Berechenbarkeit und Komplexität

Skript zur Vorlesung

Johannes Blömer

mit Ergänzungen von Christian Scheideler

Kontakt

Christian Scheideler
Universität Paderborn
Institut für Informatik
Fürstenallee 11
33102 Paderborn

email: scheideler@uni-paderborn.de

Stand: 24. Januar 2019

Aktualisierungen:

- 30.10.: Ergänzungen zum Beweis von Satz 2.8.
- 7.11.: Korrektur des Beweises von Satz 2.25.
- 13.11.: Beweis des Satzes von Rice hinzugefügt.

Inhaltsverzeichnis

1 Einleitung 1
- 1.1 Ziele der Vorlesung . . . 1
- 1.2 Notationen . . . 2

2 Berechenbarkeit 4
- 2.1 Turingmaschinen . . . 4
- 2.2 Programmiertechniken . . . 13
  - 2.2.1 Endlicher Speicher - „Im Zustand merken“ . . . 13
  - 2.2.2 Markieren von Buchstaben und Bandpositionen . . . 13
- 2.3 Mehrband Turingmaschinen und Simulationen . . . 15
- 2.4 Die Churchsche These . . . 18
- 2.5 Eigenschaften entscheidbarer und rekursiv aufzählbarer Sprachen . . . 19
- 2.6 Universelle Turingmaschinen . . . 21
  - 2.6.1 Beschreibung einer Turingmaschine - Gödelnummern . . . 21
  - 2.6.2 Eigenschaften von Turingmaschinen als entscheidbare bzw. rekursiv aufzählbare Sprachen . . . 23
- 2.7 Die Existenz nicht rekursiv aufzählbarer Sprachen . . . 27
  - 2.7.1 Eigenschaften unendlicher Mengen . . . 27
  - 2.7.2 Paradoxien in der Mengentheorie . . . 31
  - 2.7.3 Konsequenzen für die Berechenbarkeit . . . 32
  - 2.7.4 Orakelrechnungen . . . 33
- 2.8 Eine nicht rekursiv aufzählbare Sprache . . . 34
- 2.9 Reduktionen . . . 37
- 2.10 Nichtentscheidbarkeit und Reduktionen . . . 43
- 2.11 Weitere unentscheidbare Sprachen . . . 46

3 Zeitkomplexität und die Klasse P 49
- 3.1 Motivation, Überblick und einführende Beispiele . . . 49
- 3.2 $\mathcal{O}$-Notation . . . 52
- 3.3 Zeitkomplexität einer Turingmaschine . . . 52
- 3.4 Die Klasse P . . . 56
- 3.5 Die Klasse NP . . . 61
  - 3.5.1 Verifizierer und die Klasse NP . . . 61
  - 3.5.2 Nichtdeterministische Turingmaschinen und die Klasse NP . . . 65

3.6 NP-Vollständigkeit 75
3.6.1 Polynomielle Reduktionen 75
3.6.2 Definition von NP-Vollständigkeit 78
3.6.3 Der Satz von Cook-Levin - SAT ist NP-vollständig 80
3.7 Beispiele NP-vollständiger Probleme 86
3.7.1 $\mathrm{RS}_{\mathrm{ent}}$ ist NP-vollständig 87

4 NP-vollständig - Was nun? 91
4.1 Spezialfälle 91
4.2 Approximationsalgorithmen - Notation 95
4.3 Das Max-Cut Problem 95
4.4 Das Problem des Handlungsreisenden 97
4.5 Das Rucksackproblem 99
4.6 Ein Unmöglichkeitsergebnis 103
4.7 Approximationsschemata 104

Kapitel 1

Einleitung

In dieser Vorlesung werden Sie ein faszinierendes Gebiet der Informatik kennen lernen, die *Theoretische Informatik*. Einige werden jetzt denken, dass die Begriffe Theorie und Faszination nicht zusammen passen. Ziel dieser Vorlesung ist es, Sie durch eine Einführung der wesentlichen Begriffe, Ergebnisse und Techniken der Theoretischen Informatik vom Gegenteil zu überzeugen. Im weiteren Verlauf Ihres Studiums werden Sie dann feststellen, dass Sie etliche Ergebnisse und Techniken dieser Vorlesung in der Praxis verwenden können.

1.1 Ziele der Vorlesung

Die grundlegende Frage, mit der wir uns in dieser Vorlesung beschäftigen, lautet:

Was sind die wesentlichen Möglichkeiten und Grenzen von Computern?

Es mag für einige überraschend klingen, aber es gibt Probleme, die auf Computern grundsätzlich nicht gelöst werden können. Vielleicht haben Sie sich schon gefragt, warum Sie noch kein Java-Programm kennen gelernt haben, dass für jedes Java-Programm entscheidet, ob es bei einigen Eingaben in eine Endlosschleife gerät. Dieses Problem wird das *Halteproblem* genannt. Ein Programm, das das Halteproblem löst, wäre doch außerordentlich hilfreich und könnte häufig ein quälend langsamen Debuggen von Programmen ersetzen. Die Antwort, warum Sie solch ein Programm noch nicht gesehen haben, ist einfach: Es kann ein solches Programm nicht geben! Und wir werden diese Aussage in dieser Vorlesung beweisen.

