - 1\right).$$

Also ist

$$\mathrm{OPT}(I) - k \cdot \mathrm{OPT}(I_k) \le k \cdot n.$$

Da $\frac{1}{k}\mathcal{A}_\varepsilon(I) \ge \mathrm{OPT}(I_k)$ ist, folgt

$$\mathrm{OPT}(I) - \mathcal{A}_\varepsilon(I) \le k \cdot n$$

und wegen $\mathrm{OPT}(I) \ge c_{\max}$ (wir setzen wieder o.B.d.A. $W \ge w_i$ für alle $i \in M$ voraus) folgt

$$\begin{aligned}
\mathcal{R}_{\mathcal{A}_\varepsilon}(I) = \frac{\mathrm{OPT}(I)}{\mathcal{A}_\varepsilon(I)} &\le \frac{\mathcal{A}_\varepsilon(I) + kn}{\mathcal{A}_\varepsilon(I)} = 1 + \frac{kn}{\mathcal{A}_\varepsilon(I)} \\
&\le 1 + \frac{kn}{\mathrm{OPT}(I) - kn} \\
&\le 1 + \frac{kn}{c_{\max} - kn} = 1 + \frac{1}{\frac{1}{\varepsilon} + 1 - 1} \\
&= 1 + \varepsilon
\end{aligned}$$

□

Die hier benutzte Technik mit Hilfe eines pseudopolynomialen Algorithmus zu einem FPAS zu gelangen, kann auch bei anderen Optimierungsproblemen angewendet werden. Es gilt sogar:

4.40 Satz
Sei Π ein Optimierungsproblem für das gilt:

1. *$\mathrm{OPT}(I) \in \mathbb{N}$ für alle $I \in D_\Pi$*
2. *es existiert ein Polynom q mit $\mathrm{OPT}(I) \le q(\langle I\rangle + \max\#(I))$ ($\max\#(I)$ ist die größte in I vorkommende Zahl)*

Falls Π ein FPAS hat, so hat es einen pseudopolynomialen optimalen Algorithmus.

Beweis: Ähnlich wie im Beweis zu Satz 4.39. □

Kapitel 5

Grammatiken und die Chomsky-Hierarchie

Wir wollen Regelsysteme entwerfen, mit denen sich die Wörter einer vorgegebenen Sprache erzeugen lassen. Derartige Systeme werden **Grammatiken** genannt.

Beispiel :
Die Sprache aller Graphen $G = (V, E)$, die eine Clique der Größe $\frac{|V|}{2}$ enthalten, lassen sich aufbauen durch:

1. Wahl der Zahl n für $|V|$
2. Wahl einer Teilmenge der Größe $\frac{|V|}{2}$
3. Zugabe aller Kanten zwischen Knoten aus dieser Teilmenge
4. Zugabe weiterer Kanten

Dieses Regelsystem ist an drei Stellen nichtdeterministisch: 1, 2 und 4. ■

Beispiel :
Arithmetische Ausdrücke

- a, $a + a$ und $a \cdot a$ sind arithmetische Ausdrücke (a Symbol aus Alphabet).
- falls A_1 und A_2 arithmetische Ausdrücke sind, so sind auch $(A_1) + (A_2)$ und $(A_1) \cdot (A_2)$ arithmetische Ausdrücke.

Die Klammern sind teilweise überflüssig. ■

5.1 Definition
*Eine **Grammatik** G gesteht aus vier Komponenten:*

- *dem endlichen **Alphabet** Σ;*

- *einer endlichen Menge V mit $V \cap \Sigma = \emptyset$ von* ***Variablen****;*
- *dem* ***Startsymbol*** *$S \in V$;*
- *einer endlichen Menge von* ***Ableitungsregeln*** *R. Dabei ist eine Ableitungsregel ein Paar (ℓ, r), wobei $\ell \in (V \cup \Sigma)^+$ und $r \in (V \cup \Sigma)^*$ ist. Wir schreiben oft auch $\ell \to r$*

Bedeutung: *Wenn in einem Wort z das Wort ℓ Teilwort von z ist, so darf ℓ durch r in z ersetzt werden.*

Notation: *Wir schreiben $w \to z$, wenn w durch Anwendung einer Ableitungsregel in z verwandelt wird, und $w \overset{*}{\to} z$, wenn w durch eine Anwendung von mehreren Ableitungsregeln in z verwandelt wird.*

