Informatik III

Skript zur Vorlesung von
Prof. Dr. Dorothea Wagner
WS 04/05

ausgearbeitet von
Marco Gaertler und Dagmar Handke

überarbeitet von
Silke Wagner

Vorwort

Diese Vorlesungsausarbeitung beruht auf der Vorlesung Informatik III, die ich im Wintersemester 2003/2004 an der Universität Karlsruhe gehalten habe. Im Sommersemester 1993 und in den Wintersemestern 1995/1996 und 1998/1999 habe ich bereit in Halle bzw. Konstanz Vorlesungen über die theoretischen Grundlagen der Informatik angeboten. Die Vorlesung Informatik III ist aus diesen Vorlesungen weiterentwickelt worden. Inhaltlich habe ich mich dabei sehr stark auf das Buch *Theoretische Informatik* von Ingo Wegener (erschienen 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.

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 Nacharbeitung des Vorlesungsstoffs und der Prüfungsvorbereitung hilfreich sein wird.

Karlsruhe, im Oktober 2004 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€ ,
- akzeptiert nur Münzen zu 1€ und 2€,
- gibt kein Wechselgeld heraus.
- warnt bei Überzahlung (zweimal 2€) 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.

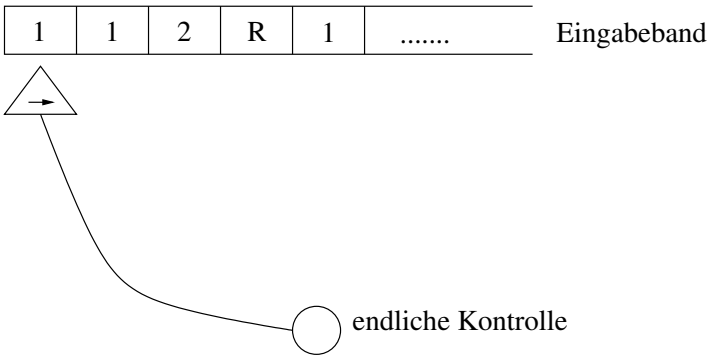


Abbildung 1.1: Modell für den Fahrkartenautomaten

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:

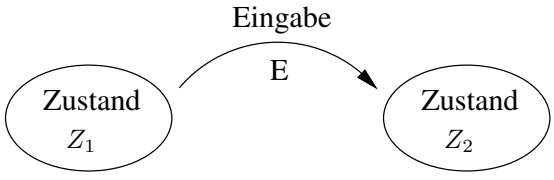


Abbildung 1.2: Zustandsübergang

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.

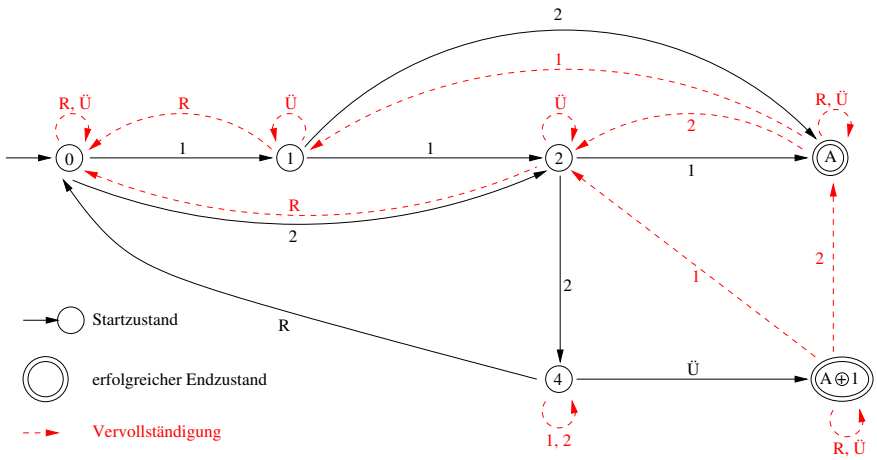


Abbildung 1.3: Übergangsgraph des Fahrkartenautomaten

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).

■

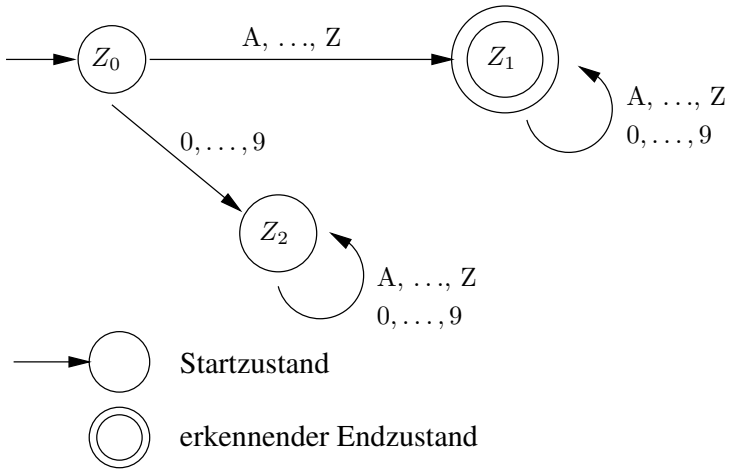


Abbildung 1.4: Dieser Automat erkennt alle korrekten Namen

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.

(3) Rundreise (TSP):

Gegeben: Fahrplan der Deutschen Bahn

Frage: Gibt es eine Verbindung, die in einer deutschen Stadt mit min. 600.000 Einwohnern[1] startet, in jeder (deutschen) Stadt mit min. 600.000 Einwohnern genau einmal hält, nur aus ICEs besteht und höchstens 20 Stunden braucht?

■

[1] In Deutschland gibt es 7 Städte mit mehr als 600.000 Einwohnern: Berlin(3.500.000), Dortmund(610.000), Essen(630.000), Frankfurt a. M.(660.000), Hamburg(1.661.000), Köln(1.003.000) und München(1.300.000).

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* $\boldsymbol{\varepsilon}$ *(*$|\varepsilon| = 0$*); 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.*

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

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:

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

■

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 Sprachen

(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:

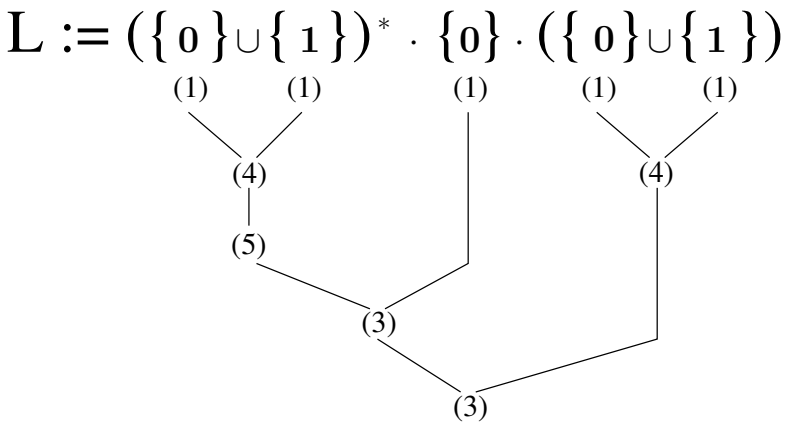


Abbildung 2.1: Aufbau der Sprache L

(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) $L :=$**(0∪1)*0(0∪1)** ist der reguläre Ausdruck für die Sprache des obigen Beispiels.

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

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

- $w \in$ **0*1*** $\Rightarrow\ w \in L$; also ist **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 $w \in$ **0*1***.

(4) $L := \{w \in \{0,1\}^* \mid w \text{ enthält 101 als Teilwort}\} =$ **(0 ∪ 1)*101(0 ∪ 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 $v_i \in$ **0*1*** für $1 \leq i \leq n$ und $w_{n+1} \in$ **ε ∪ 0(0*1*))**.

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

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

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

Insgesamt haben wir

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

$(*)$ 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).

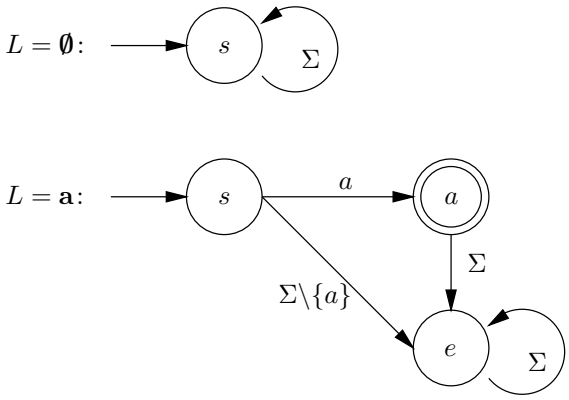


Abbildung 2.2: DEA's für den Induktionsanfang

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 $i = 1, 2$.

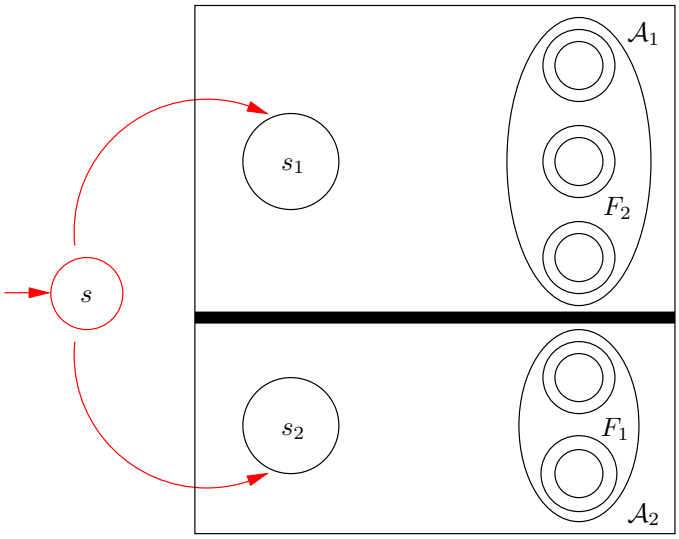


Abbildung 2.3: EA für $L = L_1 \cup L_2$

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

1. *Ein* ***nichtdeterministischer endlicher Automat*** *(NEA) besteht aus:*

   - Q, *einer endlichen Zustandsmenge;*
   - Σ, *einem endlichen 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;*

2. *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. ■

Anscheinend 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:

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

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 :

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

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

2. Erweiterung auf Mengen von Zuständen :

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

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

3. induktive Erweiterung auf Wörter:

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

$$\bar{\delta}(q,w) = \begin{cases} E(q) & \text{falls } w = \varepsilon \\ \delta(q,w) & \text{falls } w = a \in \Sigma \\ \bigcup\limits_{p \in \bar{\delta}(q,v)} \bar{\delta}(p,a) & \text{falls } w = va,\ a \in \Sigma,\ |v| > 0 \end{cases}$$

4. Analog für Mengen von Zuständen:

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

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

Oft schreiben wir δ auch an Stellen, an denen eigentlich $\bar{\delta}$ 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-Abschluss es 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

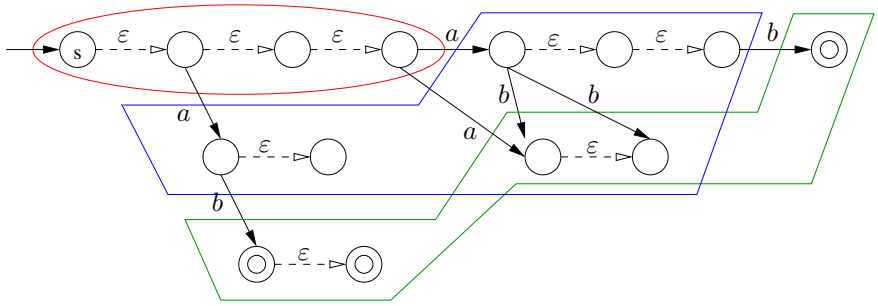


Abbildung 2.4: Mögliche Abarbeitungen von $w = ab$ in einem NEA mit $ab \in L,\ a \notin L$

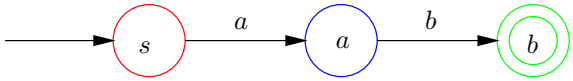


Abbildung 2.5: Die Abarbeitung von $w = ab$ bei einem äquivalenten DEA

DEA bestehen dafür aus Mengen 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 ($\equiv$ Mengen 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 $\widetilde{\delta}(\widetilde{q}, a) = \bar{\delta}(\widetilde{q}, a)$ 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:

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

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

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

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:

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

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

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

□

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:

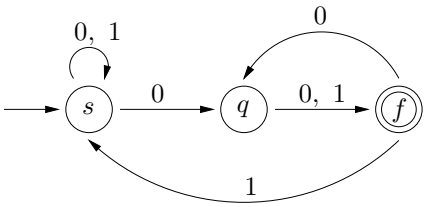


Abbildung 2.6: NEA für L

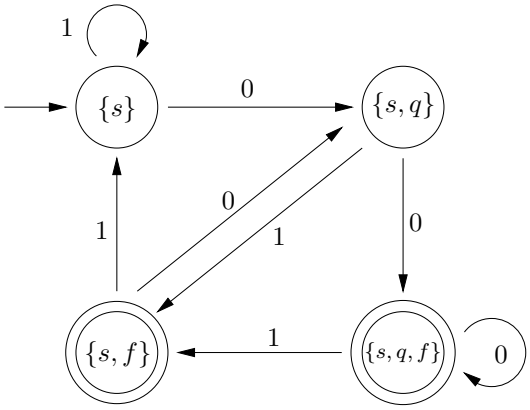


Abbildung 2.7: DEA für L

- 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.

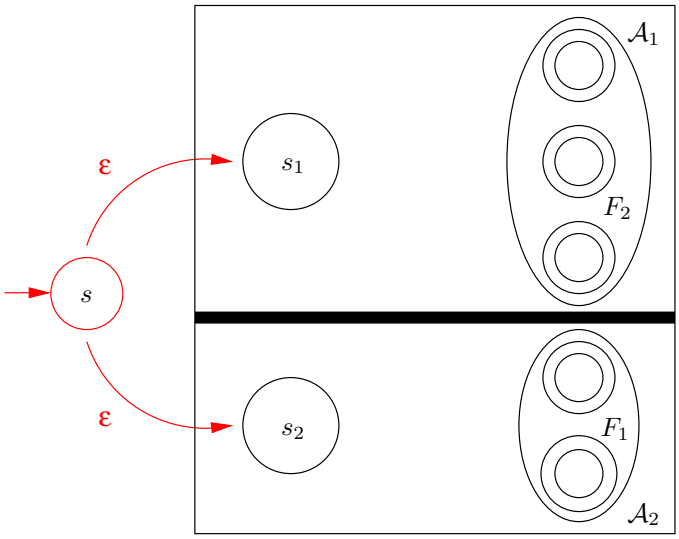


Abbildung 2.8: NEA für $L_1 \cup L_2$

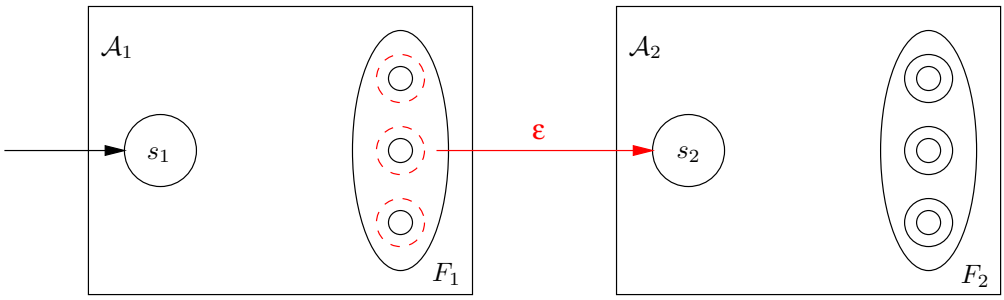


Abbildung 2.9: NEA für $L_1 \cdot L_2$

- 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

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

  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.

□

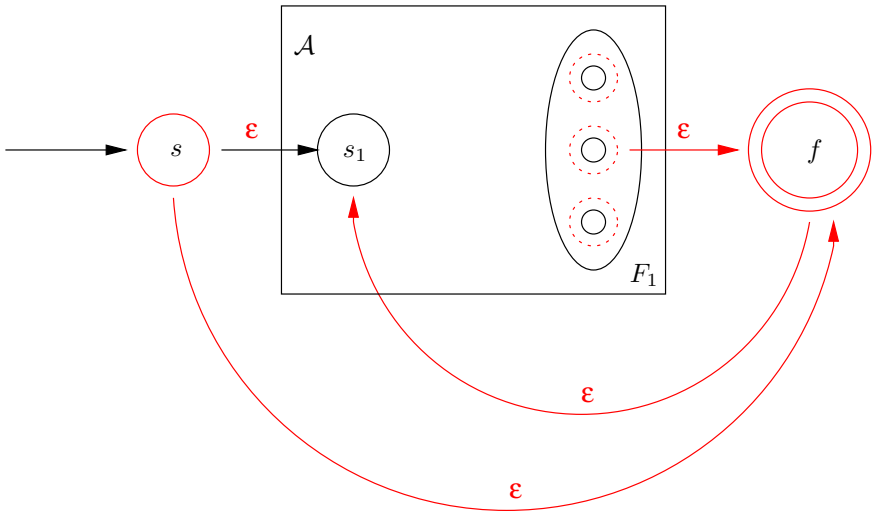


Abbildung 2.10: NEA für L_1^*

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 Automaten 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:

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

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 $q_r \xrightarrow{w} q_t$, 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.

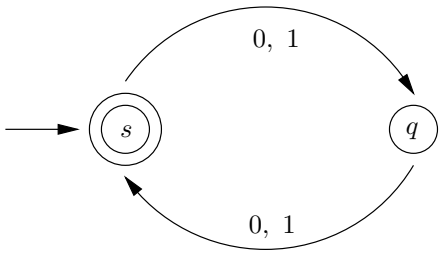


Abbildung 2.11: Beispiel

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 $i, j \in \{1,2\}, i \neq j$.

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

$$\begin{aligned} L = L_{q_1,2,q_1} &= L_{q_1,1,q_1} \cup (L_{q_1,1,q_2}(L_{q_2,1,q_2})^* L_{q_2,1,q_1}) \\ &= \varepsilon \cup (0 \cup 1)\left((0 \cup 1)(0 \cup 1)\right)^*(0 \cup 1) = ((0 \cup 1)(0 \cup 1))^* \end{aligned}$$

■

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:

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

Die Klammerung ist genau dann korrekt, wenn w gleich viele öffnende wie schließende Klammern 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 sind.)

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 müss te über unendliche viele Zustände verfügen. Die Sprache der Klammerausdrücke ist also zwar simpel, aber wohl 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 für jedes Wort $w \in L$ mit $|w| > n$ eine Darstellung

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

existiert, bei der 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 $i < j$.