In der Vorlesung *Datenstrukturen und Algorithmen* haben Sie Algorithmen für unterschiedliche Probleme gesehen. Für das Sortierproblem etwa haben Sie mehrere Algorithmen kennen gelernt, die alle eine Laufzeit (gemessen in Anzahl der Vergleiche) von $\mathcal{O}(n \log(n))$ besitzen. Das Sortierproblem lässt sich daher effizient lösen. Für das Rucksackproblem haben Sie einen auf dem Prinzip der *dynamischen Programmierung* basierenden Algorithmus kennen gelernt mit Laufzeit $\mathcal{O}(nW)$, wobei n die Anzahl der Gegenstände und W die Kapazität des Rucksacks ist. Wie wir in dieser Vorlesung sehen werden, ist dieses kein effizienter Algorithmus (können Sie sich schon jetzt erklären, warum dies kein effizienter Algorithmus ist?). Gibt es einen besseren, d.h. effizienteren Algorithmus für das Rucksackproblem? Bislang ist es noch niemandem gelungen, einen wesentlich besseren Algorithmus zu entwerfen. Warum ist dies so? Wie wir sehen werden, gibt es gute Gründe zu glauben, dass das Rucksackproblem nicht effizient gelöst werden kann. Uns interessiert in der Theoretischen Informatik nicht nur die Unterscheidung zwischen Problemen, die wie das Sortierproblem und das Rucksackproblem auf Computern algorithmisch gelöst werden können, und Problemen, die wie das Halteproblem grundsätzlich nicht algorithmisch gelöst werden können. Wir wollen auch verstehen, welche Probleme wie das

Sortierproblem effizient algorithmisch gelöst werden können, und für welche Probleme dieses nicht möglich ist.

Um diese Fragen untersuchen zu können und um Aussagen mathematisch präzise beweisen zu können, benötigen wir zunächst einmal ein formales Modell eines Computers. Dieses sollte komplex genug sein, um die Möglichkeiten moderner Computer angemessen wiederzugeben. Das Modell sollte aber auch so einfach sein, dass es uns gelingt, Aussagen über die Mächtigkeit von Computern zu beweisen. Ein solches Modell wurde bereits vor über 70 Jahren von dem englischen Mathematiker Alan Turing eingeführt, die *Turingmaschinen*. Weiter werden wir erklären und definieren müssen, wann ein Problem *effizient* gelöst werden kann. Dieses wird uns auf die Theorie der polynomiellen Algorithmen oder Turingmaschinen und schließlich auf die Theorie der *NP-Vollständigkeit* führen. Letztere wird uns helfen zu erklären, warum für viele interessante und praktisch wichtige Probleme noch keine effizienten Algorithmen bekannt sind, wir aber auch nicht zeigen können, dass diese Probleme algorithmisch nicht effizient gelöst werden können.

Die Vorlesung gliedert sich in zwei Teile. Beginnen werden wir mit der Beschreibung von Turingmaschinen und der Untersuchung von Problemen, die sich nicht auf einem Computer lösen lassen. Dieser Teil der Vorlesung ist die Theorie der *Berechenbarkeit*. Danach werden wir uns mit der Frage beschäftigen, was unter einem algorithmisch effizient lösbaren Problem verstanden werden kann. Auch hier werden wir Turingmaschinen als Rechnermodell nutzen und eine Klassifikation von Problem entsprechend ihrer effizienten Lösbarkeit oder algorithmischen *Komplexität* entwerfen.

Die Theoretische Informatik liefert viele überraschende und elegante Konzepte und Ergebnisse, die in etlichen anderen Bereichen der Informatik angewandt werden. Hauptziel der Vorlesung ist es, Ihnen diese Ergebnisse und Konzepte zu vermitteln. Den Schwerpunkt legen wir dabei auf den wesentlichen Ideen. Diese Ideen kann man jedoch nur dann richtig einordnen, wenn wir die technischen Details und Schwierigkeiten angemessen verstehen. Wir werden Ihnen daher die technischen Details nicht ersparen und nicht ersparen können, wenn dies für das tiefere Verständnis notwendig ist. Das Erlernen dieser Details und der wichtigen sich dahinter verbergenden Techniken ist wie das Erlernen einer neuen Sprache. Es mag am Anfang mühsam sein, aber einmal erlernt, wird vieles einfacher und klarer.

1.2 Notationen

Bevor wir mit dem eigentlichen Inhalt der Vorlesung beginnen, noch einige wenige wesentliche Bezeichnungen und Sprechweisen. Für eine endliche, nichtleere Menge $A=\{a_1, \dots a_l\}$ (*Alphabet*, die a_i's sind *Buchstaben* oder *Symbole*) bezeichnet:

- A^n die Menge aller *Worte* (Sätze) mit n Buchstaben aus A, also $A^n = \{b_1 b_2 b_3 \dots b_n \mid b_i \in A \text{ für } i = 1, \dots, n\}$, $n \geq 1$,
- $A^0 := \{\varepsilon\}$, wobei ε das *leere Wort* ist, also das Wort, das aus keinem Buchstaben besteht. (Beachten Sie, dass A^0 nicht die leere Menge ist!)

$$\begin{aligned} A^* &:= \bigcup_{n=0}^{\infty} A^n \\ A^+ &:= \bigcup_{n=1}^{\infty} A^n \\ A^{\leq m} &:= \bigcup_{n=0}^{m} A^n \end{aligned}$$

- $L \subseteq A^*$ ist eine *Sprache über* $\boldsymbol{A}$, $\bar{L} := A^* \setminus L$ ihr *Komplement*.

Weitere Notationen:

- Für $\alpha,\beta \in A^*$ ist $\alpha \cdot \beta$ bzw. kurz $\alpha\beta$ die *Konkatenation* von α und β, d. h. das Wort, das sich durch das Hintereinanderschreiben von α und β ergibt (Haus·maus = Hausmaus). Häufig werden wir den $\cdot$ weglassen und statt $\alpha \cdot \beta$ einfach $\alpha\beta$ für die Konkatenation von α und β schreiben.

  Es gilt: $\varepsilon \cdot \beta = \beta \cdot \varepsilon = \beta$, $\varepsilon \cdot \varepsilon = \varepsilon$, und für beliebiges $a \in A$ und $n \geq 1$ gilt:
  $a^0 := \varepsilon$, und $a^n := a^{n-1}a = (\underbrace{aaa \ldots a}_{n \text{ Mal}})$.