Die von einer Grammatik G ***erzeugte Sprache*** *$L(G)$ ist die Menge aller Wörter $z \in \Sigma^*$, für die $S \overset{*}{\to} z$ gilt.*

Beispiel :
Grammatik für die Menge aller arithmetischen Ausdrücke über a.

$$\begin{array}{lcl} \Sigma & = & \{(,), a, +, \cdot\} \\ V & = & \{S\} \\ R & : & S \to (S) + (S) \qquad S \to (S) \cdot (S) \\ & & S \to a \qquad S \to a + a \qquad S \to a \cdot a \end{array}$$

■

Man interessiert sich nun für ein $w \in \Sigma^*$ und eine Grammatik G, ob $w \in L(G)$ ist. Eine Anwendung ist zum Beispiel die Frage, ob eine Zeichenkette w bezüglich einer gegebenen Syntax einer Programmiersprache syntaktisch richtig ist.

5.2 Definition (Chomsky-Hierarchie)

1. *Grammatiken ohne weitere Einschränkungen heißen Grammatiken vom* ***Typ 0****.*
2. *Grammatiken, bei denen alle Ableitungsregeln die Form*
   - $u \to v \quad$ *mit* $u \in V^+, v \in ((V \cup \Sigma) \setminus \{S\})^+$ *und* $|u| \leq |v|$, *oder*
   - $S \to \varepsilon$

   haben, heißen ***kontextsensitiv*** *oder Grammatiken vom* ***Typ 1****.*
3. *Grammatiken, bei denen alle Ableitungsregeln die Form*
   $$A \to v \quad \text{mit } A \in V \text{ und } v \in (V \cup \Sigma)^*$$
   haben, heißen ***kontextfrei*** *oder Grammatiken vom* ***Typ 2****.*
4. *Grammatiken, bei denen alle Ableitungsregeln die Form*
   $$A \to v \qquad \text{mit } A \in V \text{ und } v = \varepsilon \text{ oder } v = aB \text{ mit } a \in \Sigma, B \in V$$
   haben, heißen ***rechtslinear*** *oder Grammatiken vom* ***Typ 3****.*

Bemerkung:
Bei kontextsensitiven Grammatiken kann die Ableitung $ABC \to AXYC$ erlaubt, aber $DBC \to DXYC$ verboten sein.

5.1 Chomsky–0–Grammatiken und rekursiv aufzählbare Sprachen

5.3 Satz
Falls L rekursiv aufzählbar (semi-entscheidbar) ist, so gibt es eine Chomsky–0–Grammatik mit $L(G) = L$.

Beweis: Da L rekursiv aufzählbar ist, gibt es eine deterministische Turing-Maschine $\mathcal{M}$, die genau die Wörter $w \in L$ akzeptiert.

O.B.d.A. habe $\mathcal{M}$ genau einen akzeptierenden Endzustand q_J, und wenn q_J erreicht wird, stehen auf dem Band nur Blanks. Außerdem soll $\mathcal{M}$ zunächst das Zeichen # hinter die Eingabe schreiben, und keine Position des Bandes rechts von # benutzen.

Dann können wir als neue Anfangskonfiguration $(q_0)w_1 \dots w_n\#$ annehmen, daß also der Kopf am Anfang des Wortes $w_1 \dots w_n\#$ steht. q_0 komme nur in der Anfangskonfiguration vor. Die Grammatik G soll nun die Berechnung aus der akzeptierenden Konfiguration (q_J) zu allen Anfangskonfigurationen rückwärts erzeugen können, wenn die Turing-Maschine aus der Anfangskonfiguration zu dem akzeptierenden Zustand gelangt. Die Grammatik vollzieht dazu alle möglichen Konfigurationen von $\mathcal{M}$ nach.

Beschreibung der Grammatik G:

1. Erzeugung der akzeptierenden Schlußkonfiguration:
$$S \to q_J, \quad q_J \to \sqcup q_J, \quad q_J \to q_J \sqcup$$
2. Rückwärtsrechnung
Falls $\delta(q, a) = (q', a', R)$, dann enthält G die Ableitungsregel
$$a'q' \to qa$$
Falls $\delta(q, a) = (q', a', L)$, dann enthält G die Ableitungsregel
$$q'ba' \to bqa \quad \text{für alle } b \in \Gamma$$
Falls $\delta(q, a) = (q', a', N)$, dann enthält G die Ableitungsregel
$$q'a' \to qa$$
3. Schlußregeln
Um aus $\sqcup \sqcup \dots \sqcup q_0 w_1 \dots w_n\#$ zu $w_1 \dots w_n$ zu kommen, reichen die Ableitungsregeln
$$\sqcup q_0 \to q_0, \quad q_0 a \to a q_0, \quad q_0\# \to \varepsilon$$