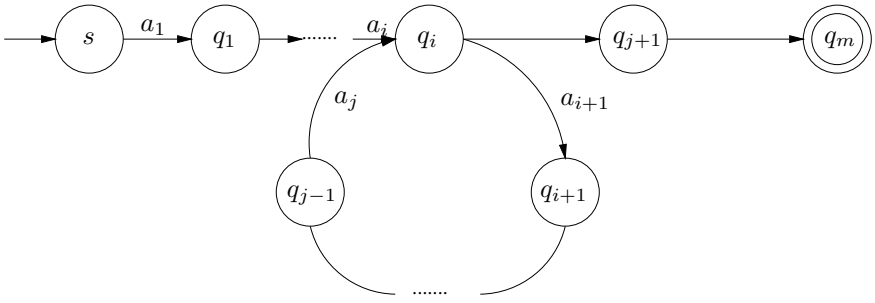


Abbildung 2.12: Abarbeitungs von w im DEA

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 $L = \left\{0^i 1^i \mid i \geq 0\right\}$. 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 entsprechend 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 beliebig aus $\mathbb{N}$ und $w = 0^{n^2} \in L$ bzw. $w = 0^4$, falls $n = 1$, 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 läss t sich mit folgendem verallgemeinertem Pumping Lemma zeigen.

■

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

für das Teilwort y existiert eine Darstellung $y = uvz$ mit $v \neq \varepsilon$ bei der auch $tuv^i zx \in L$ ist 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?

Wir können endliche Automaten als gerichtete Graphen auffassen. Die überflüssigen Zustände entsprechen dann den Knoten, zu denen es vom Anfangs-

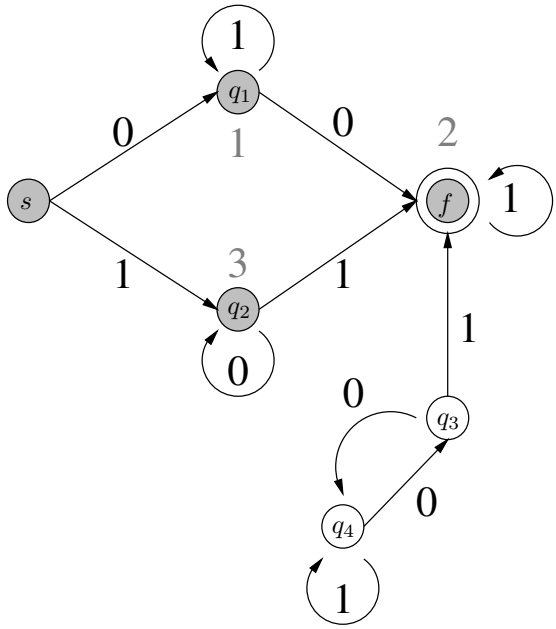


Abbildung 2.13: Beispiel

knoten 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. ■

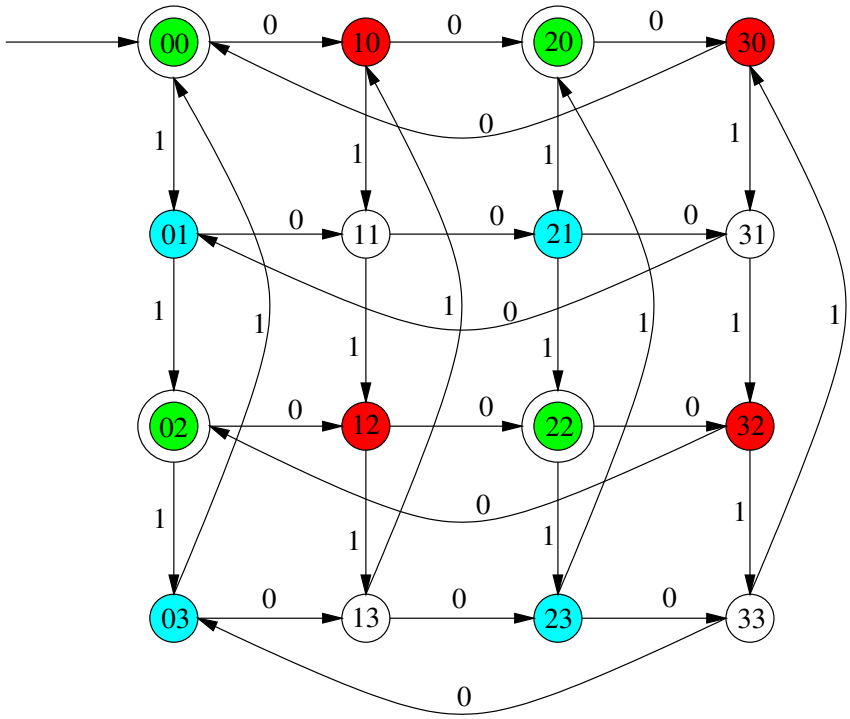


Abbildung 2.14: Gitterautomat

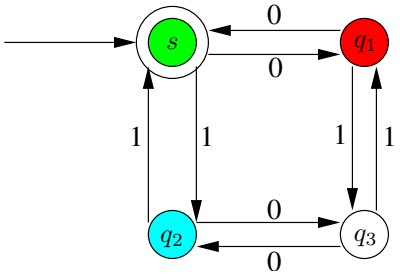


Abbildung 2.15: Äquivalenter Automat mit weniger Zuständen

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: $\delta(p, \varepsilon) \in F \Leftrightarrow \delta(q, \varepsilon) \in F$. Es ist $\delta(p, \varepsilon), \delta(q, \varepsilon) \in F$ 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 nach Definition von $\equiv$ auch für alle $a \in \Sigma$:

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

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:

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

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.

Idee: Wir testen systematisch Zustandspaare auf Nichtäquivalenz, indem wir alle Worte aus Σ^* entsprechend ihrer Länge betrachten und überprüfen, ob sie Zeuge für Nichtäquivalenz sind.

Frage: Wann kann dieses Verfahren abgebrochen werden?

Sei $w = aw'$ ein *kürzester* Zeuge für $p \not\equiv q$. Dann ist w' Zeuge für $p' := \delta(p, a) \not\equiv \delta(q, a) =: q'$. 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 für die Konstruktion von $\mathcal{A}^{\equiv}$ aus $\mathcal{A}$

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 durch:*

für $x, y \in \Sigma^*$ ist $x\ R_L\ y$ genau dann wenn $(xz \in L \Leftrightarrow yz \in L)$ für alle $z \in \Sigma^*$ 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:

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

$R_\mathcal{A}$ 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 $z \in \Sigma^* \colon xz\ R\ yz$. 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: $x\ R_L\ y \Rightarrow \delta(s,x) \equiv \delta(s,y)$

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

□

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

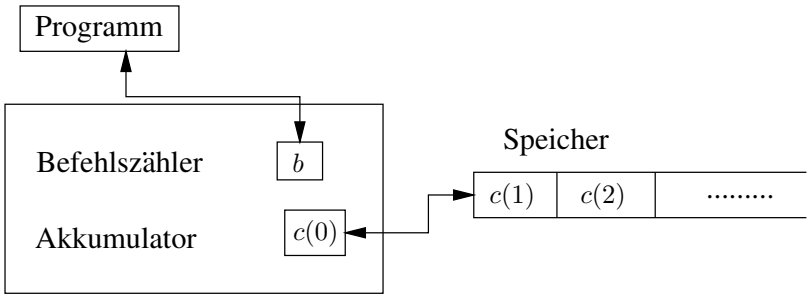


Abbildung 3.1: Schematische Darstellung der RAM

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.

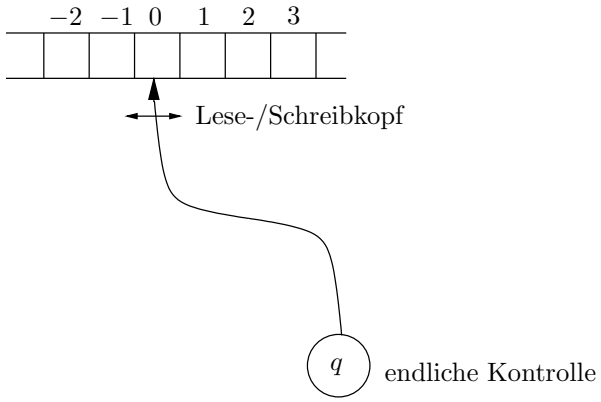


Abbildung 3.2: Schematische Darstellung der Turing-Maschine

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.

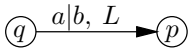


Abbildung 3.3: Übergang von Zustand q nach p

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.

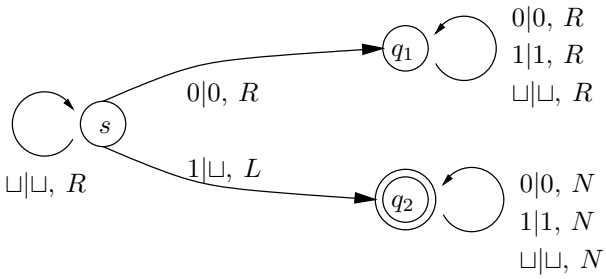


Abbildung 3.4: Turing-Maschine für die Sprache aller Wörter aus $\{0, 1\}^*$, die mit einer Eins beginnen

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 ($\equiv$ 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

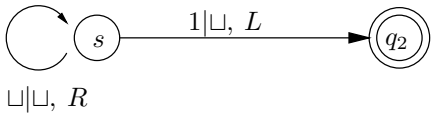


Abbildung 3.5: unvollständige Übergangsfunktion

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:

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

■

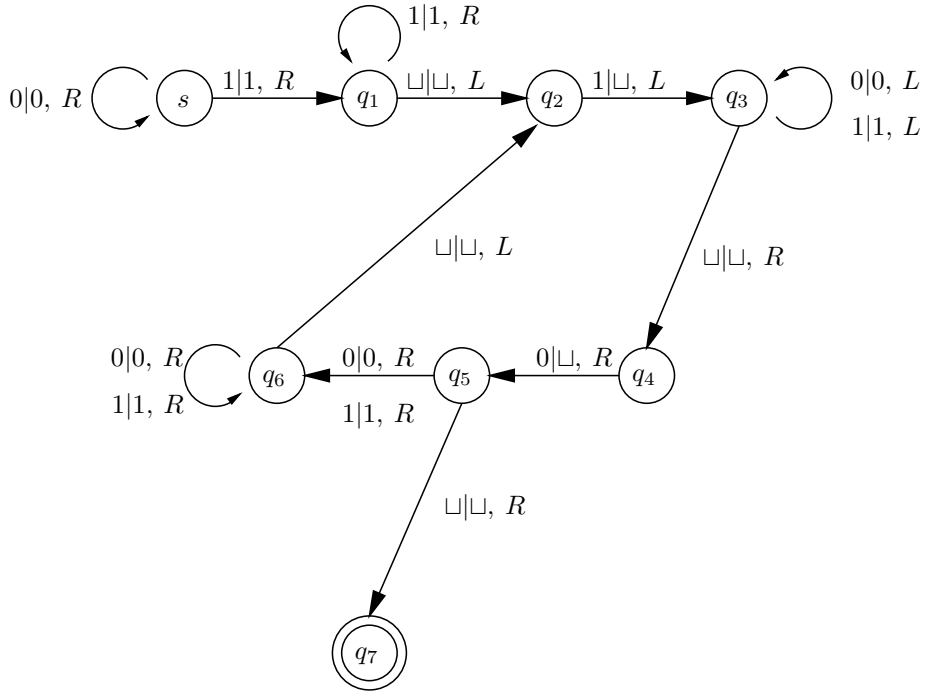


Abbildung 3.6: Turing-Maschine für $L^{=}$

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 und die Null selbst akzeptiert werden.

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)

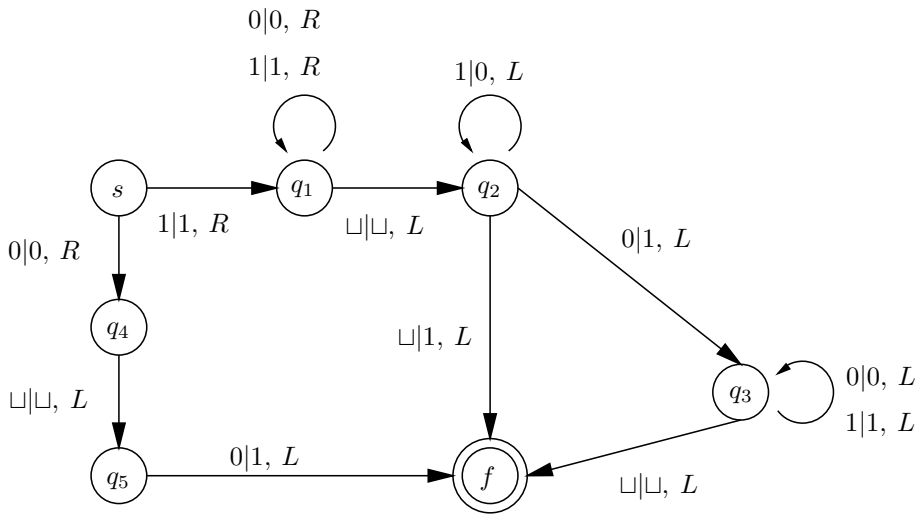


Abbildung 3.7: Turing-Maschine für Addition

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:*

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

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

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

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 Funktionen 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**

   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.

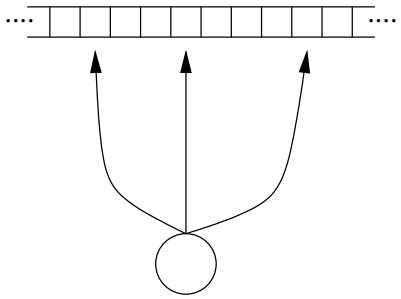


Abbildung 3.8: TM mit mehreren Lese-/Schreibköpfen

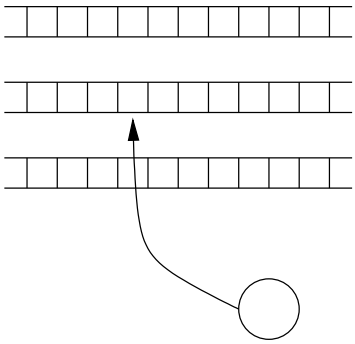


Abbildung 3.9: TM mit mehreren Bändern

2. **Mehrere Bänder**

   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 m Bänder und n Lese-/Schreibköpfe. 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

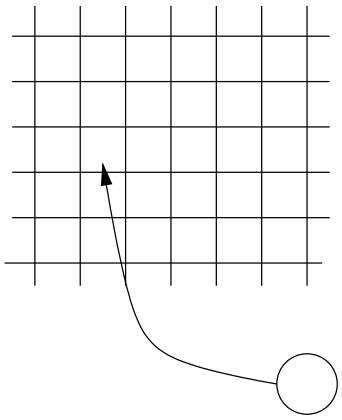


Abbildung 3.10: TM mit zweidimensionalem Band

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.

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 als Wort aus $(0 \cup 1)^*$ kodieren.

3.7 Definition
Sei $\mathcal{M} := (Q, \Sigma, \Gamma, \delta, s, F)$ eine Turing-Maschine. Die Gödelnummer von $\mathcal{M}$, bezeichnet als $\langle \mathcal{M} \rangle$, ist definiert 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:*

$$111 code_1 11 code_2 11 \ldots 11 code_z 111,$$

   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.

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

■

3.8 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. $\boldsymbol{L(T_w)}$ 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, die durch die Gödelnummer w_j kodiert ist. Wir 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							
		w_i	w_j	w_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	$w_k \notin L_d$
$\vdots$	$\vdots$							

■

3.9 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.

□

3.10 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 3.9. □

Bemerkung:
Korollar 3.10 kann folgendermaßen interpretiert werden: Das Problem, ob eine Turing-Maschine auf einer Eingabe w stoppt, ist nicht entscheidbar.

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

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

3.12 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 3.10 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. Betrachte die durch w_i kodierte Turing-Maschine $\mathcal{M}_i$.
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. □

3.13 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.

3.14 Satz
Die Sprache L_u ist nicht entscheidbar.

Beweis: Wir zeigen, dassL_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. Betrachte die durch w_i codierte Turing-Maschine $\mathcal{M}_i$.
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 3.10.
□

3.15 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.

3.16 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 Konsequenzen. 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 Korrespondenzproblem (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) $K = ((1, 111), (10111, 10), (10, 0))$ 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.

■

3.17 Satz
Das Post'sche Korrespondezproblem ist nicht entscheidbar

Beweis: Beweis über Nicht-Entscheidbarkeit des Halteproblems. □

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 $\lfloor \log_k |n| + 1 \rfloor + 1 =: \langle n \rangle$
  (eine 1 benötigt man für das Vorzeichen);
- 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 $\langle A \rangle := \sum_{i=1}^{m} \sum_{j=1}^{n} \langle a_{ij} \rangle$.
- eines Graphen $G = (V, E)$ kann zum Beispiel durch die Kodierung seiner *Adjazenzmatrix*, die eines „gewichteten" Graphen 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 $s\colon D_\Pi \to \Sigma^*$ 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

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

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*** *$T_{\mathcal{M}} \colon \mathbb{Z}^+ \to \mathbb{Z}^+$ 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 $\boldsymbol{\mathcal{P}}$ 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 an zu geben.

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

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

3. falls $\mathcal{A}(n, c, L)$ =„nein“ ist,
   setze $OPT := H$;
   sonst
   setze $OPT := L$;

4. für $i = 1 \ldots n$ führe aus
5. für $j = 1 \ldots n$ führe aus
6. setze $R := c_{ij}$; $\quad c_{ij} := m + 1$;
   falls $\mathcal{A}(n, c, OPT) =$ „nein“ ist, setze $c_{ij} := R$;
7. setze $d_{ij} = c_{ij}$;

Bemerkung:

Die Schleife in Schritt 2 bricht ab, und danach ist die Differenz $H - L$ gleich 1 oder 0, denn:

(a) Solange $H - L > 1$, ändert sich bei jedem Schleifendurchlauf einer der Werte H, L, da für $H - L > 1$ gilt, dass $L \neq \left\lceil \frac{1}{2}(H+L) \right\rceil + 1$ und $H \neq \left\lceil \frac{1}{2}(H+L) \right\rceil$ ist. Die Differenz verkleinert sich also mit jedem Durchlauf und da H und L ganzzahlig sind, tritt der Fall $H - L \le 1$ auf jeden Fall ein.

(b) Nach Abbruch der Schleife gilt $H - L \ge 0$: Es kann leicht eingesehen werden, dass eine Differenz zwischen H und L von 0 genau durch Erhöhen von L bei einer aktuellen Differenz von 2 bzw. 3 erreicht werden kann und ferner minimal ist (bei einer Differenz von weniger als 2 wird die Schleife nicht mehr betreten).

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 $\mathcal{O}(n^2 \cdot \log(\max c_{ij}))$. 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).