- $|\alpha|$ ist die Länge von ($\hat{=}$ # Buchstaben in) α.

- $bin(n)$ die Binärstellung einer natürlichen Zahl n.

- Für $b_0, \ldots, b_{r-1} \in \{0,1\}$ bezeichnet $(b_{r-1} \ldots b_0)_2$ die durch $b_0 \ldots b_{r-1}$ binär dargestellte natürliche Zahl, d.h.,

$$(b_{r-1} \ldots b_0)_2 = \sum_{i=0}^{r-1} b_i 2^i.$$

- Für eine Menge M ist $\mathcal{P}(M)$ die *Potenzmenge*, d.h. die Menge aller Teilmengen, von M.

Kapitel 2

Berechenbarkeit

In diesem Kapitel wollen wir untersuchen, welche Probleme auf Computern gelöst werden können. Dabei werden wir uns auf *Entscheidungsprobleme* konzentrieren, die wir durch *Sprachen* beschreiben können. Genauer gesagt wollen wir folgendes Problem lösen: Angenommen, wir haben eine beliebige Sprache $L \subseteq \Sigma^*$ über einem Alphabet Σ. Das Ziel ist es dann herauszufinden, ob es einen Algorithmus gibt, der für eine Eingabe $x \in \Sigma^*$ in endlicher Zeit entscheiden kann, ob $x \in L$ ist oder nicht. Ein Entscheidungsproblem oder die dazugehörige Sprache werden wir *entscheidbar* nennen, wenn das Entscheidungsproblem wie oben beschrieben gelöst werden kann. Wir werden sehen, dass nicht alle Sprachen entscheidbar sind.

Um diese Begriffe präzise zu fassen, benötigen wir ein mathematisch präzises Rechnermodell. Hierzu werden wir die *Turingmaschinen* einführen. Nach der Einführung einiger Grundbegriffe und der Vorstellung von Beispielen werden wir einige grundlegende Programmiertechniken kennenlernen, die uns in die Lage versetzen zu zeigen, dass verschiedene Turingmaschinenmodelle gleich mächtig sind bezüglich der Entscheidbarkeit von Sprachen. Das Gleiche konnte auch für verschiedene andere Rechenmodelle nachgewiesen werden wie Registermaschinen, was uns zur Churchschen These führt, die aussagt, dass der über Turingmaschinen definierte Entscheidbarkeitsbegriff mit unserer durch moderne Computer geprägten Vorstellung übereinstimmt. Anschließend werden wir die Existenz *universeller* Turingmaschinen nachweisen, d.h. Turingmaschinen, die jede andere Turingmaschine simulieren können. Mit anderen Worten, eine Turingmaschine ist in der Lage, auf Code und Daten wie eine Registermaschine zu arbeiten, welche das Grundmodell jedes modernen Computers darstellt: Ein Prozess agiert auf einem linear adressierten Speicher, in dem das Programm und die Daten gehalten werden. Zum Abschluss des Kapitels werden wir dann die Existenz unentscheidbarer Sprachen herleiten und wichtige Beispiele unentscheidbarer Sprachen kennenlernen. Dieser Teil bildet den Kern von Kapitel 2.

2.1 Turingmaschinen

Die Turingmaschine wurde 1936 (vor der Entwicklung erster elektronischer Rechenanlagen) von Alan Turing eingeführt. Sie diente ihm als formales Modell, um den Begriff „berechenbar“ zu formalisieren. Als Speichermedium besitzt eine Turingmaschine ein nach rechts unendliches Band, das in Zellen aufgeteilt ist. Eine Zelle kann ein Element aus dem sogenannten Bandalphabet speichern. Die Turingmaschine besitzt eine Kontrolle und einen Lesekopf. Die Kontrolle befindet sich stets in einem von endlich vielen Zuständen. Der Lesekopf kann den Inhalt einer Speicherzelle lesen, aber auch ein neues Zeichen in eine Zelle schreiben (schematisch dargestellt in Abbildung 2.1). Weitere Bausteine des Modells sind in der folgenden Definition

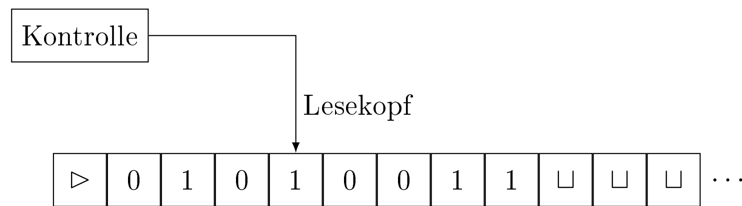


Abbildung 2.1: Schematische Darstellung einer Turingmaschine

zusammengefasst.

Definition 2.1 Eine deterministische 1-Band-*Turingmaschine* oder einfach Turingmaschine (DTM) wird durch ein 4-Tupel $M = (Q, \Sigma, \Gamma, \delta)$ beschrieben. Dabei sind Q, Σ, Γ endliche, nichtleere Mengen und

1. Q ist die *Zustandsmenge* mit $q_0, q_{\text{accept}}, q_{\text{reject}} \in Q$ und $q_{\text{accept}} \neq q_{\text{reject}}$. q_0 ist der *Startzustand*, q_{accept} ist der *akzeptierende Zustand* und q_{reject} ist der *ablehnende Zustand.*
2. Σ ist das *Eingabealphabet*, wobei das *Blank* $\sqcup$ und das *Startsymbol* $\rhd$ nicht in Σ liegen.
3. Γ ist das *Bandalphabet*, wobei $\Sigma \subset \Gamma$ und $\sqcup, \rhd \in \Gamma$.
4. $\delta : Q \backslash \{q_{\text{accept}}, q_{\text{reject}}\} \times \Gamma \to Q \times \Gamma \times \{R, L\}$ ist die *Übergangsfunktion.* Für $q_{\text{accept}} \times \Gamma$ und $q_{\text{reject}} \times \Gamma$ ist δ also *nicht* definiert.
5. Für das Symbol $\rhd \in \Gamma$ und alle Zustände $q \in Q$ gilt $\delta(q, \rhd) = (p, \rhd, R)$ mit $p \in Q$. Außerdem muss für alle $q \in Q$ und $a \in \Gamma, a \neq \rhd$, gelten $\delta(q, a) = (p, b, D)$, wobei $p \in Q, b \in \Gamma, b \neq \rhd$ und $D \in \{L, R\}$. Mit anderen Worten: $\rhd$ kann nur dann als zweite Koordinate eines Funktionswertes von δ auftauchen, wenn es auch zweite Koordinate des Argumentes ist.