Alle Wörter $w \in L$ können durch G erzeugt werden, indem die Berechnung von $\mathcal{M}$ rückwärts durchlaufen wird. Umgekehrt kann G nur Berechnungen von $\mathcal{M}$ rückwärts erzeugen. Daher ist $L(G) = L$. □

5.4 Satz
Die von Typ–0–Grammatiken G erzeugten Sprachen sind rekursiv aufzählbar.

Beweis: Wir geben zunächst zu $L(G)$ eine nichtdeterministische Turing-Maschine an, die $L(G)$ akzeptiert.

Die Maschine schreibt zunächst S auf das Band, wählt dann eine beliebige anwendbare Ableitungsregel aus und vergleicht das erzeugte Wort mit der Eingabe w. Bei Gleichheit wird w akzeptiert, ansonsten eine weitere Ableitungsregel gewählt usw. Falls $w \in L(G)$, so gibt es eine akzeptierende Berechnung.

Wie in Übungsaufgabe 30 (Blatt 11) kann nun eine deterministische Turing-Maschine konstruiert werden, die dasselbe leistet. Es werden einfach nacheinander alle nicht-deterministischen Berechnungen simuliert. □

Das heißt die Klasse der rekursiv aufzählbaren Sprachen ist genau:

1. die Klasse der von deterministischen Turing-Maschinen akzeptierten Sprachen;
2. die Klasse der von nichtdeterministischen Turing-Maschinen akzeptierten Sprachen;
3. die Klasse der von Typ–0–Grammatiken erzeugten Sprachen;

Wir haben bewiesen, daß die Typ–0–Grammatiken genau die rekursiv aufzählbaren Sprachen erzeugen. Als Grundlage für Programmiersprachen sind die Typ–0–Grammatiken also sicherlich zu allgemein. Das Wortproblem ist für Typ–0–Grammatiken insbesondere gerade die universelle Sprache L_u (siehe Definition 3.12, $L := \{wv \mid v \in L(T_w)\}$), und die ist unentscheidbar.

5.2 Chomsky–3–Grammatiken und reguläre Sprachen

5.5 Satz
Die Klasse der von endlichen Automaten akzeptierten Sprachen ist genau die Klasse der von Chomsky–3–Grammatiken erzeugten Sprachen.

Beweis: Zu zeigen ist:

$\Rightarrow$: Zu einer Sprache L, die von einem endlichen Automaten akzeptiert wird, gibt es eine rechtslineare Grammatik, die L erzeugt.

$\Leftarrow$: Zu einer rechtslinearen Grammatik G gibt es einen endlichen Automaten, der gerade die Sprache $L(G)$ akzeptiert.

Zur Erinnerung: rechtslinear bedeutet, daß alle Regeln die Form $A \to v$ mit $A \in V$ und $v = \varepsilon$ oder $v = aB$ mit $a \in \Sigma$ und $B \in V$ haben.

$\Rightarrow$: Sei $\mathcal{A}_L$ ein deterministischer endlicher Automat, der die Sprache $L \subseteq \Sigma^*$ akzeptiert, $\mathcal{A}_L = (Q, \Sigma, \delta, q_0, F)$. G_L sei definiert durch:

- $V := Q$;
- $S := q_0$;
- R enthält die Regel $q \to \varepsilon$ für alle $q \in F$ und die Regel $q \to aq'$, falls $\delta(q, a) = q'$.

Für $w = w_1 \dots w_n \in L$ durchläuft $\mathcal{A}_L$ genau die Zustände $q_0, q_1 \dots, q_n \in Q$ mit $q_n \in F$. Dann gilt:

$$q_0 \to w_1 q_1 \to w_1 w_2 q_2 \to \dots \to w q_n \to w$$

Außerdem gibt es für alle Ableitungen von G_L eine entsprechende akzeptierende Berechnung des endlichen Automatens $\mathcal{A}_L$.