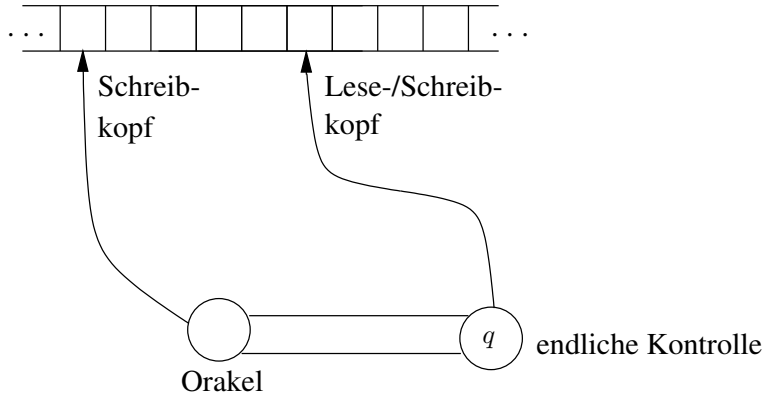


Abbildung 4.1: nichtdeterministische Turing-Maschine

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.

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.

■

Bemerkungen:

(1) Das Orakel kann ein beliebiges Wort aus Γ^* sein, es muss also nicht zwangsläufig die Struktur eines Lösungsbeispiels des Problems haben. Darum muss in der Überprüfungsphase (2.Stufe) zunächst geprüft werden, ob das Orakel ein zulässiges Lösungsbeispiel ist (beim TSP-Problem muss beispielsweise erst geprüft werden, ob das Orakel aus n Indizes besteht und ob jeder Knotenindex genau einmal vorkommt). Ist dies nicht der Fall, so kann die Berechnung zu diesem Zeitpunkt mit der Antwort „Nein“ beendet werden.

(2) Jede NTM $\mathcal{M}$ hat zu einer gegebenen Eingabe x eine unendliche Anzahl möglicher Berechnungen, eine zu jedem Orakel aus Γ^*. Endet mindestens eine in q_J, so wird x akzeptiert.

4.8 Definition

1. *Die **Zeit**, die eine nichtdeterministische Turing-Maschine $\mathcal{M}$ benötigt, um ein Wort $x \in L_\mathcal{M}$ zu akzeptieren, ist definiert als die minimale Anzahl von Schritten, die $\mathcal{M}$ in den Zustand q_J überführt.*

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

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

Bemerkungen:

(1) Zur Berechnung von $T_{\mathcal{M}}(n)$ wird für jedes $x \in L_{\mathcal{M}}$ mit $|x| = n$ die kürzeste akzeptierende Berechnung betrachtet, und anschließend von diesen kürzesten die längste bestimmt. Somit ergibt sich eine *worst–case* Abschätzung.

(2) Die Zeitkomplexität hängt nur von der Anzahl der Schritte ab, die bei einer akzeptierenden Berechnung auftreten. (Hierbei umfasst die Anzahl der Schritte auch die Schritte der Orakelphase). Per Konvention ist $T_{\mathcal{M}}(n) = 1$, 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****.)*

Bemerkung:
Alle Sprachen in $\mathcal{NP}$ sind entscheidbar (siehe Übung).

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

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

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. $\forall\, I \in D_{\Pi_1}\colon I \in J_{\Pi_1} \iff f(I) \in J_{\Pi_2}$.

  Wir schreiben dann $\Pi_1 \propto \Pi_2$.
- Ein Entscheidungsproblem 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

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

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 $t(u_1) = t(u_2) =$ `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 $O(m{\cdot}n)$ 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: $L \propto L_{\text{SAT}}$, wobei $L_{\text{SAT}} = L[\text{SAT}, s]$ 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

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

Wir benötigen eine Konstruktion von f_L, die zeigt, wie man die Überprüfung, ob eine nichtdeterministische 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 schließen:

$$
\begin{aligned}
x \in L \Leftrightarrow\ & \text{es existiert eine akzeptierende Berechnung von } \mathcal{M} \\
& \text{bei Eingabe } x \\
\Leftrightarrow\ & \text{es existiert eine akzeptierende Berechnung von } \mathcal{M} \text{ bei} \\
& \text{Eingabe } x \text{ mit höchstens } p(n) \text{ Schritten in der Überprüfungs-} \\
& \text{phase und einem Orakel } w \text{ der Länge } |w| = p(n) \\
\Leftrightarrow\ & \text{es existiert eine erfüllende Wahrheitsbelegung für die} \\
& \text{Klauselmenge } f_L(x)
\end{aligned}
$$

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:

Klausel-gruppe	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: $\mathcal{M}$ ist im Zustand q_0, der Lese-/Schreibkopf steht an Position 1 des Bandes; in den Zellen 1 bis n steht das Wort $x = s_{k_1} \ldots s_{k_n}$
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: $\quad Q[i,0] \vee \ldots \vee Q[i,r]$ 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 $0 \leq i \leq p(n)$
$\overline{H[i,j]} \vee \overline{H[i,j']}$ für $0 \leq i \leq p(n)$ und $-p(n) \leq j < j' \leq p(n)+1$
(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 $0 \leq i \leq p(n),\ -p(n) \leq j \leq p(n)+1$
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
$S[0,n+1,0], \ldots, S[0,p(n)+1,0]$

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 $0 \leq i < p(n)$, $0 \leq k \leq \ell$ und $-p(n) \leq j \leq p(n)+1$:

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

Dies ergibt die Klausel

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

$G_{6,2}$ 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

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

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: $(p(n)+1)(r+1) + (p(n)+1)\frac{1}{2}(r(r+1))$

G_2: $(p(n)+1)(2p(n)+1) + (p(n)+1)\frac{1}{2}(2p(n)\cdot(2p(n)+1))$

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

G_4: $2 + (n+1) + (p(n)+2-(n+1)) = p(n)+4$

G_5: 1

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

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** in einem Graphen $G = (V, E)$ ist $V' \subseteq V$ so, dass für alle $i, j \in V', i \neq j$, gilt: $\{i, j\} \in E$.*

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.

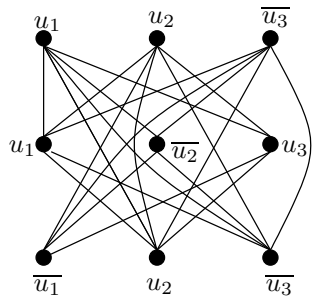


Abbildung 4.2: Graph G aus 3SAT $\propto$ CLIQUE

■

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).

Um einerseits einen „Grundstock“ $\mathcal{NP}$–vollständiger Probleme zu haben, und uns andererseits mit Techniken zum Beweis von $\mathcal{NP}$–Vollständigkeit vertraut zu machen, beweisen wir nun die $\mathcal{NP}$–Vollständigkeit einiger Probleme aus verschiedenen Gebieten.

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.

4.18 Satz
3COLOR ist $\mathcal{NP}$–vollständig.

Beweis:

1. 3COLOR $\in \mathcal{NP}$, da für $G = (V, E)$ und eine Färbung von G mit drei Farben in $\mathcal{O}(|E|)$ überprüft werden kann, ob diese zulässig ist.

2. Wir zeigen, dass 3SAT $\propto$ 3COLOR.

   Sei I ein beliebiges Beispiel für 3SAT, bestehend aus Klauseln $c_1, \ldots, c_n$ über der Variablenmenge $U = \{u_1, \ldots, u_m\}$. Wir konstruieren einen Graphen G derart, dass I erfüllbar ist genau dann, wenn G dreifärbbar ist.

   Wir beginnen die Konstruktion von G mit einem aus den „Eckknoten“ t, f und a bestehenden „Hauptdreieck“ D. t, f und a sind zugleich die drei Farben, mit denen G gefärbt werden soll.

   Interpretation: $t \mathrel{\widehat{=}}$ `wahr`; $f \mathrel{\widehat{=}}$ `falsch`.

   Für jede Variable $u \in U$ bilde ein Dreieck D_u mit Eckknoten $u, \bar{u}$ und a. Wir haben also $n + 1$ Dreiecke, die den Knoten a gemeinsam haben.

   Interpretation: Falls u mit „t“ gefärbt wird, so muss $\bar{u}$ mit „f“ gefärbt werden.

   Betrachte nun eine Klausel $c_j = x \vee y \vee z;\ x, y, z \in \{u_i, \bar{u}_i : u_i \in U\}$. Wir führen zu c_j eine Komponente C_j ein, bestehend aus sechs Knoten, einem „inneren Dreieck“ und drei „Satelliten“ (siehe Abbildung 4.3).

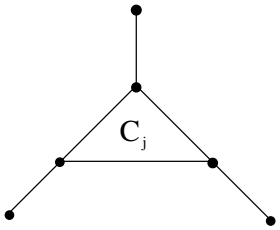


Abbildung 4.3: Komponente C_j

Jeder der drei Satelliten wird mit einem der Literale x, y, z verbunden und alle drei Satelliten werden mit dem Eckknoten t in D verbunden. Da der konstruierte Graph $\mathcal{O}(n + m)$ viele Knoten hat, ist die Transformation offensichtlich polynomial.

Beispiel :
$c_1 = \bar{u}_1 \vee u_2 \vee \bar{u}_3$, $c_2 = \bar{u}_1 \vee \bar{u}_2 \vee u_3$ mit $u_2 :=$ `wahr` und $u_3 :=$ `wahr` ergibt einen Graphen wie in Abbildung 4.4. ■

Zeige zunächst: Falls I erfüllbar, so ist G dreifärbbar.

Betrachte eine erfüllende Wahrheitsbelegung für U. D ist mit t, f und a wie angegeben „gefärbt“. Die Knoten u_i bzw. $\bar{u}_i$ werden entsprechend ihrem Wahrheitswert mit „t“ oder „f“ gefärbt. Für jede Klausel gibt es mindestens ein Literal, das `wahr` ist. Wähle pro Klausel jeweils ein solches Literal aus und färbe den mit dem entsprechenden Knoten verbundenen Satelliten in der zugehörigen Klauselkomponente mit „f“, die beiden anderen mit „a“. Dann kann das zugehörige innere Dreieck offensichtlich ebenfalls mit a, t und f zulässig gefärbt werden.

Zeige nun: Falls G dreifärbbar, so ist I erfüllbar.

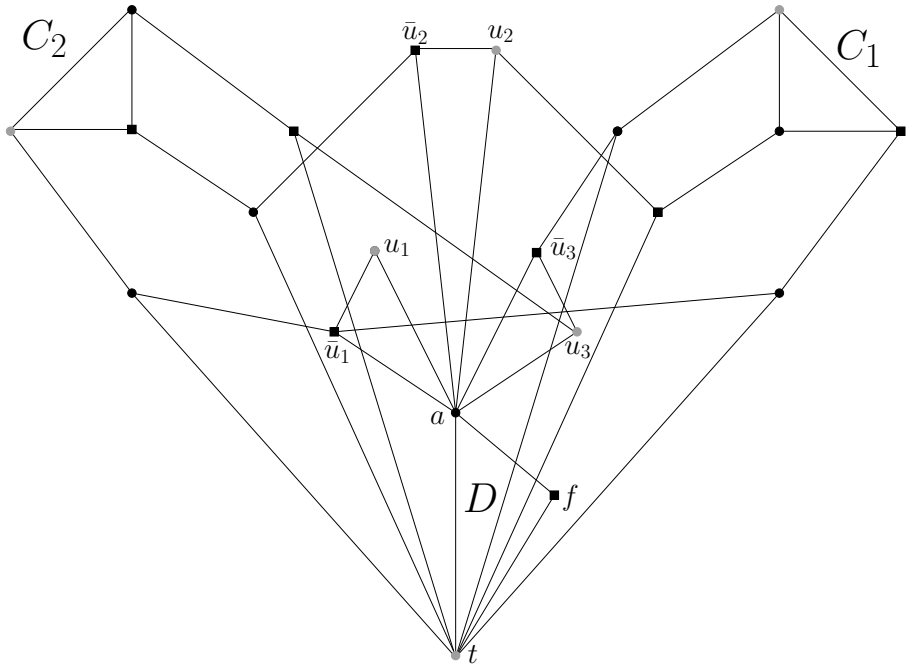


Abbildung 4.4: Graph zu einem 3SAT-Problem mit zwei Klauseln

Betrachte zunächst die Dreifärbung des Hauptdreiecks D und „interpretiere“ die Farben entsprechend den Knoten von D. Dann induziert die Dreifärbung der Dreiecke D_u eine konsistente Wahrheitsbelegung von U. Da für die Klauselkomponenten alle Satelliten mit „t“ verbunden sind, und die inneren Dreiecke der C_j-Komponenten mit drei Farben gefärbt sind, muss jeweils mindestens ein Satellit mit „f“ gefärbt sein. Daraus folgt, dass der damit verbundene Literalknoten mit „t“ gefärbt ist, das entsprechende Literal also die entsprechende Klausel erfüllt.

□

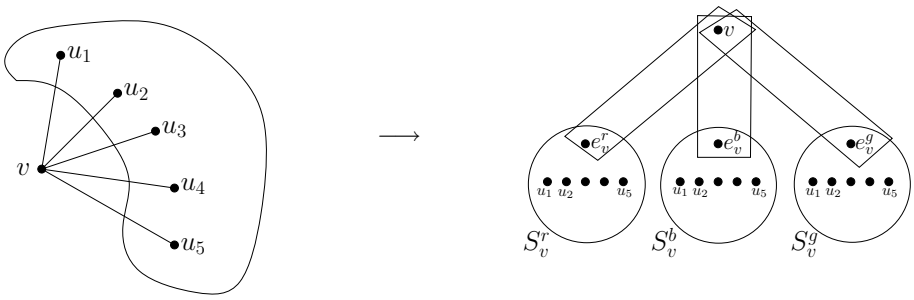


Abbildung 4.5: Konstruktion der Mengen $S_v^c, c \in C$, für einen Knoten $v \in V$ eines Graphen $G = (V, E)$.

Problem EXACT COVER

Gegeben: Eine endliche Menge X und eine Familie $\mathcal{S}$ von Teilmengen von X.

Frage: Existiert eine Menge $\mathcal{S}' \subseteq \mathcal{S}$, so dass jedes Element aus X in genau einer Menge aus $\mathcal{S}'$ liegt?

Beispiel:
$X = \{1, 2, \ldots, 7\}$,

$\mathcal{S} = \{\{1,2,3\}, \{1,2,4\}, \{2,3,4\}, \{1,3,4\}, \{1,5\}, \{3,5\}, \{1,3\}, \{5,6,7\}, \{4,5,6\}, \{4,5,7\}, \{4,6,7\}, \{5,6\}, \{5,7\}, \{6,7\}\}$,

$\mathcal{S}' = \{\{1,5\}, \{2,3,4\}, \{6,7\}\}$. ■

4.19 Satz
EXACT COVER ist $\mathcal{NP}$–vollständig.

Beweis:

Offensichtlich ist EXACT COVER in $\mathcal{NP}$.

Wir beweisen 3COLOR $\propto$ EXACT COVER.
Für einen beliebigen Graphen $G = (V, E)$ geben wir also ein Beispiel $(X, \mathcal{S})$ für EXACT COVER an, das eine exakte Überdeckung $\mathcal{S}'$ besitzt genau dann, wenn G dreifärbbar ist.
Sei $C = \{r(ot), b(lau), g(rün)\}$ für $v \in V$. Sei $N(v) := \{u \in V : \{u, v\} \in E\}$ die Nachbarschaft von v. Die Menge X enthalte für jedes $v \in V$ ein „Element“ v und jeweils $3 \cdot |N(v)| + 3$ zusätzliche Elemente. Zu jedem $v \in V$ gebe es in $\mathcal{S}$ drei disjunkte Mengen S_v^r, S_v^b, S_v^g mit jeweils $|N(v)| + 1$ Elementen. Außerdem enthalte $\mathcal{S}$ für jedes v drei zweielementige Mengen $\{v, e_v^r\}, \{v, e_v^b\}$ und $\{v, e_v^g\}$ mit $e_v^r \in S_v^r$, $e_v^b \in S_v^b$ und $e_v^g \in S_v^g$.

Interpretation: S_v^r entspricht der „Farbe“ r, enthält für jeden Knoten aus $N(v)$ eine Kopie und einen zusätzlichen Knoten e_v^r.

Beispiel:
Abbildung 4.5 zeigt, wie man zu einem Knoten $v \in V$ eines Graphen $G = (V, E)$ die Mengen $S_v^c, c \in C$, konstruiert. ■

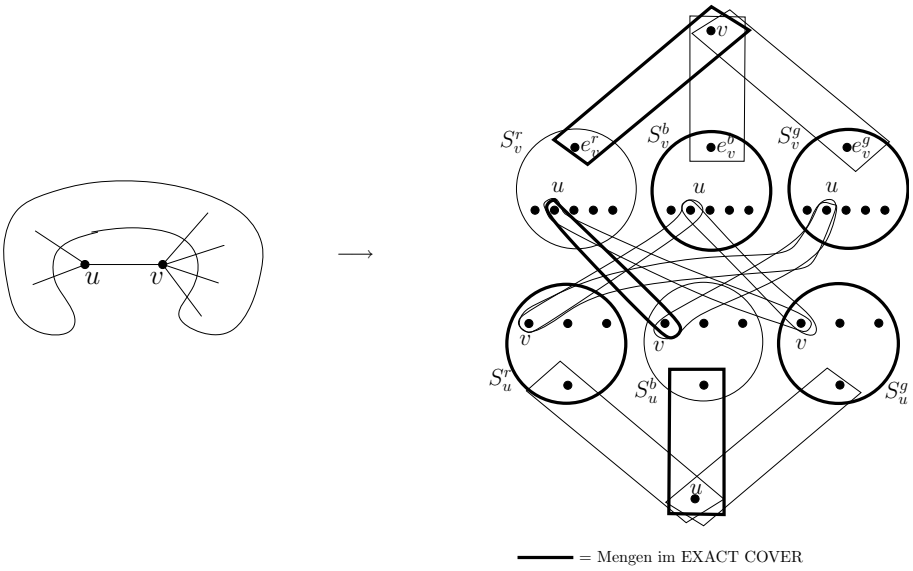


Abbildung 4.6: Konstruktion der Mengen $\{u_v^c, v_u^{c'}\}, c \neq c'$, für eine Kante $\{u, v\} \in E$ eines Graphen $G = (V, E)$.

Außerdem enthält $\mathcal{S}$ für jede Kante $\{u, v\} \in E$ und je zwei $c, c' \in C$, $c \neq c'$, die zweielementigen Mengen $\{u_v^c, v_u^{c'}\}$, $u_v^c \in S_v^c$ „Kopie“ von u, $v_u^{c'} \in S_u^{c'}$ „Kopie“ von v.