Später werden wir auch *nichtdeterministische* Turingmaschinen kennenlernen. Um späteren Verwechslungen vorzubeugen, werden wir daher auch jetzt schon stets von deterministischen Turingmaschinen (DTMs) reden.
Beispiel 1: Die DTM M ist wie folgt definiert.

- $\Sigma = \{a\}$ ist das Eingabealphabet.
- $\Gamma = \{\sqcup, \rhd, a\}$ ist das Bandalphabet.
- $Q = \{q_0, q_1, q_2, q_3\}$ ist die Zustandsmenge. Dabei ist $q_2 = q_{\text{accept}}$ der akzeptierende Zustand und $q_3 = q_{\text{reject}}$ ist der ablehnende Zustand.
- Die Übergangsfunktion δ ist durch die folgende Tabelle 2.2 definiert.

δ	a	$\rhd$	$\sqcup$
q_0	(q_1, a, R)	$(q_0, \rhd, R)$	$(q_3, \sqcup, R)$
q_1	(q_1, a, R)	$(q_1, \rhd, R)$	$(q_2, \sqcup, R)$

Abbildung 2.2: Erstes Beispiel der Übergangsfunktion einer DTM

Ändern wir die Übergangsfunktion δ, indem wir in der ersten Zeile der Tabelle $\delta(q_0, \triangleright) = (q_0, \triangleright, R)$ durch $\delta(q_0, \triangleright) = (q_0, \sqcup, L)$ ersetzen, so erhalten wir keine korrekte Beschreibung einer DTM. Denn ist die zweite Koordinate des Arguments von δ das Startsymbol $\triangleright$, so muss auch die zweite Koordinate des Funktionswertes $\triangleright$ sein. Außerdem muss in diesem Fall die dritte Koordinate des Funktionswertes R sein (5. in der Definition einer DTM).

Wir beschreiben nun die Berechnung einer DTM. Zunächst geschieht dieses informell, dann werden wir die Berechnung genauer definieren. In der äußersten linken Zelle des Bandes einer DTM steht immer das Startsymbol $\triangleright$. In den Zellen rechts vom Startsymbol $\triangleright$ steht zu Beginn der Berechnung einer DTM die *Eingabe* $w = (w_1, w_2, \ldots, w_n) \in \Sigma^*$. Die restlichen Zellen des Bandes sind mit dem Blank $\sqcup$ gefüllt. Da $\sqcup$ nicht zum Eingabealphabet Σ gehört, zeigt also das erste $\sqcup$ das Ende der Eingabe an.

Die Berechnungsschritte oder einfach Schritte der DTM erfolgen nun gemäß der Übergangsfunktion δ. Zu jedem Zeitpunkt befindet sich die Maschine in einem Zustand $q \in Q$ und der Lesekopf liest den Inhalt einer Zelle des Bandes. Ist der Inhalt der Zelle $a \in \Gamma$, so führt die Maschine die durch $\delta(q, a)$ beschriebenen Aktionen aus. Ist z.B. $\delta(q, a) = (q', b, L)$, so wird in die Zelle, die gerade vom Lesekopf gelesen wird, der Inhalt a überschrieben und durch b ersetzt, der Lesekopf bewegt sich um eine Zelle nach links und der neue Zustand der Maschine ist q'. Ist hingegen $\delta(q, a) = (q', b, R)$, so sind die Zustandsänderung und die Änderung des Inhalts der aktuell gelesenen Zelle wie bei $\delta(q, a) = (q', b, L)$, allerdings bewegt sich der Kopf nun um eine Zelle nach rechts.

Die Berechnung einer DTM wird so lange weitergeführt, bis der akzeptierende Zustand q_{accept} oder der ablehnende Zustand q_{reject} erreicht ist. Dann stoppt die Maschine. Wird keiner der Zustände $q_{\text{accept}}, q_{\text{reject}}$ erreicht, so hält die Maschine nie an, sie befindet sich dann in einer *Endlosrechnung.*

Da das Startsymbol $\triangleright$ in der ersten Zelle des Bandes nicht überschrieben werden kann und $\triangleright$ auch nie auf das Band geschrieben wird (siehe 5. in der Definition einer DTM), kann das linke Bandende einer DTM am Symbol $\triangleright$ in der äußersten linken Bandzelle erkannt werden. Wird das Symbol $\triangleright$ gelesen, und befindet sich der Kopf der DTM somit auf der äußersten linken Bandzelle, so muss der Kopf gemäß 5. in der Definition einer DTM im nächsten Schritt nach rechts bewegt werden. Der Kopf einer DTM kann also nie über das linke Bandende hinaus gehen.