$\Leftarrow$: Zu L sei die Chomsky–3–Grammatik G_L gegeben. Wir entwerfen einen nichtdeterministischen endlichen Automaten $\mathcal{A}_L := (Q, \Sigma, \delta, q_0, F)$, der L akzeptiert. Setze:

- $Q := V$;
- $q_0 := S$;
- $F := \{A \in V \mid (A \to \varepsilon) \in R\}$
- $\delta(A, a) := \{B \mid (A \to aB) \in R\}$.

Für $w = w_1 \dots w_n \in L$ hat die Ableitung von w mittels G_L das Aussehen:

$$S \to w_1 A_1 \to w_1 w_2 A_2 \to \dots \to w A_n \to w$$

Der nichtdeterministische endliche Automat $\mathcal{A}_L$ kann dann bei der Eingabe von $w = w_1 \dots w_n$ folgende Abarbeitung durchlaufen:

$$S \xrightarrow{w_1} A_1 \xrightarrow{w_2} A_2 \xrightarrow{w_3} \dots \xrightarrow{w_{n-1}} A_{n-1} \xrightarrow{w_n} A_n,$$

wobei $A_n \in F$, also w akzeptiert wird.

Außerdem gibt es für alle akzeptierenden Berechnungen von $\mathcal{A}_L$ eine entsprechende Ableitung in G_L. □

Zu den Aufgaben eines Compilers gehört es, die syntaktische Korrektheit von Programmen zu überprüfen. Dazu gehört unter anderem die Überprüfung von *Klammerstrukturen auf Korrektheit*, also insbesondere, ob die Anzahl von Klammeröffnungen gleich der Anzahl der Klammerschließungen ist. Wir wissen, daß die Sprache der korrekten Klammerausdrücke nicht regulär ist. Typ–3–Grammatiken sind also zu einschränkend, um syntaktisch korrekte Programme zu beschreiben.

5.3 Chomsky–1–Grammatiken bzw. kontextsensitive Sprachen

Die Klasse der Typ-0-Grammatiken ist so groß, daß das Wortproblem, also die Entscheidung, ob ein Wort $w \in L(G)$ zu G ist, nicht entscheidbar ist. Andererseits lassen sich mit Typ-3-Grammatiken keine sinnvollen Programmiersprachen

beschreiben. Kontextsensitive und kontextfreie Grammatiken liegen zwischen diesen beiden. Wir werden sehen, daß für gewisse kontextsensitive Grammatiken das Wortproblem $\mathcal{NP}$–vollständig ist. Damit wären kontextsensitive Grammatiken ebenfalls zu allgemein, um als Basis für Programmiersprachen zu gelten.

Man kann beweisen, daß die Klasse der kontextsensitiven Sprachen genau die Klasse der Sprachen ist, die von einer nichtdeterministischen Turing-Maschine mit einem Speicherbedarfs, der „im wesentlichen" nicht die Länge der Eingabe überschreitet, erkannt werden. Für die Betrachtung des Speicherplatzbedarf einer Turing-Maschine treffen wir die Vereinbarung, daß die Eingabe auf einem Read-only Eingabeband steht, dessen Zellen beim Platzbedarf nicht berücksichtigt werden. Der erste und der letzte Buchstabe der Eingabe sind markiert, so daß der Kopf des Eingabebandes die Eingabe nicht verlassen muß. Für den Speicherplatzbedarf zählen dann nur die Plätze eines zweiten Bandes, des **Arbeitsbandes**, die benutzt werden.

5.6 Definition
$\mathcal{DTAPE}(s(n))$ und $\mathcal{NTAPE}(s(n))$ sind die Klassen der Sprachen, die von einer deterministischen beziehungsweise einer nichtdeterministischen Turing-Maschine mit Platzbedarf $s(n)$ (bei Eingabelänge n) akzeptiert werden können.

Natürlich ist $\mathcal{DTAPE}(s(n)) \subseteq \mathcal{NTAPE}(s(n))$. Es gilt außerdem

$$\mathcal{NTAPE}(n) = \mathcal{NTAPE}(f(n)) \text{ für alle } f(n) \in \mathcal{O}(n)$$

(einfache Konstruktion).

5.7 Satz
Die Klasse der von Chomsky–1–Grammatiken erzeugten Sprachen stimmt mit der Klasse $\mathcal{NTAPE}(n)$ überein.

Beweis: (ohne Beweis) □

Es ist übrigens offen, ob $\mathcal{NTAPE}(n) = \mathcal{DTAPE}(n)$ ist. Sind nun Sprachen aus $\mathcal{NTAPE}(n)$ als Grundlage für den Entwurf von Programmiersprachen geeignet? Immerhin erscheint „linearer Platzbedarf" nicht „abschreckend". Die Frage ist, wie groß die Zeitkomplexität des Wortproblems für eine kontextsensitive Sprache sein kann.