Beispiel:
Abbildung 4.6 zeigt, wie man zu einer Kante $\{u, v\} \in E$ eines Graphen $G = (V, E)$ die zweielementigen Mengen $\{u_v^c, v_u^{c'}\}, c \neq c'$, konstruiert. Die Konstruktion ist polynomial in der Größe von G. ■

Zeige zunächst: Falls G dreifärbbar, so existiert eine exakte Überdeckung $\mathcal{S}' \subseteq \mathcal{S}$ für X. Dazu entspreche $\chi : V \to C$ einer zulässigen Dreifärbung.

$\mathcal{S}'$ enthalte für jedes $v \in V$ die Mengen $\{v, e_v^{\chi(v)}\}$ und S_v^c mit $c \neq \chi(v)$. Diese Mengen überdecken alle Elemente exakt, außer den Elementen der Form $u_v^{\chi(v)}$, $v_u^{\chi(u)}$ für $\{u, v\} \in E$. Daher enthalte $\mathcal{S}'$ für jede Kante $\{u, v\} \in E$ die Menge $\{u_v^{\chi(v)}, v_u^{\chi(u)}\}$. Diese Menge existiert, da $\chi(u) \neq \chi(v)$, und damit überdeckt $\mathcal{S}'$ jedes Element aus X genau einmal. (Siehe Abbildung 4.6 für $\chi(u) = b$ und $\chi(v) = r$. Aus Gründen der Übersichtlichkeit sind die Mengen $\{w_v^r, v_w^c\}$ für $c \neq r, \{v, w\} \in E$ bzw. $\{w_u^b, u_w^c\}$ für $c \neq b, \{u, w\} \in E$, welche ebenfalls zu $\mathcal{S}'$ gehören, nicht eingezeichnet.)

Zeige nun: Falls eine exakte Überdeckung $\mathcal{S}' \subseteq \mathcal{S}$ existiert für $(X, \mathcal{S})$, so ist G dreifärbbar.

Sei also $\mathcal{S}'$ eine exakte Überdeckung. Jedes Element v muss von genau einer Menge der Form $\{v, e_v^c\}$ überdeckt sein. Dies induziert eine Färbung χ von G mit den Farben r, b und g. Wir müssen beweisen, dass diese Färbung zulässig ist, d.h. $\chi(v) \neq \chi(u)$ für $\{u, v\} \in E$. Da für jedes v bereits $\{v, e_v^{\chi(v)}\} \in S'$, kann

e_v^c mit $c \neq \chi(v)$ nur durch die Menge S_v^c überdeckt werden. Da die Mengen der Form $\{v, e_v^{\chi(v)}\}$ und S_v^c, $c \neq \chi(v)$, alle Elemente außer den $u_v^{\chi(v)}$ mit $\{u, v\} \in E$ überdecken, müssen auch die Mengen $\{u_v^{\chi(v)}, v_u^{\chi(u)}\}$ für $\{u, v\} \in E$ in $\mathcal{S}'$ enthalten sein. Für diese gilt per Konstruktion $\chi(v) \neq \chi(u)$. □

Problem SUBSET SUM

Gegeben: Eine endliche Menge M, eine Gewichtsfunktion $w : M \to \mathbb{N}_0$ und $K \in \mathbb{N}_0$.

Frage: Existiert eine Teilmenge $M' \subseteq M$ mit

$$\sum_{a \in M'} w(a) = K \ ?$$

4.20 Lemma

SUBSET SUM ist $\mathcal{NP}$–vollständig.

Beweis: Zeigen: EXACT COVER $\propto$ SUBSET SUM.
Sei $(X, \mathcal{S})$ mit $X = \{0, 1, \ldots, m-1\}$ beliebiges Beispiel für EXACT COVER.
Setze $M := \mathcal{S}$ und definiere $w : M \to \mathbb{N}_0$ bzw. K folgendermaßen:
Zu $x \in X$ definiere $\#x := |\{Y \in \mathcal{S} : x \in Y\}|$.
Setze

$$p := \max_{x \in X} \#x + 1,$$

und ordne $Y \in \mathcal{S}$ das Gewicht

$$w(Y) := \sum_{x \in Y} p^x$$

zu. Setze

$$K := \sum_{x=0}^{m-1} p^x.$$

Sei $\mathcal{S}' \subseteq \mathcal{S}$ exakte Überdeckung von $(X, \mathcal{S})$. Dann gilt

$$\sum_{Y \in \mathcal{S}'} w(Y) = \sum_{Y \in \mathcal{S}'} \sum_{x \in Y} p^x = \sum_{x=0}^{m-1} p^x,$$

da jedes $x \in X$ genau einmal überdeckt wird. $\mathcal{S}'$ erfüllt also die Bedingung für SUBSET SUM.

Ist andererseits $\mathcal{S}' \subseteq M = \mathcal{S}$ eine geeignete Menge für SUBSET SUM, so gilt

$$\sum_{Y \in \mathcal{S}'} w(Y) = K = \sum_{x=0}^{m-1} p^x.$$

D.h. jedes $x \in X$ kommt in genau einem $Y \in \mathcal{S}'$ vor. $\mathcal{S}'$ ist also eine exakte Überdeckung. □

Veranschaulichung:
Kodiere $w(Y)$ für $Y \in \mathcal{S}$ als String aus Nullen und Einsen der Länge m, wobei an i-ter Stelle eine 1 steht genau dann, wenn $i \in Y$; entsprechend ist K ein String der Länge m aus Einsen.
Die komponentenweise Addition der zu einer Teilmenge $Y_1, \ldots, Y_n$ von $\mathcal{S}$ gehörigen Strings $w(Y_1), \ldots, w(Y_n)$ ergibt einen String der Länge m, an dessen i-ter Stelle steht in wievielen der $Y_j (j = 1, \ldots, n)$ das Element i vorkommt.
$\sum_{Y \in \mathcal{S}'} w(Y) = K$ bedeutet also, dass jedes $x \in X$ in genau einem $Y \in \mathcal{S}'$ vorkommt.
$\widehat{=}$ Zahlendarstellung zu Basis p.

Problem PARTITION
Gegeben: Eine endliche Menge M und eine Gewichtsfunktion $w : M \to \mathbb{N}_0$.

Frage: Existiert eine Teilmenge $M' \subseteq M$ mit

$$\sum_{a \in M'} w(a) = \sum_{a \in M \setminus M'} w(a) \ ?$$

4.21 Korollar
PARTITION ist $\mathcal{NP}$–vollständig.

Beweis: Zeigen: SUBSET SUM $\propto$ PARTITION.
Zu (M, w, K) Beispiel für SUBSET SUM definiere

$$N := \sum_{a \in M} w(a) + 1,\ M^* := M \cup \{b, c\} \text{ mit } w(b) := N - K \text{ und } w(c) := K + 1.$$

Dann ist (M^*, w) Beispiel für PARTITION und es gilt:

$$\exists M' \subseteq M^* \text{ mit } \sum_{a \in M'} w(a) = \sum_{a \in M^* \setminus M'} (a) \Longleftrightarrow \exists M'' \subseteq M \text{ mit } w(M'') = K.$$

Denn da b und c nicht beide in M' bzw. $M^* \setminus M'$ sein können, kann o.B.d.A. $b \in M'$ angenommen werden. Ist M' eine Menge,welche die linke Seite erfüllt, so muss $w(M') = N$ gelten (da $w(M^*) = 2N$) und $M'' := M' \setminus \{b\}$ erfüllt die Bedingung für SUBSET SUM. Ist andererseits M'' eine Menge, welche die rechte Seite erfüllt, so erfüllt $M' := M'' \cup \{b\}$ die Bedingung für PARTITION. □

Problem KNAPSACK
Gegeben: Eine endliche Menge M, eine Gewichtsfunktion $w : M \to \mathbb{N}_0$, eine Kostenfunktion $c : M \to \mathbb{N}_0$ und $W, C \in \mathbb{N}_0$.

Frage: Existiert eine Teilmenge $M' \subseteq M$ mit

$$\sum_{a \in M'} w(a) \leq W \text{ und}$$

$$\sum_{a \in M'} c(a) \geq C \ ?$$

4.22 Korollar
KNAPSACK ist $\mathcal{NP}$–vollständig.

Beweis: Zeigen: PARTITION $\propto$ KNAPSACK.
Wähle zu (M, w) Beispiel von PARTITION Menge M, Gewichtsfunktion $w' := 2w$, Kostenfunktion $c := 2w$ und $W = C := \sum_{a \in M} w(a)$. □

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.

4.23 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.

4.24 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 $L \in \mathcal{NP}$.

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

Es liegt vermutlich eine Situation wie in Abbildung 4.7 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.

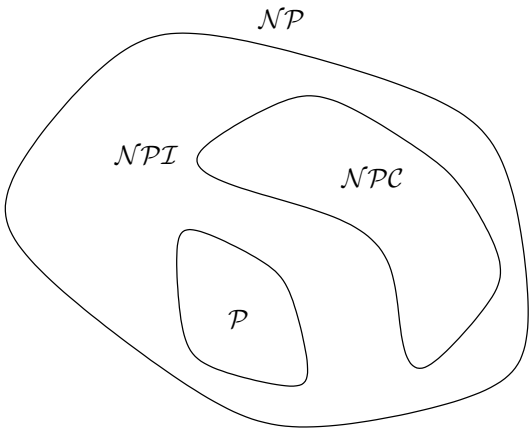


Abbildung 4.7: Komplexitätsklassen

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).

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

Problem co–TSP

Gegeben: Graph $G = (V, E)$, $c\colon E \to \mathbb{Z}^+$ 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“.

4.26 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 $\mathcal{NP} \neq \text{co} - \mathcal{NP}$ 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 $\mathcal{NPI}$.

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).

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

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

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

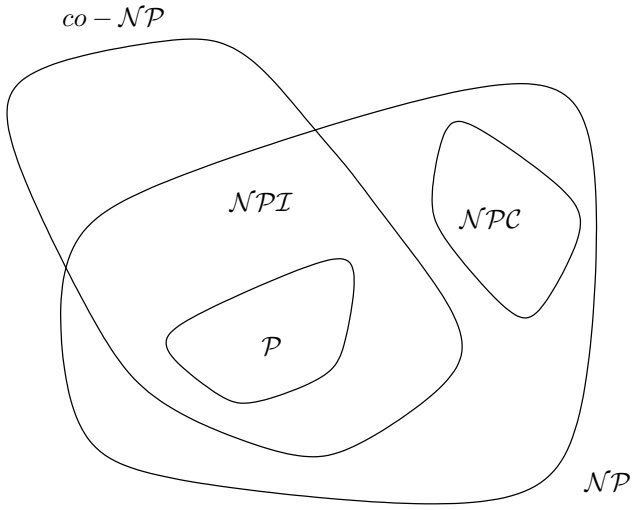


Abbildung 4.8: Komplexitätsklassen

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.

4.27 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

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

4.28 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.

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

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

4.29 Definition
Eine Funktion $f\colon \Sigma^* \to \Sigma^*$ ***realisiert*** *eine Relation* R*, wenn für alle* $x \in \Sigma^*$ *gilt: Falls es kein* $y \in \Sigma^* \backslash \{\varepsilon\}$ *mit* $(x, y) \in R$*, so ist* $f(x) = \varepsilon$*. Ansonsten ist* $f(x) = y$ *für ein* $y \in \Sigma^* \backslash \{\varepsilon\}$ *mit* $(x, y) \in R$*.*
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.

4.30 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 *des 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*.*

4.31 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 $R \propto_T R''$.

4.32 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.

Problem INTEGER PROGRAMMING

Gegeben: $a_{ij} \in \mathbb{N}_0$, $b_i, c_j \in \mathbb{N}_0$, $1 \leq i \leq m$, $1 \leq j \leq n$, $B \in \mathbb{N}_0$.

Frage: Existieren $x_1, \ldots, x_n \in \mathbb{N}_0$, so dass

$$\sum_{j=1}^{n} c_j \cdot x_j = B \text{ und}$$

$$\underbrace{\sum_{j=1}^{n} a_{ij} \cdot x_j \leq b_i \text{ für } 1 \leq i \leq m}_{A \cdot \bar{x} \leq \bar{b}} ?$$

4.33 Korollar
INTEGER PROGRAMMING ist $\mathcal{NP}$–schwer.

Beweis: Zeigen: SUBSET SUM $\propto$ INTEGER PROGRAMMING.
Zu M, $w : M \to \mathbb{N}_0$ und $K \in \mathbb{N}_0$ Beispiel für SUBSET SUM wähle $m = n := |M|$, o.B.d.A. $M = \{1, \ldots, n\}$, $c_j := w(j)$, $B := K$, $b_i = 1$ und $A = (a_{ij})$ Einheitsmatrix. Dann gilt:

$$\exists M' \subseteq M \text{ mit } \sum_{j \in M'} w(j) = K$$

$$\Updownarrow$$

$$\exists x_1, \ldots, x_n \in \mathbb{N}_0 \text{ mit } \sum_{j \in M} w(j) \cdot x_j = B \text{ und } x_j \leq 1 \text{ für } 1 \leq j \leq n.$$

$M' = \{j \in M : x_j = 1\}$ □

Bemerkung:

- INTEGER PROGRAMMING $\in \mathcal{NP}$ ist nicht so leicht zu zeigen. Siehe: Papadimitriou „On the complexity of integer programming“, J.ACM, 28, 2, pp. 765-769, 1981.
- Wie der Beweis von 4.33 zeigt, ist INTEGER PROGRAMMING sogar schon $\mathcal{NP}$–schwer, falls $a_{ij}, b_i \in \{0, 1\}$ und $x_i \in \{0, 1\}$. Es kann sogar unter der Zusatzbedingung $c_{ij} \in \{0, 1\}$ $\mathcal{NP}$–Vollständigkeit gezeigt werden (ZERO-ONE PROGRAMMING).
- Für beliebige lineare Programme ($a_{ij}, c_j, b_i \in \mathbb{Q}$; $x_i \in \mathbb{R}$) existieren polynomiale Algorithmen.
- Lineare Programme spielen eine bedeutende Rolle in der kombinatorischen Optimierung, da sich viele kombinatorische Optimierungsprobleme leicht als lineare Programme formulieren lassen und mit allgemeinen Methoden zur Lösung von linearen Programmen lösen lassen.

4.34 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.
Für das KNAPSACK-Problem gibt es einen pseudopolynomialen Algorithmus:

4.35 Lemma
Ein beliebiges Beispiel (M, w, c, W, C) für KNAPSACK kann in $\mathcal{O}(|M| \cdot W)$ entschieden werden.

Beweis: Sei o.B.d.A. $M = \{1, \ldots, n\}$. Für jedes $w \in N_0, w \leq W$ und $i \in M$ definiere

$$c_i^w := \max_{M' \subseteq \{1,\ldots,i\}} \left\{ \sum_{j \in M'} c(j) : \sum_{j \in M'} w(j) = w \right\}.$$

Dann kann c_{i+1}^w für $0 \leq i < n$ leicht berechnet werden als

$$c_{i+1}^w = \max \left\{ c_i^w, c(i+1) + c_i^{w-w(i+1)} \right\}.$$

□

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

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

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

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 allgemeine KNAPSACK-Suchproblem

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:

4.37 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 $x_1, \ldots, x_n$ mit Wert $\sum_{i=1}^{n} x_i c_i' = \mathcal{A}(I')$, für den gilt:

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

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

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

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 $\mathcal{P} \neq \mathcal{NP}$. □

4.38 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

4.39 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

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

*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 $\mathcal{O}(n \log n)$. ■

4.40 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 für den Greedy-Algorithmus. 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

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

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

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

4.42 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 4.9 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 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, gilt:

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

Andererseits, falls G nicht dreifärbbar ist, gilt

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

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 der Voraussetzung $\mathcal{P} \neq \mathcal{NP}$.

4.43 Satz
Für das TSP mit Dreiecksungleichung (d.h. die Kantengewichte erfüllen die Dreiecksungleichung) existiert ein Approximationsalgorithmus $\mathcal{A}$ mit $\mathcal{R}_{\mathcal{A}} \leq 2$ für alle Instanzen I.

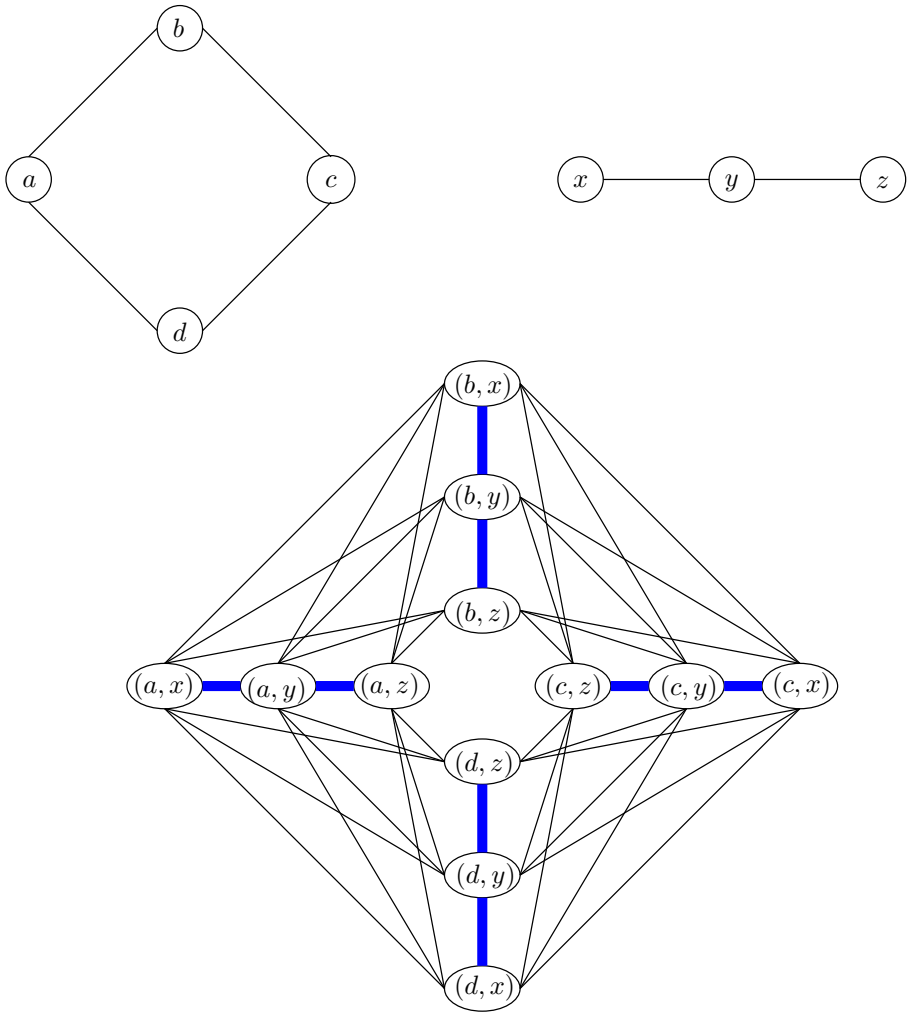
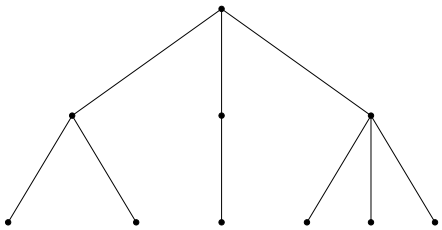
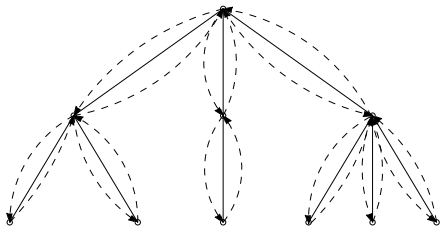


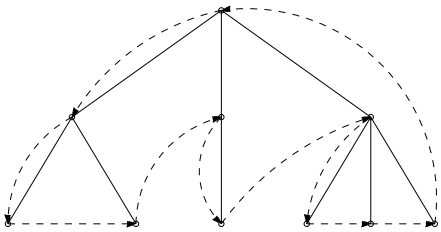
Abbildung 4.9: Graphen G_1, G_2 und $G_1[G_2]$ (von links nach rechts)



(a) MST eines Graphen



(b) Tiefensuch-Tour durch den MST



(c) TSP-Tour als abgekürzte Tiefensuch-Tour

Abbildung 4.10: Konstruktion einer TSP-Tour mit Hilfe eines MST

Beweis: Sei $G = (V, E)$ ein vollständiger Graph, und $c : E \to \mathbb{Z}^+$ eine Gewichtsfunktion auf den Kanten, die die Dreiecksungleichung erfüllt. Betrachte folgenden Algorithmus:

1. Berechne einen Minimum Spanning Tree (MST) von G.

2. Wähle einen beliebigen Knoten w als Wurzel und durchlaufe den Baum in einer Tiefensuche; dies liefert eine Tour T mit Start- und Endpunkt w, die jede Kante zweimal durchläuft.