Betrachten wir die Berechnungen der in Beispiel 1 beschriebenen DTM M. Bei jeder beliebigen Eingabe $w \in \Sigma^* = \{a\}^*$ wandert der Lesekopf einmal nach rechts über das Eingabeband bis es auf das erste $\sqcup$ trifft. Wird das erste $\sqcup$ gelesen, hält die DTM M. Wurde nun ein a gelesen, bevor die DTM auf das erste $\sqcup$ trifft, so hält M im akzeptierenden Zustand $q_2 = q_{\text{accept}}$. Folgt hingegen das erste $\sqcup$ unmittelbar auf das Startsymbol $\triangleright$, so hält M im ablehnenden Zustand $q_3 = q_{\text{reject}}$. Mit anderen Worten bei allen Eingaben der Form $a^n, n \geq 1$ hält die DTM M im akzeptierenden Zustand, nur bei der Eingabe $a^0 = \epsilon$ hält die DTM im ablehnenden Zustand.

Um die Arbeitsweise einer DTM formal zu fassen, benötigen wir zunächst den Begriff der *Konfiguration* einer DTM. Eine Konfiguration einer DTM M ist ein Element aus $\Gamma^* \times Q \times \Gamma^*$. *Die DTM M ist in Konfiguration $\alpha q \beta$* heißt:

- In den äußersten linken Zellen des Bandes steht $\alpha\beta \in \Gamma^*$. In den übrigen Zellen des Bandes stehen ausschließlich $\sqcup$'s,
- der Kopf steht auf dem ersten Buchstaben von β,
- die DTM befindet sich im Zustand q.

Beachten Sie, dass es Konfigurationen gibt, die niemals von einer DTM angenommen werden können. Ein Beispiel sind Konfigurationen, die mehr als ein Startsymbol $\triangleright$ enthalten. In

Abbildung 2.3 haben wir die Konfiguration einer DTM schematisch dargestellt.

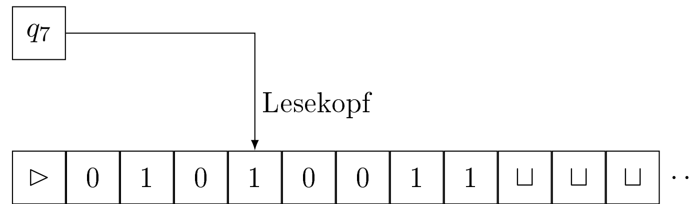


Abbildung 2.3: DTM in Konfiguration $\triangleright 010q_7 10011$

Sind K_1 und K_2 Konfigurationen einer DTM M, so *führt die Konfiguration* K_1 *zur Konfiguration* K_2, wenn die DTM durch einen Berechnungsschritt von der Konfiguration K_1 in die Konfiguration K_2 gelangt. Wir nennen K_2 dann auch *Nachfolgekonfiguration* von K_1. Seien $a, b, c \in \Gamma, \alpha, \beta \in \Gamma^*$ und $q_i, q_j \in Q$.

- Ist $K_1 = \alpha a q_i b \beta$ und $K_2 = \alpha q_j a c \beta$, so führt die Konfiguration K_1 zur Konfiguration K_2, wenn
$$\delta(q_i, b) = (q_j, c, L),$$
wenn also die DTM im Zustand q_i und bei gelesenem Symbol b, in den Zustand q_j wechselt, das Symbol b durch das Symbol c ersetzt und den Lesekopf nach links bewegt.
- Ist $K_1 = \alpha a q_i b \beta$ und $K_2 = \alpha a c q_j \beta$, so führt die Konfiguration K_1 zur Konfiguration K_2, wenn
$$\delta(q_i, b) = (q_j, c, R),$$
wenn also die DTM im Zustand q_i und bei gelesenem Symbol b, in den Zustand q_j wechselt, das Symbol b durch das Symbol c ersetzt und den Lesekopf nach rechts bewegt.
- Ist $K_1 = q_i b \beta$, so muss $b = \triangleright$ gelten. Führt dann K_1 zur Konfiguration K_2, so ist $K_2 = \triangleright q_j \beta$ und wie in der Definition von Turingmaschinen haben wir
$$\delta(q_i, \triangleright) = (q_j, \triangleright, R).$$

Beachten Sie, dass wir in einer Konfiguration für α oder β auch die leere Folge zulassen.
Ist $w \in \Sigma^*$ die Eingabe einer DTM, so nennen wir die Konfiguration $q_0 \triangleright w$ in der sich die DTM zu Beginn der Berechnung mit Eingabe w befindet, die *Startkonfiguration* bei Eingabe $w \in \Sigma^*$. Eine Konfiguration $\alpha q \beta$ mit $q = q_{\text{accept}}$ nennen wir *akzeptierende Konfiguration*. Ist $q = q_{\text{reject}}$, so nennen wir die Konfiguration *ablehnende Konfiguration*.

Wir kommen nun zu den grundlegende Begriffen der Berechenbarkeitstheorie.

Definition 2.2 Sei $M = (Q, \Sigma, \Gamma, \delta)$ eine DTM und $w \in \Sigma^*$.

1. Die DTM M *akzeptiert* w, falls es eine Folge von Konfigurationen $K_1, K_2, \ldots, K_l$ gibt, so dass
   (a) K_1 die Startkonfiguration von M bei Eingabe w ist,
   (b) die Konfiguration K_i zu Konfiguration K_{i+1} führt, $i = 1, \ldots, l-1$,
   (c) K_l eine akzeptierende Konfiguration ist.
2. Die DTM M *lehnt* w *ab*, falls es eine Folge von Konfigurationen $K_1, K_2, \ldots, K_l$ gibt, so dass

(a) K_1 die Startkonfiguration von M bei Eingabe w ist,

(b) die Konfiguration K_i zu Konfiguration K_{i+1} führt, $i = 1, \ldots, l-1$,

(c) K_l eine ablehnende Konfiguration ist.

Die von einer DTM M akzeptierten Worte fassen wir zu einer Menge oder Sprache zusammen.