5.8 Satz
Das Cliquen-Problem gehört zu $\mathcal{DTAPE}(n)$.

Beweis: Gegeben sei $G = (V, E)$ mit $V = \{1, \ldots, n\}$ und $1 \leq K \leq n$. Auf linearem Platz kann ein beliebiger n-Vektor c über $\{0, 1\}$ dargestellt werden, wobei $c \in \{0, 1\}^n$ mit der Menge $C \subseteq V$ wie folgt korrespondiert:

$$c_i = 1 \quad \Longleftrightarrow i \in C$$

Nun kann mit linearem Speicherplatz für jeden Vektor $c \in \{0, 1\}^n$ getestet werden, ob die zugehörige Menge $C \subseteq V$ eine Clique der Größe K in G ist. Dazu muß nur die Anzahl der Einsen in c gezählt werden, und für jedes Paar c_i, c_j mit $c_i = c_j = 1$ in der Eingabe nachgesehen werden, ob $\{i, j\} \in E$.

Alle Vektoren $c \in \{0, 1\}^n$ können nacheinander, beginnend mit $(0, 0, \ldots, 0)$, durchgetestet werden, wobei:

- nach einem „positiven“ Test (G, K) akzeptiert wird;
- nach einem „negativen“ Test der Vektor durch seinen lexikalischen Nachfolger überschrieben wird.

Dazu wird insgesamt nur linearer Speicherplatz benötigt. □

Falls $\mathcal{P} \neq \mathcal{NP}$, kann also das Wortproblem für kontextsensitive Grammatiken im allgemeinen nicht in polynomialer Zeit entschieden werden.

Für das Arbeiten mit Chomsky–1–Grammatiken ist folgende Eigenschaft interessant: Chomsky–1–Grammatiken können „normalisiert“ werden, d.h. für jede Chomsky–1–Grammatik gibt es eine äquivalente Chomsky–1–Grammatik, bei der alle Regeln folgende Form haben:

- $A \to C$
- $A \to CD$
- $AB \to CD$
- $A \to a$
- $S \to \varepsilon$

wobei jeweils $A, B \in V$, $C, D \in V \backslash \{S\}$ und $a \in \Sigma$.

5.4 Chomsky–2–Grammatiken bzw. kontextfreie Sprachen und Syntaxbäume

Es bleibt die Frage ist, ob die kontextfreien Sprachen für den Entwurf von Programmiersprachen geeignet sind. Wir geben zunächst Chomsky–2–Grammatiken für einige Sprachen, von denen wir wissen, daß sie nicht regulär sind.

Notation
Statt Regeln

$$S \to \alpha \text{ und } S \to \beta$$

schreiben wir abkürzend

$$S \to \alpha \mid \beta$$

Beispiel :
$L = \{0^n 1^n \mid n \geq 1\}$ wird erzeugt durch die Grammatik: $V = \{S\}$, $\Sigma = \{0, 1\}$ und R besteht aus:

$$S \to 01 \mid 0S1$$

■

Beispiel :
$L = \{w \in \{0,1\}^* \mid w = w^R\}$ ist die Sprache der Palindrome über $\{0,1\}$. Es gilt:

1. $0, 1, \varepsilon$ sind Palindrome
2. falls w Palindrom, so auch $0w0$ und $1w1$
3. alle Palindrome lassen sich durch endliche viele Anwendungen von (1.) und (2.) erzeugen

Zugehörige kontextfreie Grammatik: $V = \{S\}$, $\Sigma = \{0,1\}$ und R enthält die Regeln:

$$S \to \varepsilon \mid 0 \mid 1$$

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

■

Beispiel :
L sei die Sprache aller $w \in \{0,1\}^+$ bei denen die Anzahl der Nullen gleich der Anzahl der Einsen ist. Bei der Erzeugung dieser Sprache muß jedesmal, wenn eine 0 bzw. eine 1 erzeugt wird, gespeichert werden, daß irgendwann eine 1 bzw. eine 0 erzeugt werden muß. Setze $V := \{S, A, B\}$, $\Sigma = \{0,1\}$ und

$$R := \{S \to 0B \mid 1A,\ A \to 0 \mid 0S \mid 1AA,\ B \to 1 \mid 1S \mid 0BB\}$$

D.h. aus A kann „eine Eins ausgeglichen“ werden, oder es wird eine Eins erzeugt und es müssen dann zwei Einsen ausgeglichen werden. Bei B gilt dies analog für die Nullen. Aus S kann eine 0 oder eine 1 erzeugt werden. Wenn eine 0 erzeugt wurde, muß diese mittels B ausgeglichen werden, analog für Einsen mittels A.