3. Konstruiere aus T eine Tour T' indem bereits besuchte Knoten übersprungen werden und die Tour beim nächsten unbesuchten Knoten fortgesetzt wird.

Abbildung 4.10 zeigt die Funktionsweise des Algorithmus.

Das Überspringen der bereits besuchten Knoten kann als „Abkürzen“ interpretiert werden, wobei die Dreiecksungleichung gewährleistet, dass T' kürzer oder gleich lang ist wie T. Es gilt also: $c(T') \leq c(T) = 2 \cdot c(MST)$ (wobei $c(T)$ (bzw. $c(MST)$) die Summe der Kantengewichte in T (bzw. im MST) bezeichnet).

Da eine TSP-Tour als ein aufspannender Baum plus eine zusätzliche Kante betrachtet werden kann, gilt: $c(MST) \leq c(\text{OPT})$.

Insgesamt erhält man also:

$$c(T') \leq c(T) = 2 \cdot c(MST) \leq 2 \cdot c(\text{OPT}) \text{ , also } \mathcal{R}_{\mathcal{A}} = \frac{c(T')}{c(\text{OPT})} \leq 2.$$

□

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?

4.44 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.*

4.45 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:

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

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 ein $\mathcal{NP}$–vollständiges Problem (KNAPSACK)

Problem KNAPSACK

Gegeben: Eine Menge von „Teilen“ $M = \{1, \ldots, n\}$, Kosten $c_1, \ldots, c_n \in \mathbb{N}$ 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

$$w_r^j := \min_{M' \subseteq \{1,\ldots,j\}} \left\{ \sum_{i \in M'} w_i \;\middle|\; \sum_{i \in M'} c_i = r \right\}$$

für $r \in \mathbb{N}_0$.

1. Initialisierung:
   für $1 \leq j \leq n$ setze $w_0^j := 0$
   setze $c := \sum_{i=1}^n c_i$
2. solange $w_r^j \leq W$ berechne für $2 \leq j \leq n$und $1 \leq r \leq c$ den Wert
   $$w_r^j = \min \left\{ w_{r-c_j}^{j-1} + w^j, w_r^{j-1} \right. .$$
3. gib
   $$c^* := \max_{1 \leq i \leq n} \left\{ r \mid w_r^j \leq W \right\}$$
   und die entsprechende Menge $M' \subseteq M$ 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 KNAPSACK-Problem.

Betrachte nun das „skalierte“ Problem Π_k zu konstantem k mit $c_i' := \left\lfloor \frac{c_i}{k} \right\rfloor$ 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

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

4.46 Satz
$\mathcal{R}_{\mathcal{A}_\varepsilon}(I) \leq 1 + \varepsilon$ 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 KNAPSACK.

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

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

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

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

Also ist

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

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

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

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

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

□

Die hier benutzte Technik mit Hilfe eines pseudopolynomialen Algorithmus zu einem FPAS zu gelangen, kann auch bei anderen Optimierungsproblemen angewendet werden. Für ein bestimmte Klasse von Optimierungsproblemen kann sogar die umgekehrte Richtung gezeigt werden:

4.47 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* $\mathrm{OPT}(I) \leq q(\langle I \rangle + \max \#(I))$ *(*$\max \#(I)$ *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 4.46. □

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 besteht aus vier Komponenten:*

- *einem endlichen **Alphabet** Σ (auch Terminalalphabet genannt);*

- *einer endlichen Menge V mit $V \cap \Sigma = \emptyset$ von* ***Variablen*** *(auch Nichtterminale genannt);*
- *dem* ***Startsymbol*** *$S \in V$;*
- *einer endlichen Menge von* ***Ableitungsregeln*** *R (auch Produktionen genannt). 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 $w \xrightarrow{*} z$, 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 $S \xrightarrow{*} z$ 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, in der 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 $\sqcup\sqcup\ldots\sqcup q_0 w_1 \ldots w_n\#$ zu $w_1 \ldots w_n$ 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.

Gemäß Übungsaufgabe 7 (Blatt 7) 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 3.13, $L := \{wv \mid v \in L(T_w)\}$), 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:

⇒: 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

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

(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 an, 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 :
$L = \left\{w \in \{0,1\}^* \mid w = w^R\right\}$ 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.

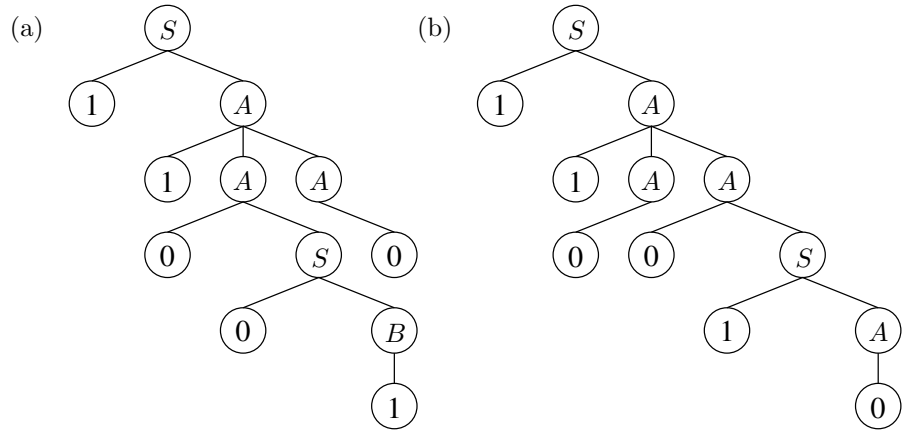


Abbildung 5.1: Syntaxbäume für Ableitungen des Wortes 110010

Zu jeder Ableitung gehört genau ein Syntaxbaum; zu jedem Syntaxbaum gehören jedoch verschiedene Ableitungen des gleichen Wortes. Hier auch:

$$S \to 1A \to 11AA \to 110SA \to 1100BA \to 11001A \to 110010$$

■

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 $A \stackrel{*}{\to} \varepsilon$ existiert: Es werden erst alle A mit $A \to \varepsilon$ aufgenommen. Dann wird geprüft, ob neue Regeln $B \to \varepsilon$ entstehen, wenn man A in allen Regeln auf der rechten Seite A durch ε ersetzt. Ist dies der Fall, so werden die entsprechenden Variablen B in V' aufgenommen und genauso behandelt. (Das Ersetzen von A durch ε wird nicht wirklich durchgeführt, sondern nur „testweise“, um herauszufinden, ob dabei neue ϵ-Regeln entstehen). Am Ende enthält V' alle Variablen A mit $A \stackrel{*}{\to} \varepsilon$. Nun werden alle Regeln $A \to \varepsilon$ gestrichen und für $A \to BC$ wird die zusätzliche Regel $A \to B$ falls

$C \in V'$ beziehungsweise die Regel $A \to C$ falls $B \in V'$ eingeführt. (Die Regel $A \to BC$ wird nicht gestrichen).

Es läßt sich leicht zeigen, dass dadurch 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.:

$$A_1 \to A_2 \to A_3 \to \ldots \to A_r \to A_1$$

Dann ersetzen wir alle $A_2, \ldots, A_r$ in den Regeln durch A_1 und entfernen die Regel $A_1 \to A_1$. Danach sortieren wir die verbleibenden A_i topologisch, sodass die Kettenregeln $A_i \to A_j$ nur existieren, falls $i < j$. Wir haben nun als Variablen $A_1, \ldots, A_m$. Diese werden in der Reihenfolge $A_m, \ldots, A_1$ abgearbeitet, und wenn eine Kettenregel $A_k \to A_\ell$ existiert, für die $A_\ell \to \alpha$ ein Regel ist, so wird sie durch $A_k \to \alpha$ ersetzt, für $\alpha \in (V \cup \Sigma)^+$.

□

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 $w = w_1 \ldots w_n$. Für alle $1 \leq i \leq j \leq n$ soll die Menge $V_{ij} \subseteq V$ berechnet werden, so dass gilt: $A \stackrel{*}{\to} w_i \ldots w_j$ 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 $A \stackrel{*}{\to} w_i$. Da G in Chomsky–Normalform ist, gilt $A \stackrel{*}{\to} w_i$ nur, wenn $(A \to w_i) \in R$. Die Berechnung von V_{ii} ist für alle $i \in \{1, \ldots n\}$ in $\mathcal{O}(|R|)$ möglich.

für $\ell > 0$: Konstruiere die Mengen V_{ij}, d.h. alle $A \in V$ mit $A \stackrel{*}{\to} w_i \ldots w_j$. Da $j - i = \ell > 0$ ist, muss jede Ableitung von $w_i \ldots w_j$ aus A mit einer Regel der Form $A \to BC$ beginnen, wobei ein $k \in \{i, \ldots, j-1\}$ existiert mit $B \stackrel{*}{\to} w_i \ldots w_k$ und $C \stackrel{*}{\to} w_{k+1} \ldots w_j$. V_{ij} läßt sich nun aus den zuvor bestimmten Mengen $V_{ik}, V_{k+1\,j}$ für $k \in \{i, \ldots, j-1\}$ 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 $B \stackrel{*}{\to} w_i \dots w_k$ und $C \stackrel{*}{\to} w_{k+1} \dots w_j$ 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.

Beweis: Sei L kontextfreie Sprache und G Grammatik zu L mit Variablenmenge V in Chomsky-Normalform, d.h. alle Regeln haben die Form $A \to BC$ oder $A \to a$. Setze $n := 2^{|V|+1}$. Wähle beliebiges Wort $z \in L$ mit $|z| \geq n$ und betrachte einen Syntaxbaum zu z. Der Syntaxbaum hat $|z|$ Blätter und wegen der Chomsky-Normalform ist er „im wesentlichen" binär, d.h. alle inneren Knoten außer den Vorgängern der Blätter haben Grad 2, ansonsten Grad 1. Seien mindestens n Blätter markiert. Durchlaufe einen Weg von der Wurzel zu einem Blatt, wobei stets der Nachfolger gewählt wird, auf dessen Seite die größere Anzahl markierter Blätter liegt. Nenne Knoten auf dem Weg, für die rechter und linker Unterbaum markierte Blätter hat, **Verzweigungsknoten**.

Wegen $n > 2^{|V|}$ liegen auf dem Weg mindestens $|V| + 1$ Verzweigungsknoten und von den letzten $|V| + 1$ Verzweigungsknoten entsprechen mindestens zwei Knoten v_1, v_2 derselben Variablen A (siehe Abbildung 5.2). Sei vwx Wort unter Teilbaum mit Wurzel v_1 und w Wort unter Teilbaum mit Wurzel v_2. Damit sind u und y eindeutig bestimmt. Es gilt nun:

- Da v_1 Verzweigungsknoten ist, enthält vx mindestens einen markierten Buchstaben.
- Da der Unterbaum von v_1 inkl. v_1 nur $|V|+1$ Verzweigungsknoten enthält, gibt es in vwx höchstens $2^{|V|+1} = n$ markierte Buchstaben.
- Zu G existieren die Ableitungen

$$S \stackrel{*}{\to} uAy, \qquad A \stackrel{*}{\to} vAx, \qquad A \stackrel{*}{\to} w.$$

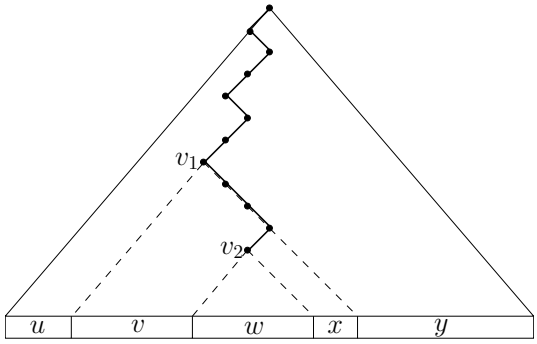


Abbildung 5.2: Weg im Syntaxbaum mit zwei Verzweigungsknoten v_1, v_2, die derselben Variablen entsprechen.

Daraus kann z abgeleitet werden durch

$$S \overset{*}{\to} uAy \overset{*}{\to} uvAxy \overset{*}{\to} uvwxy = z,$$

aber auch uv^iwx^iy für jedes $i \geq 1$ durch

$$S \overset{*}{\to} uAy \overset{*}{\to} uvAxy \overset{*}{\to} uv^2Ax^2y \overset{*}{\to} \cdots \to uv^iAx^iy \to iv^iwx^iy.$$

Also ist auch $uv^iwx^iy \in L$ für $i \geq 0$.

□

Bemerkung:
Der Spezialfall von Odgen's Lemma, in dem alle Buchstaben von z markiert sind, ist gerade das Pumping–Lemma.

5.15 Satz
Die Chomsky-Hierarchie ist echt, d.h. $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$, *wobei* $\mathcal{L}_i, 0 \leq i \leq 3$, *Klasse der durch Typ-i-Grammatiken erzeugten Sprachen.*

Beweis: Zu zeigen ist:

(i) Es gibt eine kontextfreie Sprache, die nicht regulär ist.

(ii) Es gibt eine kontextsensitive Sprache, die nicht kontextfrei ist.

(iii) Es gibt eine semi-entscheidbare Sprache, die nicht kontextsensitiv ist.

Zu (i): $L = \{a^ib^i | i \geq 1\}$ ist kontextfrei (siehe Beispiel nach Satz 2.15), aber nicht regulär (siehe Beispiel am Anfang von Kap.5.4).

Zu (ii): $L = \{a^ib^ic^i | i \geq 1\}$ ist kontextsensitiv (siehe Übung 11, Aufgabe 2), da L durch eine DTM mit Speicherbedarf n entschieden werden kann. L ist aber nicht kontextfrei (siehe Übung 13, Aufgabe 6).

Zu (iii): Die universelle Sprache L_u ist nicht kontextsensitiv, da L_u nicht entscheidbar ist. Es gilt aber für jede Sprache, die durch eine NTM mit linearem

Speicher entschieden werden kann, dass sie durch eine DTM mit exponentieller Laufzeit entschieden werden kann. Denn mit linearem Arbeitsband können nur exponentiell viele verschiedene Konfigurationen eintreten. Man kann also mit exponentiellem Aufwand testen, ob für eine feste Anfangskonfiguration eine akzeptierende Endkonfiguration erreicht werden kann. □

5.16 Definition
Sei G eine kontextfreie Grammatik. Eine Variable A heißt ***nutzlos****, falls es keine Ableitung $S \stackrel{*}{\to} w$ gibt, $w \in \Sigma^*$, in der A vorkommt.*

5.17 Satz
Für eine kontextfreie Grammatik kann die Menge der nutzlosen Variablen (in polynomialer Zeit) berechnet werden.

Beweis: Wir berechnen die Menge der nutzlosen Variablen in zwei Schritten.

Schritt 1: Berechnung von $V' \subseteq V$ mit $A \in V'$ genau dann wenn es $w \in \Sigma^*$ gibt mit $A \stackrel{*}{\to} w$.

Füge zunächst alle $A \in V$ mit $A \to w$ für ein $w \in \Sigma^*$ in eine Queue Q sowie in V' ein. Entferne der Reihe nach alle Elemente aus Q; für A aktuelles Element aus Q ersetze jede Regel $B \to \alpha A \beta$ mit $\alpha, \beta \in (V \cup \Sigma)^*$ durch die Regeln $B \to \alpha w \beta$, wobei $w \in \Sigma^*$ und $A \to w$ Regel. Wenn dabei eine Regel der Form $B \to w', w' \in \Sigma^*$, entsteht, füge B in Q und V' ein. Das Verfahren endet, wenn Q leer ist. Per Induktion über die Länge der kürzesten Ableitungsregel der Form $A \stackrel{*}{\to} w$ kann für A gezeigt werden, dass $A \in V'$.
Falls $S \notin V'$, breche das Verfahren ab. G erzeugt dann die leere Sprache und alle Variablen sind nutzlos.

Schritt 2: Berechnung von $V'' \subseteq V'$ aller $A \in V'$, für die es $\alpha, \beta \in (V' \cup \Sigma)^*$ gibt, sodass eine Ableitung $S \stackrel{*}{\to} \alpha A \beta$ existiert, oder $A = S$.

Füge zu allen Regeln $S \to \alpha A \beta$ mit $\alpha, \beta \in (V' \cup \Sigma)^*, A \in V'$, A in V'' ein. Für jedes in V'' eingefügte A mit $A \to \alpha B \beta$, $\alpha, \beta \in (V' \cup \Sigma)^*$ und $B \in V'$ füge auch B in V'' ein usw.

Per Induktion über die Länge der kürzesten Ableitungsregel der Form $S \to \alpha A \beta$, $\alpha, \beta \in (V' \cup \Sigma)^*$, kann dann wieder die Korrektheit bewiesen werden.

Es gilt insgesamt, dass V'' die Menge aller nützlichen Variablen ist. □

5.18 Korollar
Für eine kontextfreie Grammatik G kann (in polynomialer Zeit) entschieden werden, ob $L(G) = \emptyset$ ist.