Definition 2.3 Sei $M = (Q, \Sigma, \Gamma, \delta)$ eine DTM. Die Menge aller von M akzeptierten $w \in \Sigma^*$ ist die von M *akzeptierte Sprache* L. Wir schreiben dann auch $L = L(M)$.

Die von einer DTM akzeptierte Sprache ist stets eine Teilmenge der Menge aller endlichen Folgen über dem Eingabealphabet Σ der DTM M. Außerdem ist es wichtig, an dieser Stelle zu erkennen, dass wir in dieser Definition erlauben, dass M für $w \notin L$ nicht im ablehnenden Zustand q_{reject} hält, sondern möglicherweise in eine Endlosrechnung gerät. Deshalb ein, wie wir später sehen werden, stärkerer Begriff:

Definition 2.4 Sei $M = (Q, \Sigma, \Gamma, \delta)$ eine DTM. Die DTM M *entscheidet* die von ihr akzeptierte Sprache $L(M)$, wenn M alle Worte, die nicht in $L(M)$ liegen, ablehnt.

Definition 2.5 Sei Σ eine endliche Menge und $L \subseteq \Sigma^*$.

1. L heißt *rekursiv aufzählbar* genau dann, wenn es eine DTM mit Eingabealphabet Σ gibt, die L akzeptiert.

2. L heißt *rekursiv* bzw. *entscheidbar* genau dann, wenn es eine DTM mit Eingabealphabet Σ gibt, die L entscheidet.

Jetzt wollen wir die oben eingeführten Begriffe an einigen Beispielen deutlich machen. Zunächst zurück zu unserer DTM im Beispiel 1. Wie wir bereits gesehen haben, hält die DTM M bei Eingabe $a^n, n \geq 1$, im akzeptierenden Zustand q_{accept}, während bei Eingabe a^0 die DTM M im ablehnenden Zustand hält. Damit akzeptiert M alle Worte der Gestalt $a^n, n \geq 1$, und lehnt das Wort $a^0 = \epsilon$ ab. Die von M akzeptierte Sprache ist daher

$$L(M) = \Sigma^+ = \{a\}^+ = \{a^n \mid n \geq 1\}.$$

Da die DTM bei jeder Eingabe hält, entscheidet M diese Sprache auch.

Beispiel 2:

Wir wollen eine DTM konstruieren, die die Sprache $L = \{0^n 1^n \mid n \geq 1\}$ entscheidet.

Bei der Konstruktion einer DTM, die eine bestimmte Sprache entscheiden soll, ist es wie beim Programmieren wichtig, sich zunächst zu überlegen, welche Vorgehensweise die DTM umsetzen soll. Man braucht also zunächst eine informelle Beschreibung der DTM. Um die Sprache $L = \{0^n 1^n \mid n \geq 1\}$ zu entscheiden, wollen wir eine DTM konstruieren, die die beiden folgenden Schritte umsetzt.

1. Teste zuerst, ob die Eingabe von der Form $0^i 1^j$ ist (sonst nicht akzeptieren).

2. Falls die Eingabe von der Form $0^i 1^j$ ist, teste ob $i = j$ gilt, indem jeweils die erste 0 und letzte 1 gestrichen werden. Falls hierbei alle Nullen und Einsen gestrichen werden, so gilt $i = j$ und die Eingabe ist in L.

Folgende DTM mit $Q = \{q_0, \ldots, q_8\}$, $q_{\text{accept}} = q_7, q_{\text{reject}} = q_8$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \sqcup, \triangleright\}$ setzt die Beschreibung um und entscheidet L. Der Zustand q_0 prüft, ob es in der Eingabe mindestens eine 0 gibt. Die Zustände q_1 und q_2 prüfen ob $w = 0^i 1^j$ ist. Ist dieses sichergestellt, wird geprüft,

δ	0	1	$\sqcup$	$\triangleright$
q_0	$(q_1, 0, R)$	$(q_8, 1, R)$	$(q_8, \sqcup, R)$	$(q_0, \triangleright, R)$
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	$(q_8, \sqcup, R)$	$(q_8, \triangleright, R)$
q_2	$(q_8, 0, R)$	$(q_2, 1, R)$	$(q_3, \sqcup, L)$	$(q_8, \triangleright, R)$
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	$(q_4, \sqcup, R)$	$(q_4, \triangleright, R)$
q_4	$(q_5, \sqcup, R)$	$(q_8, 1, R)$	$(q_7, \sqcup, R)$	$(q_8, \triangleright, R)$
q_5	$(q_5, 0, R)$	$(q_5, 1, R)$	$(q_6, \sqcup, L)$	$(q_8, \triangleright, R)$
q_6	$(q_8, 0, R)$	$(q_3, \sqcup, L)$	$(q_8, \sqcup, R)$	$(q_8, \triangleright, R)$

Abbildung 2.4: Die Übergangsfunktion der DTM für $L = \{0^n1^n \mid n \geq 1\}$.

ob $i = j$ ist, indem jeweils die erste noch verbliebene 0 und letzte noch verblieben 1 durch ein $\sqcup$ ersetzt werden (Zustände q_4 bzw. q_6). Die Zustände q_3 und q_5 dienen zum Auffinden von verbliebenen 0 bzw. 1.

An diesem Beispiel wollen wir uns die aufeinanderfolgenden Konfigurationen der Maschine bei Eingabe 0011 anschauen. Die folgende Tabelle ist dabei spaltenweise zu lesen.