Durch Induktion über die Länge der durch G erzeugten Wörter läßt sich beweisen, daß $L = L(G)$. ■

Graphische Ableitung

Kontextfreie Grammatiken bestehen aus Regeln, deren linke Seite genau eine Variable aus V ist. Ihre Ableitungen lassen sich sehr gut als **Syntaxbäume** darstellen.

An der Wurzel eines solchen Syntaxbaumes steht das Startsymbol. Jeder innere Knoten enthält eine Variable. Die Blätter sind Symbole aus Σ oder ε. Wenn also ein innerer Knoten A als Nachfolger von links nach rechts $\alpha_1, \ldots, \alpha_r \in V \cup \Sigma$ hat, so muß $A \to \alpha_1 \ldots \alpha_r$ eine Ableitungsregel der Grammatik sein.

Beispiel :
Zur Sprache aller Wörter, die gleich viele Einsen wie Nullen enthalten, gibt es die Ableitung:

$$S \to 1A \to 11AA \to 11A0 \to 110S0 \to 1100B0 \to 110010$$

Der zugehörige Syntaxbaum ist in Abbildung 5.1(a) dargestellt.

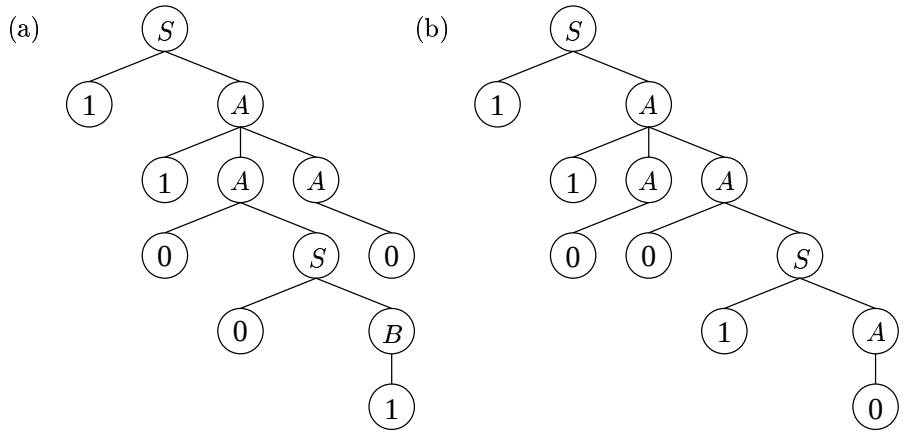

Abbildung 5.1: Syntaxbäume für Ableitungen des Wortes 110010

Zu jeder Ableitung gehört genau ein Syntaxbaum; zu jedem Syntaxbaum gehören jedoch verschiedene Ableitungen des gleichen Wortes. Hier auch:

$$S \to 1A \to 11AA \to 110SA \to 1100BA \to 11001A \to 1100110$$

■

Wegen der Kontextfreiheit ist die Reihenfolge, in der abgeleitet wird, für das Ergebnis unerheblich. Eine **Linksableitung** (**Rechtsableitung**) ist eine Ableitung, bei der in jedem Schritt die linkeste (rechteste) Variable abgeleitet wird.

5.9 Definition
*Eine kontextfreie Grammatik G heißt **eindeutig**, wenn es für jedes Wort $w \in L(G)$ genau einen Syntaxbaum gibt. Eine kontextfreie Sprache L heißt **eindeutig**, wenn es eine eindeutige Grammatik G mit $L(G) = L$ gibt. Ansonsten heißt L **inhärent mehrdeutig**.*

Die Grammatiken für $\{0^n 1^n \mid n \geq 1\}$ beziehungsweise $\{w \in \{0,1\}^* \mid w = w^R\}$ sind eindeutig. Die Grammatik für die Sprache der Wörter mit gleichvielen Nullen wie Einsen ist nicht eindeutig. Zu 110010 gibt es einen weiteren Syntaxbaum (siehe Abbildung 5.1(b))

Um zu entscheiden, ob nun eine Grammatik G die Sprache $L(G)$ mit $w \in L(G)$ erzeugt, ist es sehr hilfreich, wenn die Grammatiken in „Normalform“ sind.