Beweis: $L(G) = \emptyset$ genau dann, wenn S nutzlos. □

5.19 Satz
Für eine kontextfreie Grammatik G kann (in polynomialer Zeit) entschieden werden, ob $L(G)$ endlich ist.

Beweis: Entferne zunächst alle nutzlosen Variablen und überführe G in eine äquivalente Grammatik in Chomsky-Normalform. Betrachte dann den gerichteten Graphen, der für jede Variable einen Knoten und für Variablen A, B die Kante (A, B) genau dann enthält, wenn es eine Regel der Form $A \to BC$ oder $A \to CB$ gibt. Mit Tiefensuche kann entschieden werden, ob dieser Graph einen Kreis enthält. Man kann sich leicht überlegen, dass $L(G)$ genau dann endlich ist, wenn der entsprechende Graph keinen Kreis enthält. □

5.20 Satz
Die Klasse der kontextfreien Sprachen ist abgeschlossen bzgl. Vereinigung, Konkatenation und Kleenschem Abschluss.

Beweis: Seien $L(G_1), L(G_2)$ kontextfreie Sprachen für die o.B.d.A. G_1 und G_2 disjunkte Variablenmengen haben. Dann gibt es eine kontextfreie Grammatik, die $L(G_1) \cup L(G_2)$ erzeugt: Setze dazu $V = V_1 \cup V_2 \cup \{S\}$, S neues Startsymbol, $R = R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}$.

Eine kontextfreie Grammatik, die $L(G_1) \cdot L(G_2)$ erzeugt, besteht aus $V = V_1 \cup V_2 \cup \{S\}$ und $R = R_1 \cup R_2 \cup \{S \to S_1 S_2\}$.

Eine kontextfreie Grammatik für $L(G_1)^*$ besteht aus $V = V_1 \cup \{S\}$ und $R = R_1 \cup \{S \to \varepsilon, S \to SS, S \to S_1\}$. □

5.21 Satz
Die Klasse der kontextfreien Sprachen ist nicht abgeschlossen bzgl. Komplementbildung und Durchschnitt.

Beweis: Betrachte $L_1 = \{a^n b^n | n \geq 1\}$, $L_2 = \{c\}^*$, $L_3 = \{a\}^*$, $L_4 = \{b^n c^n | n \geq 1\}$. Alle diese Sprachen sind kontextfrei. Nach Satz 5.20 sind dann auch $L_1 \cdot L_2$ und $L_3 \cdot L_4$ kontextfrei. $L = L_1 L_2 \cap L_3 L_4$ ist dann gerade $L = \{a^n b^n c^n | n \geq 1\}$. Diese Sprache ist nicht kontextfrei.

Angenommen, die Klasse der kontextfreien Sprachen wäre bzgl. Komplementbildung abgeschlossen. Dann würde für beliebige kontextfreie Sprachen L_1, L_2 gelten $(L_1^c \cup L_2^c)^c = L_1 \cap L_2$ ist wieder kontextfrei. Dies ist ein Widerspruch zur ersten Aussage des Satzes. □

5.5 Kontextfreie Sprachen und Kellerautomaten

5.22 Definition
Eine kontextfreie Grammatik ist in ***Greibach-Normalform****, wenn alle Ableitungsregeln von der Form*

$$A \to a\alpha \text{ mit } A \in V, a \in \Sigma \text{ und } \alpha \in V^*$$

sind.

5.23 Satz
Für jede kontextfreie Grammatik G, für die $L(G)$ das leere Wort nicht enthält, kann eine äquivalente kontextfreie Grammatik G'(d.h. $L(G) = L(G')$) in Greibach-Normalform konstruiert werden.

Beweis: Folgende Ersetzungen von Regeln können immer vorgenommen werden, ohne dass sich die von der entsprechenden Grammatik erzeugte Sprache ändert.

(i) Eine Regel $A \to \alpha_1 B \alpha_2$ mit $B \to \beta_1, B \to \beta_2, \ldots, B \to \beta_r$ alle Regeln, deren linke Seite B ist, kann durch die Regeln $A \to \alpha_1\beta_1\alpha_2, A \to \alpha_1\beta_2\alpha_2, \ldots, A \to \alpha_1\beta_r\alpha_2$ ersetzt werden.

(ii) Seien $A \to A\alpha_1, \ldots, A \to A\alpha_r$ und $A \to \beta_1, \ldots, A \to \beta_s$, wobei β_i nicht mit A beginnen, alle Regeln, deren linke Seite A ist. Dann können die Regeln $A \to A\alpha_1, \ldots, A \to A\alpha_r$ durch die Regeln $A \to \beta_1 B, \ldots, A \to \beta_s B, B \to \alpha_1, \ldots, B \to \alpha_r, B \to \alpha_1 B, \ldots, B \to \alpha_r B$ ersetzt werden. Dabei sei B eine neu eingeführte Variable.

Wir gehen nun davon aus, dass G in Chomsky-Normalform ist. Es sei $V = \{A_1, \ldots, A_m\}$ und $\Sigma = \{a_1, \ldots, a_n\}$, d.h. alle Regeln sind von der Form $A_i \to A_j A_k$ oder $A_i \to a_j$. Die Grammatik in Greibach-Normalform wird zusätzlich die Regeln $\{B_1, \ldots, B_m\}$ benutzen. Sei $V' := \{A_1, \ldots, A_m, B_1, \ldots, B_m\}$. Die Variablenmenge zur Umformung von G in G' wird die folgenden Invarianten erfüllen:

1.Invariante: Die rechte Seite einer Regel, deren rechte Seite mit einer Variablen beginnt, besteht nur aus Variablen.

2.Invariante: Die rechte Seite einer Regel, deren rechte Seite mit einer Variablen beginnt, beginnt mit einer Variablen aus $V = \{A_1, \ldots, A_m\}$.

3.Invariante: Symbole aus Σ kommen nur als erstes Zeichen der rechten Seite einer Regel vor.

4.Invariante: Die rechte Seite einer Regel, deren linke Seite aus $V = \{A_1, \ldots, A_m\}$ ist und deren rechte Seite mit einer Variablen aus V beginnt, beginnt sogar mit zwei Variablen aus V.

5.Invariante: Die rechte Seite einer Regel, deren linke Seite aus $V' \backslash V = \{B_1, \ldots, B_m\}$ ist, besteht nur aus Variablen aus V'.

Offensichtlich erfüllt G alle fünf Invarianten. Wir formen G zunächst so um, dass außer Invarianten 1-5 noch die

6.Invariante: Falls $A_i \to A_j\alpha$ Regel ist, so gilt $j > i$.

erfüllt ist. Dabei wenden wir (in dieser Reihenfolge)

(ii) zur Ersetzung aller Regeln $A_1 \to A_1\alpha$
(i) zur Ersetzung aller Regeln $A_2 \to A_1\alpha$
(ii) zur Ersetzung aller Regeln $A_2 \to A_2\alpha$
(i) zur Ersetzung aller Regeln $A_3 \to A_1\alpha$
(i) zur Ersetzung aller Regeln $A_3 \to A_2\alpha$
(ii) zur Ersetzung aller Regeln $A_3 \to A_3\alpha$
$\vdots$

usw. an. Es wird also $\binom{m}{2}$-mal (i) und m-mal (ii) angewendet.
Es ist leicht zu sehen, dass die so umgeformte Grammatik alle Invarianten erfüllt. Danach sind alle Regeln von der Form $A \to a\alpha$ oder $A \to \alpha$ mit $\alpha \in (V')^*, a \in \Sigma$, wobei es keine ε-Regeln oder Kettenregeln mit linker Seite A gibt.
Wegen der 6.Invarianten gibt es keine Regel $A_m \to \alpha$ und alle $A_{m-1} \to \alpha$-Regeln beginnen mit A_m. Diese Regeln ersetzen wir nun nach Methode (i) und fahren so für absteigendes $k, k = m-2, \ldots, k = 1$, fort. (Wenn wir die Regeln $A_k \to \alpha$ betrachten, so beginnt α mit einem A_j, für das $j > k$ gilt, für das die rechte Seite also schon die gewünschte Form hat.)
Danach sind alle Regeln mit linker Seite aus V in der Form $A_k \to a\alpha, a \in \Sigma$.
Wegen der zweiten und der fünften Invarianten beginnen alle rechten Seiten von Regeln, deren linke Seite aus $V'\backslash V = \{B_1, \ldots, B_m\}$ ist, mit einer Variablen aus $\{A_1, \ldots, A_m\}$. Ersetze diese mit (i). Damit erhalten wir G' in Greibach-Normalform. □

Beispiel :
$V = \{A_1, A_2, A_3\}$, $\Sigma = \{0, 1\}$, $S = A_1$ und
$R = \{A_1 \to A_2A_3,\ A_2 \to A_3A_1,\ A_2 \to 1,\ A_3 \to A_1A_2,\ A_3 \to 0\}$.

Zunächst wird $A_3 \to A_1A_2$ ersetzt durch $A_3 \to A_2A_3A_2$, dann durch $A_3 \to A_3A_1A_3A_2$ und $A_3 \to 1A_3A_2$.
$A_3 \to A_3A_1A_3A_2$ wird ersetzt durch $A_3 \to 0B_3,\ A_3 \to 1A_3A_2B_3,\ B_3 \to A_1A_3A_2$ und $B_3 \to A_1A_3A_2B_3$.
Nun haben alle Regeln mit linker Seite A_3 die gewünschte Form.

Es wird nun $A_2 \to A_3A_1$ ersetzt durch $A_2 \to 0B_3A_1$, $A_2 \to 1A_3A_2A_1$, $A_2 \to 0A_1$ und $A_2 \to 1A_3A_2B_3A_1$.
Dann wird $A_1 \to A_2A_3$ ersetzt durch $A_1 \to 1A_3,\ A_1 \to 0B_3A_1A_3,\ A_1 \to 1A_3A_2A_1A_3,\ A_1 \to 0A_1A_3$ und $A_1 \to 1A_3A_2B_3A_1A_3$.

Schließlich werden $B_3 \to A_1A_3A_2$ und $B_3 \to A_1A_3A_2B_3$ ersetzt durch $B_3 \to 1A_3A_3A_2,\ B_3 \to 0B_3A_1A_3A_3A_2,\ B_3 \to 1A_3A_2A_1A_3A_3A_2,\ B_3 \to 0A_1A_3A_3A_2,$ $B_3 \to 1A_3A_2B_3A_1A_3A_3A_2$ und $B_3 \to 1A_3A_3A_2B_3,\ B_3 \to 0B_3A_1A_3A_3A_2B_3,$ $B_3 \to 1A_3A_2A_1A_3A_3A_2B_3,\ B_3 \to 0A_1A_3A_3A_2B_3,\ B_3 \to 1A_3A_2B_3A_1A_3A_3A_2B_3$.

Die neue Grammatik hat also 24 Regeln. ■

5.24 Definition
Ein (nichtdeterministischer) ***Kellerautomat*** *(NPDA bzw. PDA) besteht aus* $(Q, \Sigma, \Gamma, q_0, Z_0, \delta, F)$*, wobei*

- Q *endliche Zustandsmenge*
- Σ *endliches Eingabealphabet*
- Γ *endliches STACK-Alphabet*

- $q_0 \in Q$ *Anfangszustand*
- $Z_0 \in \Gamma$ *Initialisierung des STACK*
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to 2^{Q \times \Gamma^*}$, *d.h.* $\delta(q, a, Z) \subseteq \{(q, \gamma) : q \in Q, \gamma \in \Gamma^*\}$ *und* $\delta(q, \varepsilon, Z) \subseteq \{(q, \gamma) : q \in Q, \gamma \in \Gamma^*\}$
- $F \subseteq Q$ *Menge der akzeptierenden Endzustände,* $F = \emptyset$ *ist möglich.*

Eine ***Konfiguration eines PDA*** *ist ein Tripel* (q, w, α) *mit* $q \in Q$, $\alpha \in \Gamma^*$ *STACK-Inhalt und* $w \in \Sigma^*$ *der Teil der Eingabe, der noch nicht gelesen wurde. Zu* $(q, w_1 \dots w_k, Z_1 \dots Z_m)$ *gibt es folgende* ***Nachfolgekonfigurationen****:*

$$(q', w_2 \dots w_k, Z_1' \dots Z_r' Z_2 \dots Z_m) \text{ für alle } (q', Z_1' \dots Z_r') \in \delta(q, w_1, Z_1)$$

und

$$(q', w_1 \dots w_k, Z_1' \dots Z_r' Z_2 \dots Z_m) \text{ für alle } (q', Z_1' \dots Z_r') \in \delta(q, \varepsilon, Z_1).$$

Ein PDA ***akzeptiert*** *ein* $w \in \Sigma^*$ ***durch leeren Stack****, wenn es eine zulässige Folge von Konfigurationen aus der Anfangskonfiguration* (q_0, w, Z_0) *in eine Konfiguration* $(q, \varepsilon, \varepsilon), q \in Q$, *gibt.*

Ein PDA ***akzeptiert*** *ein* $w \in \Sigma^*$ ***durch einen akzeptierenden Endzustand****, wenn es eine zulässige Folge von Konfigurationen aus der Anfangskonfiguration* (q_0, w, Z_0) *in eine Konfiguration* (q, ε, γ) *mit* $q \in F$ *und* $\gamma \in \Gamma^*$ *gibt.*

Ein PDA ist ***deterministisch****, falls*

$$|\delta(q, a, Z)| + |\delta(q, \varepsilon, Z)| \leq 1$$

für alle $q \in Q, a \in \Sigma, Z \in \Gamma$.

Beispiel :
Ein DPDA für $L = \{w\#w^R | w \in \{0,1\}^*\}$.
Betrachte beliebiges Wort $w_1 \dots w_n \# w_n \dots w_1 \in L$. Der DPDA liest $w_1 \dots w_n$ und schreibt jeweils w_i auf den STACK. Beim Lesen von # geht der DPDA dazu über, $w_n \dots w_1$ zu lesen und den gelesenen Buchstaben jeweils mit dem obersten Buchstaben auf dem STACK zu vergleichen. Bei Gleichheit wird der oberster Buchstabe vom STACK genommen, ansonsten wird in einen nichtakzeptierenden Zustand gegangen und gestoppt. Ist nur noch Z_0 auf dem STACK, wird Z_0 entfernt und die Eingabe „mit leerem STACK“ akzeptiert.

Eingabe: *abc#cba*			STACK								
$\delta(q_0, a, Z_0)$	=	(q_1, aZ_0)									
$\delta(q_1, b, a)$	=	(q_1, ba)									
$\delta(q_1, c, b)$	=	(q_1, cb)				c	c				
$\delta(q_1, \#, c)$	=	(q_2, c)			b	b	b	b			
$\delta(q_2, c, c)$	=	(q_2, ε)		a	a	a	a	a	a		
$\delta(q_2, b, b)$	=	(q_2, ε)	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	Z_0	—
$\delta(q_2, a, a)$	=	(q_2, ε)									
$\delta(q_2, \varepsilon, Z_0)$	=	(q_2, ε)									

■

Beispiel :
PDA, der $L = \{ww^R | w \in \{0,1\}^*\}$ akzeptiert.
PDA ist nichtdeterministisch, indem er wie DPDA aus letztem Beispiel arbeitet, bis auf den Wechsel von „Lesezustand“ q_1 in „Vergleichszustand“ q_2. Dies tut er nichtdeterministisch. ■

Bemerkung:
Für die Sprache $L = \{ww^R | w \in \{0,1\}^*\}$ gibt es keinen DPDA. NPDAs können also mehr als DPDAs.

5.25 Satz
Zu einem PDA, der eine Sprache L durch einen akzeptierenden Endzustand akzeptiert, kann ein PDA konstruiert werden, der L mit leerem STACK akzeptiert.

Beweis: Sei $\mathcal{A}_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_0^1, Z_0^1, F_1)$ PDA, der L durch Übergang in einen Zustand aus F_1 akzeptiert. Wir konstruieren dazu einen PDA $\mathcal{A}_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_0^2, Z_0^2)$, der dieselbe Sprache L durch leeren STACK akzeptiert. Dazu muss $\mathcal{A}_2$ in der Lage sein, den STACK zu leeren, wenn $\mathcal{A}_1$ in einen Zustand aus F_1 kommt. Diese Übergänge dürfen jedoch nicht zur Akzeptanz eines Wortes $w \notin L$ führen. Andererseits kann $\mathcal{A}_1$ zu einem leeren STACK führen, ohne dass die entsprechende Eingabe akzeptiert wird. In diesem Fall darf $\mathcal{A}_2$ keinen leeren STACK erzeugen. Die Konstruktion von $\mathcal{A}_2$ beruht entsprechend auf der Einführung eines neuen Zustands q_E, bei dessen Erreichen der STACK geleert wird, und der Einführung eines neuen STACK-Symbols Z_0^2, das zu Beginn auf den STACK gelegt wird, um einen Übergang in einen leeren STACK, in dem das gelesene Wort nicht akzeptiert werden soll, zu vermeiden.
Definiere entsprechend

$$\begin{aligned} Q_2 &:= Q_1 \cup \{q_0^2, q_E\}, \quad q_0^2 \text{ Anfangszustand von } \mathcal{A}_2 \\ \Gamma_2 &:= \Gamma_1 \cup \{Z_0^2\}, \quad Z_0^2 \text{ Initialisierung des STACKs von } \mathcal{A}_2 \end{aligned}$$

Die Menge $\delta_2(q, a, Z)$ für $a \in \Sigma \cup \{\varepsilon\}$ und $Z \in \Gamma_2$ sei durch folgende Bedingungen festgelegt:

$$\begin{aligned} \delta_2(q_0^2, \varepsilon, Z_0^2) &= \{(q_0^1, Z_0^1 Z_0^2)\} \\ \delta_2(q, a, Z) &= \delta_1(q, a, Z) \text{ für } q \in Q_1,\ a \neq \varepsilon,\ Z \in \Gamma_1 \\ &\qquad \text{oder } q \in Q_1 \backslash F_1,\ a = \varepsilon,\ Z \in \Gamma_1 \\ \delta_2(q, \varepsilon, Z) &= \delta_1(q, \varepsilon, Z) \cup \{(q_E, \varepsilon)\} \text{ für } q \in F_1,\ Z \in \Gamma_2 \\ \delta_2(q_E, \varepsilon, Z) &= \{(q_E, \varepsilon)\} \text{ für } Z \in \Gamma_2 \\ \delta(\cdot) &= \emptyset \text{ sonst} \end{aligned}$$

$\mathcal{A}_2$ akzeptiert genau $w \in L$ mit leerem STACK, da Z_0^2 nur im Zustand q_E vom STACK entfernt werden kann und q_E nur aus Zuständen aus F_1 erreicht werden kann. □

5.26 Satz
Zu einem PDA, der eine Sprache L mit leerem STACK akzeptiert, kann ein PDA konstruiert werden, der L durch einen akzeptierenden Endzustand akzeptiert.