$q_0 \triangleright 0011\sqcup$	$\triangleright q_3 0011\sqcup$	$\triangleright \sqcup q_3 01 \sqcup \sqcup$
$\triangleright q_0 0011\sqcup$	$q_3 \triangleright 0011\sqcup$	$\triangleright q_3 \sqcup 01 \sqcup \sqcup$
$\triangleright 0 q_1 011\sqcup$	$\triangleright q_4 0011\sqcup$	$\triangleright \sqcup q_4 01 \sqcup \sqcup$
$\triangleright 00 q_1 11\sqcup$	$\triangleright \sqcup q_5 011\sqcup$	$\triangleright \sqcup \sqcup q_5 1 \sqcup \sqcup$
$\triangleright 001 q_2 1\sqcup$	$\triangleright \sqcup 0 q_5 11\sqcup$	$\triangleright \sqcup \sqcup 1 q_5 \sqcup \sqcup$
$\triangleright 0011 q_2 \sqcup$	$\triangleright \sqcup 01 q_5 1\sqcup$	$\triangleright \sqcup \sqcup q_6 1 \sqcup \sqcup$
$\triangleright 001 q_3 1\sqcup$	$\triangleright \sqcup 011 q_5 \sqcup$	$\triangleright \sqcup q_3 \sqcup \sqcup \sqcup \sqcup$
$\triangleright 00 q_3 11\sqcup$	$\triangleright \sqcup 01 q_6 1\sqcup$	$\triangleright \sqcup \sqcup q_4 \sqcup \sqcup\sqcup$
$\triangleright 0 q_3 011\sqcup$	$\triangleright \sqcup 0 q_3 1 \sqcup \sqcup$	$\triangleright \sqcup \sqcup \sqcup q_7 \sqcup \sqcup$

Abbildung 2.5: Berechnung der DTM für $L = \{0^n1^n \mid n \geq 1\}$ bei Eingabe 0011.

An diesem Beispiel wollen wir uns auch den Unterschied zwischen Akzeptieren und Entscheiden einer Sprache klar machen. Hierzu ändern wir die DTM von oben etwas ab. Die Übergangsfunktion δ sei nun definiert wie in Abbildung 2.6.

Der einzige Unterschied zur Übergangsfunktion vorher liegt in der zweiten Zeile in der fett herausgehobenen Stelle. Vorher waren wir an dieser Stelle in den ablehnenden Zustand $q_8 = q_{\text{reject}}$ übergegangen. Nun bleibt der Zustand derselbe und der Lesekopf geht eine Zelle weiter nach rechts. Vergleichen wir das Verhalten der beiden Maschinen bei Eingabe der Form 0^n. Unsere erste Maschine wird bei einer solchen Eingabe im Zustand $q_8 = q_{\text{reject}}$ halten und die Eingabe ablehnen. Die zweite Maschine hingegen wird bei einer solchen Eingabe nicht halten, stattdessen wird der Lesekopf auf dem Band immer weiter nach rechts wandern. Auf Eingaben, die nicht von der Form 0^n sind, werden die beiden Maschinen gleich arbeiten.

δ	0	1	$\sqcup$	$\triangleright$
q_0	$(q_1, 0, R)$	$(q_8, 1, R)$	$(q_8, \sqcup, R)$	$(q_0, \triangleright, R)$
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	$(\mathbf{q_1}, \sqcup, \mathbf{R})$	$(q_8, \triangleright, R)$
q_2	$(q_8, 0, R)$	$(q_2, 1, R)$	$(q_3, \sqcup, L)$	$(q_8, \triangleright, R)$
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	$(q_4, \sqcup, R)$	$(q_4, \triangleright, R)$
q_4	$(q_5, \sqcup, R)$	$(q_8, 1, R)$	$(q_7, \sqcup, R)$	$(q_8, \triangleright, R)$
q_5	$(q_5, 0, R)$	$(q_5, 1, R)$	$(q_6, \sqcup, L)$	$(q_8, \triangleright, R)$
q_6	$(q_8, 0, R)$	$(q_3, \sqcup, L)$	$(q_8, \sqcup, R)$	$(q_8, \triangleright, R)$

Abbildung 2.6: Alternative Übergangsfunktion einer DTM für $L = \{0^n 1^n \mid n \geq 1\}$.

Die erste Maschine entscheidet die Sprache $L = \{0^n 1^n \mid n \geq 1\}$, denn sie hält bei allen Eingaben, und sie hält im Zustand $q_7 = q_{\text{accept}}$ genau auf Eingaben aus der Sprache L. Die zweite Maschine hält zwar auch auf allen Eingaben aus L im Zustand $q_7 = q_{\text{accept}}$, allerdings gibt es nicht in L liegende Eingaben, bei denen die Maschine in eine Endlosschleife gerät. Die zweite Maschine akzeptiert daher L, aber sie entscheidet L nicht.

Beispiel 3:
Gesucht: Eine DTM, die $L = \{0^{2^n} \mid n \geq 0\}$ entscheidet.

Idee:

1. Gehe über die Eingabe von links nach rechts und streiche dabei jede zweite Null.
2. Wird im ersten Schritt festgestellt, dass es nur noch eine 0 auf dem Band gibt, akzeptiere.
3. Wird im ersten Schritt festgestellt, dass es noch mehr als eine 0, aber eine ungerade Anzahl von 0 auf dem Band gibt, lehne ab.
4. Gehe wieder zum Beginn des Bandes
5. Zurück zum ersten Schritt.

Folgende DTM mit $Q = \{q_0, \ldots, q_6\}$, $q_{\text{accept}} = q_5, q_{\text{reject}} = q_6$, $\Sigma = \{0\}$, $\Gamma = \{0, x, \sqcup, \triangleright\}$ setzt diese Idee um. Das Streichen einer 0 wird umgesetzt, indem die 0 durch ein x ersetzt wird. Zustand q_1 dient zur Überprüfung, ob nur noch eine 0 auf dem Band ist. Die Zustände q_2, q_3 dienen dazu, jede zweite 0 zu streichen. Gleichzeitig wird mit ihnen überprüft, ob die Anzahl der auf dem Band verbliebenen 0 gerade oder ungerade ist. Zustand q_4 schließlich realisiert das Zurückgehen zum Beginn des Bandes.