5.10 Definition
*Eine kontextfreie Grammatik ist in **Chomsky–Normalform**, wenn alle Regeln von der Form:*

$$A \to BC \qquad \text{oder} \qquad A \to a$$

sind, mit $A, B, C \in V$ und $a \in \Sigma$. Grammatiken in Chomsky–Normalform können also nicht das Wort ε erzeugen. Für kontextfreie Sprachen, die ε enthalten, läßt sich eine Grammatik leicht „ergänzen“ durch die Regeln

$$S' \to \varepsilon \qquad \text{und} \qquad S' \to S$$

wobei S' ein neues Startsymbol zur Erzeugung von ε ist.

5.11 Satz
Jede kontextfreie Grammatik, die nicht das leere Wort erzeugt, kann in eine Grammatik in Chomsky–Normalform überführt werden.

Beweis: Wir geben eine „Schritt–für–Schritt"–Überführung der Regeln in Regeln in Normalform an.

1. Schritt: Alle Regeln enthalten auf der rechten Seite nur Symbole aus V oder nur ein Symbol aus Σ.

Ersetze dazu in allen rechten Seiten von Regeln Symbole aus $a \in \Sigma$ durch neue Variablen Y_a und füge die Regeln $Y_a \to a$ hinzu.

2. Schritt: Alle rechten Seiten haben Länge ≤ 2.

Sei $A \to B_1 \dots B_m$ Regel mit $m > 2$. Führe $m-2$ neue Variablen $C_1, \dots, C_{m-2}$ ein, und ersetze die Regel durch neue Regeln

$$\begin{gathered} A \to B_1 C_1 \\ C_i \to B_{i+1} C_{i+1} \text{ für } 1 \leq i \leq m-3 \\ C_{m-2} \to B_{m-1} B_m \end{gathered}$$

3. Schritt: Es kommen keine Regeln $A \to \varepsilon$ vor.

Zunächst berechnen wir die Menge V' aller Variablen A für die $A \overset{*}{\to} \varepsilon$ existiert: Es werden erst alle A mit $A \to \varepsilon$ aufgenommen. Dazu wird jedes rechte Vorkommen von A durch ε ersetzt. Dabei entstehen eventuell neue Regeln $B \to \varepsilon$, für die B in V' aufgenommen und genauso behandelt wird. Am Ende enthält V' alle Variablen A mit $A \overset{*}{\to} \varepsilon$. Nun werden alle Regeln $A \to \varepsilon$ gestrichen und für $A \to BC$ wird $A \to B$ falls $C \in V'$ beziehungsweise $A \to C$ falls $B \in V'$ eingeführt.

Es läßt sich leicht zeigen, daß dadruch keine neuen Wörter erzeugt werden können beziehungsweise alle Wörter, die vorher erzeugt werden konnten, immer noch erzeugt werden können.

4. Schritt: Ersetzung aller Kettenregeln $A \to B$.

Wir können zunächst alle Kettenregeln weiterverfolgen bis sich ein Kreis bildet. (DFS) D.h.:

$$A_1 \to A_2 \to A_3 \to \dots \to A_r \to A_1$$

Dann ersetzen wir alle $A_2, \dots, A_r$ in den Regeln durch A_1. Danach sortieren wir die verbleibenden A_i topologisch, so daß die Kettenregeln $A_i \to A_j$ nur existieren, falls $i < j$. Wir haben nun als Variablen $A_1, \dots, A_m$. Diese werden in der Reihenfolge $A_m, \dots, A_1$ abgearbeitet, und wenn eine Kettenregel $A_k \to A_\ell$ existiert, für die $A_\ell \to a$ ein Regel ist, so wird sie durch $A_k \to a$ ersetzt, für $a \in \Sigma$.

□

Der Cocke–Younger–Kasami Algorithmus für das Wortproblem bei kontextfreien Sprachen (1967)

5.12 Satz
Es gibt einen Algorithmus (den Cocke–Younger–Kasami Algorithmus), der für eine kontextfreie Grammatik G in Chomsky–Normalform und ein Wort $w \in \Sigma^*$ in Zeit $\mathcal{O}(|R| \cdot n^3)$ entscheidet, ob $w \in L(G)$, wobei $n = |w|$ und $|R|$ die Anzahl der Regeln von G ist.