Beweis: Ähnlich wie im Beweis zu Satz 5.25 konstruieren wir zu einem PDA $\mathcal{A}_1 = (Q_1, \Sigma, \Gamma_1, \delta_1, q_0^1, Z_0^1)$, der $w \in L$ mit leerem STACK akzeptiert, einen

PDA $\mathcal{A}_2 = (Q_2, \Sigma, \Gamma_2, \delta_2, q_0^2, Z_0^2, F_2)$, der genau die $w \in L$ durch Übergang in einen Zustand $q \in F_2$ akzeptiert. Dazu wird auf den STACK Z_0^2 gelegt, und nur gelöscht, wenn die Abarbeitung von $\mathcal{A}_1$ mit leerem STACK beendet worden wäre. In diesem Schritt wird in einen neu eingeführten Endzustand q_F gegangen. Definiere

$$\begin{aligned} Q_2 &:= Q_1 \cup \{q_0^2, q_F\}, \text{ wobei } q_0^2 \text{ Anfangszustand von } \mathcal{A}_2 \text{ und } F_2 := \{q_F\} \\ \Gamma_2 &:= \Gamma_1 \cup \{Z_0^2\}, \text{ wobei } Z_0^2 \text{ Initialisierung des STACK von } \mathcal{A}_2, \end{aligned}$$

und δ_2 sei festgelegt durch

$$\begin{aligned} \delta_2(q_0^2, a, X) &= \begin{cases} \{q_0^1, Z_0^1 Z_0^2\} & \text{falls } a = \varepsilon \text{ und } X = Z_0^2 \\ \emptyset & \text{sonst} \end{cases} \\ \delta_2(q, a, Z) &= \delta_1(q, a, Z), \text{ falls } q \in Q_1, a \in \Sigma \cup \{\varepsilon\} \text{ und } Z \in \Gamma_1 \\ \delta_2(q, \varepsilon, Z_0^2) &= \{(q_F, \varepsilon)\} \text{ für } q \in Q_1. \end{aligned}$$

Offensichtlich akzeptiert $\mathcal{A}_2$ genau die $w \in L$. □

5.27 Satz
Für eine Grammatik G in Greibach-Normalform kann ein PDA konstruiert werden, der $L(G)$ mit leerem STACK akzeptiert.

Beweis: Wir geben zur Grammatik G mit Variablenmenge V und Startsymbol S, Regelmenge R und Alphabet Σ einen PDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ an, der genau bei Eingabe $w \in L$ in eine Konfiguration mit leerem STACK überführt. Setze $Q := \{q_0\}$, $\Gamma := V$, $Z_0 := S$ und definiere

$$\delta(q_0, a, A) := \{(q_0, \alpha) | (A \to a\alpha) \in R\}.$$

Per Induktion über die Länge i einer Ableitung beweisen wir, dass
$S \overset{*}{\to} w_1 \dots w_i A_1 \dots A_m$ genau dann, wenn $\mathcal{A}$ beim Lesen von $w_1 \dots w_i$ den STACK-Inhalt $A_1 \dots A_m$ erzeugen kann. Daraus folgt, dass $\mathcal{A}$ das Wort $w_1 \dots w_n$ mit leerem STACK genau dann erkennt, wenn $S \overset{*}{\to} w_1 \dots w_n$ in G existiert, d.h. $w_1 \dots w_n \in L(G)$.

Induktionsanfang ist mit $i = 0$ trivialerweise erfüllt.

Induktionsschritt: Sei $i \geq 1$ und „$\overset{j}{\to}$“ stehe für eine Ableitung der Länge j. Dann gilt

$$S \overset{i}{\to} w_1 \dots w_i A_1 \dots A_m \iff \begin{array}{l} \exists A' \in V, r \in \{1, \dots, m\} \text{ mit} \\ S \overset{i-1}{\longrightarrow} w_1 \dots w_{i-1} A' A_r \dots A_m \\ \to w_1 \dots w_i A_1 \dots A_m. \end{array}$$

Mit der Induktionsvoraussetzung ist dies genau dann der Fall, wenn gilt:
$\exists A' \in V, r \in \{1, \dots, m\}$ so, dass $\mathcal{A}$ das Wort $w_1 \dots w_{i-1}$ lesen und dabei STACK-Inhalt $A' A_r \dots A_m$ erzeugen kann und $A' \to w_i A_1 \dots A_{r-1}$ Regel von G ist.
Nach der Definition eines PDA ist dies genau dann erfüllt, wenn $\mathcal{A}$ das Wort $w_1 \dots w_i$ lesen und dabei den STACK-Inhalt $A_1 \dots A_m$ erzeugen kann. □

5.28 Satz
Jede durch einen PDA (mit leerem STACK oder durch akzeptierende Endzustände) akzeptierte Sprache ist kontextfrei.

Beweis: Betrachte PDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, der $L_{\mathcal{A}}$ durch leeren STACK akzeptiert. Wir geben eine Grammatik G mit Variablenmenge V, Startsymbol S und Regelmenge R an, für die $L_{\mathcal{A}} = L(G)$ ist. G wird „fast“ in Greibach-Normalform sein. Die Konstruktion von G heißt **Tripelkonstruktion**. Setze $V := \{[q, X, p] | p, q \in Q,\ X \in \Gamma\} \cup \{S\}$, wobei S Startsymbol von G sei. Entsprechend sollen aus einer Variable $[q, X, p]$ genau die $w \in \Sigma^*$ ableitbar sein, für die es eine Abarbeitung von $\mathcal{A}$ gibt, die im Zustand q mit oberstem STACK-Symbol X beginnt und nach Lesen von w im Zustand p mit leerem STACK endet. Die Regelmenge R enthalte dementsprechend folgende Regeln:

(i) $S \to [q_0, Z_0, q]$ für alle $q \in Q$

(ii) $[q, X, q_{m+1}] \to a[q_1, Y_1, q_2] \dots [q_m, Y_m, q_{m+1}]$ für alle $q_2, \dots, q_{m+1} \in Q$, falls $(q_1, Y_1 \dots Y_m) \in \delta(q, a, X)$.

Wir werden per Induktion beweisen, dass für alle $p, q \in Q,\ X \in \Gamma$ und $w \in L$ gilt:

$$[q, X, p] \overset{*}{\to} w \text{ in } G \iff \begin{array}{l}\text{es gibt in } \mathcal{A} \text{ eine Folge von} \\ \text{Konfigurationen von } (q, w, X) \text{ nach } (p, \varepsilon, \varepsilon)\end{array} \tag{5.1}$$

Für eine Folge von Konfigurationen (bzw. k Konfigurationen) (q, w, X) nach (p, w', Y) schreiben wir auch „$(q, w, X) \overset{*}{\vdash} (p, w', Y)$“ $\left(\text{bzw. } (q, w, X) \overset{k}{\vdash} (p, w', Y)\right)$.
Aus dieser Behauptung folgt dann

$$\begin{aligned} w \in L_{\mathcal{A}} &\iff \begin{array}{l}\exists p \in Q \text{ mit } (q_0, w, Z_0) \overset{*}{\vdash} (p, \varepsilon, \varepsilon), \text{ wobei} \\ (q_0, w, Z_0) \text{ Anfangskonfiguration von } \mathcal{A} \text{ ist}\end{array} \\ &\iff \exists p \in Q \text{ mit } [q_0, Z_0, p] \overset{*}{\to} w \\ &\iff \exists p \in Q \text{ mit } S \to [q_0, Z_0, p] \overset{*}{\to} w \\ &\iff w \in L(G) \end{aligned}$$

Beweis von (5.1):
„$\Rightarrow$“
Wir zeigen zunächst per Induktion über die Länge k einer Ableitung $[q, X, p] \overset{k}{\to} w$ in G, dass es in $\mathcal{A}$ eine Abarbeitung $(q, w, X) \overset{*}{\vdash} (p, \varepsilon, \varepsilon)$ gibt.

Induktionsanfang: Für $k = 1$ gilt, dass $[q, X, p] \to w$ eine Regel in G ist, also ist $(p, \varepsilon) \in \delta(q, w, X)$ und $|w| \leq 1$. Also gibt es die Abarbeitung
$(q, w, X) \overset{1}{\vdash} (p, \varepsilon, \varepsilon)$ in $\mathcal{A}$.

Induktionsschritt: Betrachte eine Ableitung $[q, X, p] \overset{k}{\vdash} w$. Dann kann diese Ableitung geschrieben werden als

$$[q, X, p] \to a[q_1, Y_1, q_2][q_2, Y_2, q_3] \dots [q_m, Y_m, q_{m+1}] \overset{k-1}{\vdash} w,$$

wobei $q_{m+1} = p$.
Entsprechend lässt sich w schreiben als $w = aw_1 \dots w_m$, mit $w_i \in \Sigma^*$, $a \in \Sigma$ und $[q_j, Y_j, q_{j+1}] \overset{k'}{\to} w_j$ mit $k' \leq k-1$ für alle $1 \leq j \leq m$. Nach Induktionsvoraussetzung gilt dann $(q_j, w_j, Y_j) \overset{*}{\vdash} (q_{j+1}, \varepsilon, \varepsilon)$ für alle $1 \leq j \leq m$. Also gilt auch $(q_j, w_j, Y_j \dots Y_m) \overset{*}{\vdash} (q_{j+1}, \varepsilon, Y_{j+1} \dots Y_m)$ für alle $1 \leq j \leq m$.
Daraus folgt

$$\begin{aligned}(q, w, X) &\vdash (q_1, w_1 \dots w_m, Y_1 \dots Y_m) \\ &\overset{*}{\vdash} (q_2, w_2 \dots w_m, Y_2 \dots Y_m) \\ &\overset{*}{\vdash} (q_3, w_3 \dots w_m, Y_3 \dots Y_m) \\ &\overset{*}{\vdash} \dots \overset{*}{\vdash} (q_m, w_m, Y_m) \overset{*}{\vdash} (q_{m+1}, \varepsilon, \varepsilon) = (p, \varepsilon, \varepsilon)\end{aligned}$$

und damit die Behauptung.

„⇐“

Wir zeigen per Induktion über die Länge k der Abarbeitung $(q, w, X) \overset{k}{\vdash} (p, \varepsilon, \varepsilon)$ von $\mathcal{A}$, dass eine Ableitung $[q, X, p] \overset{*}{\vdash} w$ in G existiert.

Induktionsanfang: Für $k = 1$ folgt aus $(q, w, X) \vdash (p, \varepsilon, \varepsilon)$, dass $w \in \Sigma \cup \{\varepsilon\}$ und $(p, \varepsilon) \in \delta(q, w, X)$. Dann ist $[q, X, p] \to w$ eine Regel von G.

Induktionsschritt: Betrachte eine Abarbeitung $(q, X, w) \overset{k}{\vdash} (p, \varepsilon, \varepsilon)$, wobei $w = aw'$ sei mit $a = \varepsilon$ und $w' = w$, falls der erste Schritt von $\mathcal{A}$ ein ε-Übergang ist, $a \in \Sigma$ sonst. Sei $(q_1, w', Y_1 \dots Y_m)$ die Konfiguration von $\mathcal{A}$ nach dem ersten Schritt. Dann gilt

$$(q, aw', X) \vdash (q_1, w', Y_1 \dots Y_m) \overset{k'}{\vdash} (p, \varepsilon, \varepsilon)$$

mit $k' \leq k-1$.
Sei $w' = w_1 \dots w_m$ Zerlegung von w mit $w_j \in \Sigma^*$ so gewählt, dass $\mathcal{A}$ startend mit der Konfiguration $(q_1, w', Y_1 \dots Y_m)$ bei der betrachteten Abarbeitung gerade nach dem Lesen von $w_1 \dots w_j$ zum ersten Mal den STACK-Inhalt $Y_{j+1} \dots Y_m$ erzeugt; q_{j+1} sei der zu diesem Zeitpunkt erreichte Zustand. Dann gilt: $q_{m+1} = p$ und

$$(q_j, w_j \dots w_m, Y_j \dots Y_m) \overset{k'}{\vdash} (q_{j+1}, w_{j+1} \dots w_m, Y_{j+1} \dots Y_m),$$

$k' \leq k-1$, und während der gesamten Abarbeitung liegt $Y_{j+1} \dots Y_m$ ungelesen auf dem STACK.
Also gilt auch

$$(q_j, w_j, Y_j) \overset{k'}{\vdash} (q_{j+1}, \varepsilon, \varepsilon).$$

Nach Induktionsvoraussetzung folgt daraus, dass $[q_j, Y_j, q_{j+1}] \overset{*}{\to} w_j$ in G existiert. Damit erhalten wir, dass auch

$$[q, X, p] \to a[q_1, Y_1, q_2][q_2, Y_2, q_3] \dots [q_m, Y_m, q_{m+1}] \overset{*}{\to} aw_1 \dots w_m = w$$

in G existiert.
Insgesamt folgt damit die Behauptung. □

5.29 Korollar
Die Klasse der von nichtdeterministischen Kellerautomaten akzeptierten Sprachen ist gleich der Klasse der kontextfreien Sprachen.

5.30 Satz
Sei $L \subseteq \Delta^*$ eine kontextfreie Sprache, $h : \Sigma \to \Delta^*$ ein Homomorphismus. Dann ist auch die Sprache $h^{-1}(L) = \{w \in \Sigma^* | h(w) \in L\}$ kontextfrei.

Beweis: Die Abbildung $h : \Sigma \to \Delta^*$ heißt Homomorphismus, da $h(w) = h(w_1) \cdots h(w_n)$, wobei $w = w_1 \dots w_n$. Wir benutzen Korollar 5.29 für den Beweis, indem wir einen PDA für $h^{-1}(L)$ angeben und dazu die Existenz eines PDA für L benutzen. Sei $\mathcal{A} = (Q, \Delta, \Gamma, \delta, q_0, Z_0, F)$ ein PDA für L. Der PDA $\mathcal{A}' = (Q', \Delta', \Gamma', \delta', q_0', Z_0', F')$ simuliert, was $\mathcal{A}$ auf dem Wort $h(a)$ tun kann. Dazu muss $\mathcal{A}'$ sich merken, wieviel $\mathcal{A}$ bei Abarbeitung von $h(a)$ nach jedem Übergang bereits gelesen hat. Setze daher

$$Q' := Q \times S_h$$

wobei S_h Menge der Suffixe von Wörtern $h(a)$, $a \in \Sigma$ (inkl. $h(a)$ und ε), ist.

$$q_0' := (q_0, \varepsilon)$$

$$F' := \{(q, \varepsilon) | q \in F\}$$

Außerdem sei $\Gamma' := \Gamma$, $Z_0' := Z_0$ und δ' definiert durch:

(i) $\delta'((q, \varepsilon), a, Y) := \{(q, h(a)), Y\}$ und
$\delta'((q, x), a, Y) := \emptyset$, falls $x \neq \varepsilon$
für $a \in \Sigma$.

(ii) $\delta'((q, x), \varepsilon, Y) \ni ((p, x), \gamma)$, falls $(p, \gamma) \in \delta(q, \varepsilon, Y)$ für alle $x \in S_h$.

(iii) $\delta'((q, ax), \varepsilon, Y) \ni ((p, x), \gamma)$, falls $(p, \gamma) \in \delta(q, a, Y)$ für alle $ax \in S_h$.

(i) bewirkt, dass $\mathcal{A}'$ beim Lesen von $a \in \Sigma$ abspeichert, dass $\mathcal{A}$ auf $h(a)$ simuliert werden muss. Die Simulation erfolgt durch ε-Übergänge. Durch (ii) werden die ε-Übergänge von $\mathcal{A}$ übernommen. Mittels (iii) wird die Abarbeitung von $\mathcal{A}$ simuliert.
Sei nun $h(w) = h(w_1) \cdots h(w_n) \in L$, dann kann $\mathcal{A}'$ auf w die Abarbeitung von $\mathcal{A}$ bei Eingabe $h(w)$ simulieren. Andererseits ist per Konstruktion von $\mathcal{A}'$ eine Abarbeitung einer Eingabe w eine Simulation von $\mathcal{A}$ bei Eingabe $h(w)$. □

5.6 Unentscheidbare Probleme für kontextfreie Grammatiken

Wir haben bereits ein paar Probleme für kontextfreie Sprachen bzw. Grammatiken betrachtet, welche „leicht“ entschieden werden können. Es kann in polynomialer Laufzeit entschieden werden, ob zu einer kontextfreien Grammatik G die Sprache $L(G)$ leer bzw. endlich ist. Das Wortproblem für kontextfreie Grammatiken ist ebenfalls in polynomialer Laufzeit entscheidbar. Außerdem sind für kontextfreie Grammatiken G, G_1 und G_2 die Sprachen $L(G)^*$, $L(G_1) \cup L(G_2)$ und

$L(G_1) \cdot L_(G_2)$ wieder kontextfrei. Andererseits sind $L(G_1) \cap L(G_2)$ und $(L(G))^c$ nicht notwendig kontextfrei. Wir werden sehen, dass es sogar unentscheidbar ist, ob der Durchschnitt zweier kontextfreier Sprachen bzw. das Komplement einer kontextfreien Sprache wieder kontextfrei ist. Ebenfalls unentscheidbar ist für kontextfreie Grammatiken G, G_1 und G_2 (über Σ), ob $L(G) = \Sigma^*$, $L(G_1) = L(G_2)$, $L(G_1) \subseteq L(G_2)$, $L(G_1) \cap L(G_2) = \emptyset$, $L(G)$ regulär, $L(G)$ inhärent mehrdeutig bzw. G mehrdeutig ist.

5.31 Satz
Das Problem für kontextfreie Grammatiken G_1 und G_2 zu entscheiden, ob $L(G_1) \cap L(G_2) = \emptyset$ ist, ist nicht entscheidbar.