Beweis: Sei $w = w_1 \dots w_n$. Für alle $1 \leq i \leq j \leq n$ soll die Menge $V_{ij} \subseteq V$ berechnet werden, so daß $A \overset{*}{\to} w_i \dots w_j$ impliziert $A \in V_{ij}$. Dann ist $w \in L(G)$ genau dann, wenn $S \in V_{1n}$ ist. Die Tabelle der V_{ij} wird nach wachsendem $\ell := j - i$ aufgebaut, beginnend mit $\ell = 0$. Für $j - i = \ell > 0$ wird die Berechnung von V_{ij} systematisch auf zuvor berechnete V_{ik}, $V_{k+1\ j}$ mit $i \leq k < j$ zurückgeführt ($\to$ dynamische Programmierung).

für $\ell = 0$: Konstruiere die Mengen V_{ii}, d.h. alle $A \in V$ mit $A \overset{*}{\to} w_i$. Da G in Chomsky–Normalform ist, gilt $A \overset{*}{\to} w_i$ nur, wenn $(A \to w_i) \in R$. Die Berechnung von V_{ii} ist für alle $i \in \{1, \dots n\}$ in $\mathcal{O}(|R|)$ möglich.

für $\ell > 0$: Konstruiere die Mengen V_{ij}, d.h. alle $A \in V$ mit $A \overset{*}{\to} w_i \dots w_j$. Da $j - i = \ell > 0$ ist, muß jede Ableitung von $w_i \dots w_j$ aus A mit einer Regel der Form $A \to BC$ beginnen, wobei ein $k \in \{i, \dots, j-1\}$ existiert mit $B \overset{*}{\to} w_i \dots w_k$ und $C \overset{*}{\to} w_{k+1} \dots w_j$. V_{ij} läßt sich nun aus den zuvor bestimmten Mengen $V_{ik}, V_{k+1\ j}$ für $k \in \{i, \dots, j-1\}$ mit Aufwand $\mathcal{O}(n \cdot |R|)$ wie folgt berechnen:

Speichere alle Mengen V_{rs} als Arrays der Länge $|V|$, in denen für jedes $A \in V$ markiert ist, ob $A \in V_{rs}$. Zur Berechnung von V_{ij} für festes $1 \leq i < j \leq n$ wird für jede Regel $(A \to BC) \in R$ und jedes k, $i \leq k < j$ in $\mathcal{O}(1)$ überprüft, ob $B \overset{*}{\to} w_i \dots w_k$ und $C \overset{*}{\to} w_{k+1} \dots w_j$ durch Ansehen der Stelle B im Array zu V_{ik} und C im Array zu $V_{k+1\ j}$.

Da insgesamt weniger als n^2 Mengen V_{ij} konstruiert werden müssen, ist der Gesamtaufwand des Verfahrens in $\mathcal{O}(|R| \cdot n^3)$. □

Das Pumping-Lemma und Ogden's Lemma für kontextfreie Sprachen

Ähnlich zum Pumping–Lemma für reguläre Sprachen gibt es auch ein Pumping–Lemma für kontextfreie Sprachen, das es ermöglicht, für gewisse Sprachen nachzuweisen, daß sie nicht kontextfrei sind.

5.13 Satz (Pumping–Lemma)
Für jede kontextfreie Sprache L gibt es eine Konstante $n \in \mathbb{N}$, so daß sich jedes Wort $z \in L$ mit $|z| \geq n$ so als $z = uvwxy$ schreiben läßt, daß $|vx| \geq 1$, $|vwx| \leq n$ und für alle $i \geq 0$ das Wort $uv^iwx^iy \in L$ ist.

Dieses Lemma läßt sich noch verallgemeinern.

5.14 Satz (Ogden's Lemma)
Für jede kontextfreie Sprache L gibt es eine Konstante $n \in \mathbb{N}$, so daß für jedes Wort $z \in L$ mit $|z| \geq n$ gilt: Wenn wir in z mindestens n Buchstaben markieren,

so läßt sich z so als $z = uvwxy$ schreiben, daß von den mindestens n markierten Buchstaben mindestens einer zu vx gehört und höchstens n zu vwx gehören und für alle $i \geq 0$ das Wort $uv^iwx^iy \in L$ ist.

Bemerkung:
Der Spezialfall von Odgen's Lemma, in dem alle Buchstaben von z markiert sind, ist gerade das Pumping–Lemma.