Beweis: Wir beweisen, dass das Post'sche Korrespondenzproblem (PKP) auch entscheidbar wäre, wenn $L(G_1) \cap L(G_2) = \emptyset$ entscheidbar wäre. Das ist ein Widerspruch zur Unentscheidbarkeit des PKPs.
Das PKP besteht darin, für eine beliebige Menge $K = \{(x_1, y_1), \ldots, (x_k, y_k)\}$ mit $x_i, y_i \in \Sigma^*$ zu entscheiden, ob es eine Folge von Indizes $i_1, \ldots, i_n$ gibt, sodass $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$ ist. Wir geben für jede Instanz K des Post'schen Korrespondenzproblems kontextfreie Grammatiken G_1 und G_2 an, sodass es ein Wort $w \in L(G_1) \cap L(G_2)$ genau dann gibt, wenn es eine Lösung für K gibt. Das Alphabet zu G_1 und G_2 sei $\Sigma \cup \{a_1, \ldots, a_k\}$, $k = |K|$ und $V_1 = \{S_1\}$, $V_2 = \{S_2\}$. G_1 enthalte die Regeln

$$S_1 \to a_i x_i \text{ und } S_1 \to a_i S_1 x_i \text{ für alle } 1 \leq i \leq k;$$

analog enthalte G_2 die Regeln

$$S_2 \to a_i y_i \text{ und } S_2 \to a_i S_2 y_i \text{ für alle } 1 \leq i \leq k.$$

Dann gilt offensichtlich

$$L(G_1) = \{a_{i_n} \cdots a_{i_1} x_{i_1} \cdots x_{i_n} \mid n \in \mathbb{N},\ 1 \leq i_j \leq k\}$$

und

$$L(G_2) = \{a_{i_n} \cdots a_{i_1} y_{i_1} \cdots y_{i_n} \mid n \in \mathbb{N},\ 1 \leq i_j \leq k\}.$$

K besitzt genau dann eine Lösung, wenn es Indizes $i_1, \ldots, i_n$ gibt mit $x_{i_1} \cdots x_{i_n} = y_{i_1} \cdots y_{i_n}$. Dies ist aber genau dann der Fall, wenn es ein Wort $w = a_{i_n} \cdots a_{i_1} x_{i_1} \cdots x_{i_n} = a_{i_n} \cdots a_{i_1} y_{i_1} \cdots y_{i_n}$ gibt, also $w \in L(G_1) \cap L(G_2)$ und damit $L(G_1) \cap L(G_2) \neq \emptyset$. □

5.32 Satz
Das Problem, für eine kontextfreie Grammatik G zu entscheiden, ob sie eindeutig ist, ist nicht entscheidbar.

Beweis: G ist eindeutig, wenn es für jedes $w \in L(G)$ genau einen Syntaxbaum gibt. Wir zeigen, dass aus der Nichtentscheidbarkeit von $L(G_1) \cap L(G_2) = \emptyset$ für beliebige kontextfreie Grammatiken G_1 und G_2 die Behauptung folgt. Dazu benutzen wir die im Beweis zu Satz 5.31 konstruierten Grammatiken G_1 und G_2 und geben dazu eine kontextfreie Grammatik G an, die genau dann mehrdeutig ist, wenn $L(G_1) \cap L(G_2) \neq \emptyset$. Dazu habe G die Variablenmenge $V = \{S_1, S_2, S\}$, wobei S ein neues Startsymbol sei, und zusätzlich zu den Regeln von G_1 und

G_2 noch die Regeln $S \to S_1$ und $S \to S_2$.
Da G_1 und G_2 eindeutig sind, existiert $w \in L(G_1) \cap L(G_2)$ genau dann, wenn es in G Ableitungen $S \to S_1 \overset{*}{\to} w$ und $S \to S_2 \overset{*}{\to} w$ gibt, also G mehrdeutig ist. □

Als Hilfsmittel zum Beweis der Unentscheidbarkeit weiterer Probleme für kontextfreie Grammatiken benutzen wir die Sprache aller korrekten Rechenwege einer Turingmaschine.
Sei $\mathcal{M}$ eine TM, bestehend aus $(Q, \Sigma, \Gamma, \sqcup, q_0, \delta, F)$. Dann kann eine Berechnung von $\mathcal{M}$ durch die Folge der durchlaufenen **Konfigurationen** $\alpha q \beta$ mit $\alpha, \beta \in \Gamma^*$ und $q \in Q$ beschrieben werden. $\alpha q \beta$ bedeutet, dass auf dem Band das Wort $\alpha\beta$, umgeben von Blanksymbolen, steht, die Turingmaschine im Zustand q ist und der Lese-/Schreibkopf auf die Stelle des Bandes, an der das erste Symbol von β steht, zeigt.
Wenn $w_1, w_2, \ldots, w_n$ die Abfolge der Konfigurationen einer Berechnung von $\mathcal{M}$ ist, so kann dieser Rechenweg durch das Wort $w_1 \# w_2 \# \ldots \# w_n \#$, mit $\# \notin \Gamma$ Trennsymbol, kodiert werden. Allerdings lässt sich die Sprache aller Wörter, die in dieser Weise die korrekten Rechenwege einer TM kodieren, nicht unbedingt durch kontextfreie Grammatiken beschreiben. Daher wird ein „Trick“ angewendet und jede zweite Konfiguration gespiegelt kodiert.

5.33 Definition
Die **Sprache $\mathbf{B}_{\mathcal{M}}$ der korrekten Rechenwege einer TM $\mathcal{M}$** *besteht aus allen Worten*

$$w_1 \# w_2^R \# w_3 \# w_4^R \ldots w_n^R \#, \text{ falls } n \text{ gerade und}$$

$$w_1 \# w_2^R \# w_3 \# w_4^R \ldots w_n \#, \text{ falls } n \text{ ungerade,}$$

wobei die w_i, $1 \leq i \leq n$, Konfigurationen von $\mathcal{M}$ sind, w_1 eine Anfangskonfiguration, w_n eine akzeptierende Konfiguration und für alle $1 \leq i \leq n-1$ die Konfiguration w_{i+1} die direkte Nachfolgekonfiguration von w_i bei einer korrekten Berechnung von $\mathcal{M}$ ist.

5.34 Lemma
Für alle Turingmaschinen $\mathcal{M}$ ist $B_{\mathcal{M}}$ der Durchschnitt zweier Sprachen $L_1 = L(G_1)$ und $L_2 = L(G_2)$, wobei G_1 und G_2 kontextfreie Grammatiken sind.

Beweis: Wir konstruieren L_1 und L_2 aus den Sprachen

$$L := \{u \# v^R | v \text{ ist direkte Nachfolgekonfiguration von } u \text{ für } \mathcal{M}\}$$

bzw.

$$L' := \{v^R \# u | u \text{ ist direkte Nachfolgekonfiguration von } v \text{ für } \mathcal{M}\}$$

Falls L und L' kontextfrei sind, so sind auch

$$L_1 := (L\{\#\})^*(\{\varepsilon\} \cup \Gamma^* F \Gamma^* \{\#\})$$

und

$$L_2 := \{q_0\}\Sigma^*\{\#\}(L'\{\#\})^*(\{\varepsilon\} \cup \Gamma^* F \Gamma^* \{\#\})$$

kontextfrei, wobei Γ Bandalphabet, Σ Eingabealphabet, q_0 Anfangszustand und F Endzustandsmenge von $\mathcal{M}$. Offensichtlich haben alle Wörter aus L_1 die Form

$$w_1 \# w_2^R \# \dots w_{2i-1} \# w_{2i}^R \# \text{ oder}$$

$$w_1 \# w_2^R \# \dots w_{2i-1} \# w_{2i}^R \# w_{2i+1} \#$$

mit w_j Konfiguration von $\mathcal{M}$ und w_{2j} direkte Nachfolgekonfiguration von w_{2j-1} für alle $1 \leq j \leq i$ und w_{2i+1} akzeptierende Konfiguration, falls vorhanden.
Analog haben alle Wörter aus L_2 die Form

$$w_1 \# w_2^R \# \dots w_{2i-1} \# w_{2i}^R \# \text{ oder}$$

$$w_1 \# w_2^R \# \dots w_{2i-2}^R \# w_{2i-1} \#$$

mit w_j Konfiguration von $\mathcal{M}$, w_1 Anfangskonfiguration, w_{2j+1} direkte Nachfolgekonfiguration von w_{2j} für alle $1 \leq j \leq i-1$ und w_{2i} akzeptierende Konfiguration, falls vorhanden.
Dann ist $B_{\mathcal{M}} = L_1 \cap L_2$.

Wir geben nun eine kontextfreie Grammatik G für L an mit Startvariable S und zusätzlicher Variable A. G enthalte folgende Regeln:

(i) alle Regeln $S \to aSa$, $a \in \Gamma \backslash \{\sqcup\}$;

(ii) für alle Übergänge $\delta(q, a) = (q', b, R)$ von $\mathcal{M}$ die Regeln $S \to qaAq'b$;

(iii) für alle Übergänge $\delta(q, a) = (q', b, L)$ von $\mathcal{M}$ die Regeln $S \to xqaAbxq'$, wobei x Symbol links von a beim Lesen von a im Zustand q;

(iv) für alle Übergänge $\delta(q, a) = (q', b, N)$ von $\mathcal{M}$ die Regeln $S \to qaAbq'$;

(v) für alle $a \in \Gamma$ die Regeln $A \to aAa$;

(vi) die Regel $A \to \#$.

Analog kann eine kontextfreie Grammatik G' für L' angegeben werden. Es ist leicht zu zeigen, dass $L(G) = L$ und $L(G') = L'$ ist. Damit ist die Behauptung bewiesen. □

Bemerkung:
Falls $\mathcal{M}$ in jeder Berechnung nur höchstens einen Rechenschritt ausführt, ist $B_{\mathcal{M}}$ sogar selbst kontextfrei.

5.35 Lemma
Sei $\mathcal{M}$ eine TM, die auf jeder Eingabe mindestens zwei Rechenschritte ausführt. Dann ist die Sprache $B_{\mathcal{M}}$ genau dann kontextfrei, wenn $L(\mathcal{M})$ endlich ist.

Beweis:

„⇐“ Falls $L(\mathcal{M})$ endlich ist, ist $B_{\mathcal{M}}$ auch endlich, da es zu jeder Eingabe aus $L(\mathcal{M})$ genau eine akzeptierende Berechnung gibt. Jede endliche Sprache ist regulär, also auch kontextfrei.

„⇒“ Angenommen $L(\mathcal{M})$ ist unendlich und $B_{\mathcal{M}}$ wäre kontextfrei. Wir führen diese Annahme unter Benutzung von Ogden's Lemma zum Widerspruch. Da

$L(\mathcal{M})$ unendlich ist, gibt es zu der Konstanten n aus Ogden's Lemma ein $w \in B_{\mathcal{M}}$ mit $w = w_1 \# w_2^R \# \ldots$ und $|w_2^R| \geq n$. Wenn alle Symbole aus $\# w_2^R \#$ markiert werden, muss es eine Zerlegung $uvwxy$ von w geben, sodass vx mindestens einen und vwx höchstens n markierte Buchstaben enthält und $uv^i wx^i y \in B_{\mathcal{M}}$ für alle $i \geq 0$. Da $\mathcal{M}$ mindestens zwei Berechnungsschritte ausführt, existieren die Konfigurationen w_1, w_2 und w_3. Entsprechend der Zerlegung von w enthalten $\# w_2^R \#$ und vx mindestens einen gemeinsamen Buchstaben, und nur eines der Worte w_1 und w_3 hat ebenfalls gemeinsame Buchstaben mit vx. Wenn w_1 keinen gemeinsamen Buchstaben mit vx hat, ist $uv^2wx^2y \notin B_{\mathcal{M}}$, da die Berechnung für die Anfangskonfiguration w_1 eindeutig ist. Aus demselben Grund ist $uv^2wx^2y \notin B_{\mathcal{M}}$, falls $w_1\#$ Präfix von uv^2 ist. Falls v ein Teilwort von w_1 wäre, müsste x ein Teilwort von w_2^R sein, damit für großes i das Wort $uv^i wx^i y \in B_{\mathcal{M}}$ ist, da zwei aufeinanderfolgende Konfigurationen etwa gleich lang sind. Dann wäre aber w_3 als Nachfolgekonfiguration zu kurz, $uv^i wx^i y$ also keine Kodierung eines korrekten Rechenweges von $\mathcal{M}$. □

5.36 Lemma
Für jede TM $\mathcal{M}$ ist das Komplement $(B_{\mathcal{M}})^c$ von $B_{\mathcal{M}}$ kontextfrei.

Beweis: (Skizze)
Wir geben drei kontextfreie Sprachen L_1, L_2 und L_3 an, sodass $(B_{\mathcal{M}})^c = L_1 \cup L_2 \cup L_3$. Wir nennen

$$w := \begin{cases} w_1 \# w_2^R \# \ldots w_n^R \# & \text{für gerades } n \\ w_1 \# w_2^R \# \ldots w_n \# & \text{für ungerades } n \end{cases}$$

wohlgeformt, wobei w_i Konfiguration für alle i, $1 \leq i \leq n$, w_1 Anfangskonfiguration und w_n akzeptierende Konfiguration.

$$L_1 := \{w | w \text{ nicht wohlgeformt}\}$$

$$L_2 := \left\{ w \;\middle|\; \begin{array}{l} w \text{ wohlgeformt und es gibt } i \text{ ungerade, sodass } w_i \# w_{i+1}^R \\ \text{Teilwort von } w, \text{ für das } w_{i+1} \text{ nicht Nachfolgekonfiguration} \\ \text{von } w_i \text{ ist.} \end{array} \right\}$$

$$L_3 := \left\{ w \;\middle|\; \begin{array}{l} w \text{ wohlgeformt und es gibt } i \text{ gerade, sodass } w_i^R \# w_{i+1} \\ \text{Teilwort von } w, \text{ für das } w_{i+1} \text{ nicht Nachfolgekonfiguration} \\ \text{von } w_i \text{ ist.} \end{array} \right\}$$

Dann ist offensichtlich $(B_{\mathcal{M}})^c = L_1 \cup L_2 \cup L_3$. Es bleibt zu zeigen, dass L_1, L_2 und L_3 kontextfrei sind. L_1 ist das Komplement des regulären Ausdrucks

$$q_0 \Sigma^* \# (\Gamma^* Q \Gamma^* \#)^* \Gamma^* F \Gamma^* \#$$

und ist damit sogar regulär. Für L_1 und L_2 können kontextfreie Grammatiken angegeben werden, wobei die Regeln ähnlich wie im Beweis zu Lemma 5.34 hergeleitet werden. □

5.37 Satz
Seien G, G_1, G_2 kontextfreie Grammatiken über Σ. Es ist nicht entscheidbar, ob

(i) $(L(G))^c$ kontextfrei ist,

(ii) $L(G_1) \cap L(G_2)$ *kontextfrei ist,*

(iii) $L(G) = \Sigma^*$,

(iv) $L(G_1) = L(G_2)$,

(v) $L(G_1) \subseteq L(G_2)$.

Beweis:

(i) Angenommen, es gäbe eine TM $\mathcal{M}'$, die für G entscheidet, ob $(L(G))^c$ kontextfrei ist. Wir zeigen, dass es dann auch eine TM $\mathcal{M}''$ gäbe, die

$$L := \{\langle \mathcal{M} \rangle \,|L(\mathcal{M}) \text{ endlich}\}$$

entscheidet. Dies wäre ein Widerspruch zum Satz von Rice (3.16).
O.B.d.A. enthalte L nur $\langle \mathcal{M} \rangle$, für die $\mathcal{M}$ mindestens zwei Rechenschritte ausführt. Die TM $\mathcal{M}''$ berechnet für jede Eingabe $\langle \mathcal{M} \rangle$ mit Lemma 5.36 eine kontextfreie Grammatik $G_{\mathcal{M}}$ für $(B_{\mathcal{M}})^c$. Dann simuliert $\mathcal{M}''$ die TM $\mathcal{M}'$ auf der Eingabe $G_{\mathcal{M}}$ und entscheidet damit, ob $B_{\mathcal{M}}$ kontextfrei ist. Nach Lemma 5.35 ist dies äquivalent dazu, dass $L(\mathcal{M})$ endlich ist.

(ii) Angenommen, es gäbe eine TM $\mathcal{M}'$, die für G_1 und G_2 entscheidet, ob $L(G_1) \cap L(G_2)$ kontextfrei ist. Wir betrachten eine TM $\mathcal{M}''$, die auf Eingaben $\langle \mathcal{M} \rangle$ entsprechend Lemma 5.34 kontextfreie Grammatiken $G_{1,\mathcal{M}}$ und $G_{2,\mathcal{M}}$ berechnet mit $L(G_{1,\mathcal{M}}) \cap L(G_{2,\mathcal{M}}) = B_{\mathcal{M}}$. Dann simuliert $\mathcal{M}''$ die TM $\mathcal{M}'$ auf den Eingaben $G_{1,\mathcal{M}}$ und $G_{2,\mathcal{M}}$ und entscheidet damit, ob $B_{\mathcal{M}}$ kontextfrei ist, also wieder ob $L(\mathcal{M})$ endlich ist. Dies ist wie in (i) ein Widerspruch zum Satz von Rice.

(iii) Angenommen, es gäbe eine TM $\mathcal{M}'$, die für G und Σ entscheidet, ob $L(G) = \Sigma^*$ ist. Wir betrachten eine TM $\mathcal{M}''$, die für $\langle \mathcal{M} \rangle$ entscheidet, ob $L(\mathcal{M}) = \emptyset$ ist. Dies wäre wieder ein Widerspruch zum Satz von Rice. $\mathcal{M}''$ berechnet für $\langle \mathcal{M} \rangle$ mit Lemma 5.36 eine kontextfreie Grammatik $G_{\mathcal{M}}$ für $(B_{\mathcal{M}})^c$. Dann simuliert $\mathcal{M}''$ die TM $\mathcal{M}'$ auf $G_{\mathcal{M}}$ und entscheidet, ob $(B_{\mathcal{M}})^c = (\Gamma \cup \{\#\})^*$ ist. Dies ist äquivalent dazu, dass $B_{\mathcal{M}} = \emptyset$ und damit $L(\mathcal{M}) = \emptyset$ ist.

(iv) Die Entscheidung, ob $L(G_1) = \Sigma^*$ ist, ist ein Spezialfall von $L(G_1) = L(G_2)$. Damit ist $L(G_1) = L(G_2)$ ebenfalls unentscheidbar.

(v) Die Entscheidung, ob $\Sigma^* = L(G_2)$ ist, ist ein Spezialfall von $L(G_1) \subseteq L(G_2)$. Damit ist auch $L(G_1) \subseteq L(G_2)$ unentscheidbar.

□

5.38 Satz
Die Sprache $L = \{a^i b^j c^k | i = j \text{ oder } j = k\}$ *ist inhärent mehrdeutig.*

Beweis: (Skizze)
Zunächst stellen wir fest, dass $L = \{a^i b^j c^k | i = j \text{ oder } j = k\}$ kontextfrei ist. Es gilt $L = L_1 \cup L_2$, wobei

$$L_1 := \{a^i b^i | i \geq 0\} \cdot \{c\}^*$$

$$L_2 := \{a\}^* \cdot \{b^i c^i | i \geq 0\}.$$

Der Schnitt $L_1 \cap L_2 = \{a^i b^i c^i | i \geq 0\}$ ist jedoch nicht kontextfrei. Der Beweis der Mehrdeutigkeit von L beruht nun darauf, für gewisse Wörter aus $L_1 \cap L_2$ zu zeigen, dass jede beliebige kontextfreie Grammatik G zu L zwei Syntaxbäume für diese Wörter besitzt. Ausgehend von den Wörtern $a^n b^n c^{n+n!}$ und $a^{n+n!} b^n c^n$ wird unter Benutzung von Ogden's Lemma gezeigt, dass es zu $a^{n+n!} b^{n+n!} c^{n+n!}$ zwei Syntaxbäume gibt. □