δ	0	x	$\sqcup$	$\triangleright$
q_0	$(q_1, 0, R)$	$(q_6, \sqcup, R)$	$(q_6, \sqcup, R)$	$(q_0, \triangleright, R)$
q_1	(q_2, x, R)	(q_1, x, R)	$(q_5, \sqcup, L)$	$(q_6, \triangleright, R)$
q_2	$(q_3, 0, R)$	(q_2, x, R)	$(q_4, \sqcup, L)$	$(q_6, \triangleright, R)$
q_3	(q_2, x, R)	(q_3, x, R)	$(q_6, \sqcup, R)$	$(q_6, \triangleright, R)$
q_4	$(q_4, 0, L)$	(q_4, x, L)	$(q_6, \sqcup, R)$	$(q_0, \triangleright, R)$

Abbildung 2.7: Übergangsfunktion der DTM für die Sprache $L = \{0^{2^n} \mid n \geq 0\}$.

An diesem Beispiel wollen wir uns die aufeinanderfolgenden Konfigurationen der Maschine bei Eingabe 0000 anschauen. Die Tabelle in Abbildung 2.8 ist spaltenweise zu lesen.

$q_0 \rhd 0000\sqcup$	$q_4 \rhd 0x0x\sqcup$	$q_4 \rhd 0xxx\sqcup$
$\rhd q_0 0000\sqcup$	$\rhd q_0 0x0x\sqcup$	$\rhd q_0 0xxx\sqcup$
$\rhd 0 q_1 000\sqcup$	$\rhd 0 q_1 x0x\sqcup$	$\rhd 0 q_1 xxx\sqcup$
$\rhd 0x q_2 00\sqcup$	$\rhd 0x q_1 0x\sqcup$	$\rhd 0x q_1 xx\sqcup$
$\rhd 0x0 q_3 0\sqcup$	$\rhd 0xx q_2 x\sqcup$	$\rhd 0xx q_1 x\sqcup$
$\rhd 0x0x q_2\sqcup$	$\rhd 0xxx q_2\sqcup$	$\rhd 0xxx q_1\sqcup$
$\rhd 0x0 q_4 x\sqcup$	$\rhd 0xx q_4 x\sqcup$	$\rhd 0xx q_5 x\sqcup$
$\rhd 0x q_4 0x\sqcup$	$\rhd 0x q_4 xx\sqcup$	
$\rhd 0 q_4 x0x\sqcup$	$\rhd 0 q_4 xxx\sqcup$	
$\rhd q_4 0x0x\sqcup$	$\rhd q_4 0xxx\sqcup$	

Abbildung 2.8: Berechnung der DTM für $L = \{0^{2^n} \mid n \geq 0\}$ bei Eingabe 0000.

Beispiel 4: Ein Palindrom $w = w_1 w_2 \cdots w_n \in \Sigma^*$ über einem Alphabet Σ ist ein Wort, das von hinten nach vorn gelesen wieder das Wort w ergibt. In Formeln, es muss gelten $w_i = w_{n-i+1}$ für $i = 1, \ldots, n$.

Gesucht: Eine DTM, die $L = \{w \in \{0,1\}^* \mid w \text{ ist ein Palindrom}\}$ entscheidet.

Idee:

1. Lese das erste Zeichen in w.
2. Merke Dir dieses Zeichen und ersetze es durch ein $\sqcup$.
3. Gehe zum letzten Zeichen, das kein $\sqcup$ ist, überprüfe, ob dieses Zeichen mit dem im vorangegangenen Schritt gemerkten Zeichen übereinstimmt und ersetze es durch ein $\sqcup$.
4. Gehe zum am weitesten links liegenden Zeichen, das kein $\sqcup$ ist und gehe zum zweiten Schritt zurück.

Folgende DTM mit $Q = \{q_0, \ldots, q_7\}$, $q_6 = q_{\text{accept}}, q_7 = q_{\text{reject}}$, $\Sigma = \{0,1\}$, $\Gamma = \{0,1,\sqcup,\rhd\}$ entscheidet L.
Turingmaschinen können auch Funktionen berechnen.

Definition 2.6 Sei $M = (Q, \Sigma, \Gamma, \delta)$ eine DTM. Die DTM M berechnet die Funktion $f : \Sigma^* \to (\Gamma \backslash \{\sqcup, \rhd\})^*$, falls die Berechnung von M bei jeder Eingabe $w \in \Sigma^*$ in einer akzeptierenden Konfiguration hält und dabei der Bandinhalt $f(w)$ ist. Hierbei werden das Startsymbol $\rhd$ und alle Blanks des Bandes ignoriert.

Wir werden uns in dieser Vorlesung hauptsächlich mit der Entscheidbarkeit von Sprachen und nicht so sehr mit der Berechenbarkeit von Funktionen beschäftigen. Einerseits ist die

δ	0	1	$\sqcup$	$\rhd$
q_0	$(q_1, \sqcup, R)$	$(q_2, \sqcup, R)$	$(q_6, \sqcup, R)$	$(q_0, \rhd, R)$
q_1	$(q_1, 0, R)$	$(q_1, 1, R)$	$(q_3, \sqcup, L)$	$(q_7, \rhd, R)$
q_2	$(q_2, 0, R)$	$(q_2, 1, R)$	$(q_4, \sqcup, L)$	$(q_7, \rhd, R)$
q_3	$(q_5, \sqcup, L)$	$(q_7, 1, L)$	$(q_6, \sqcup, L)$	$(q_7, \rhd, R)$
q_4	$(q_7, 0, L)$	$(q_5, \sqcup, L)$	$(q_6, \sqcup, L)$	$(q_7, \rhd, R)$
q_5	$(q_5, 0, L)$	$(q_5, 1, L)$	$(q_0, \sqcup, R)$	$(q_7, \rhd, R)$

Abbildung 2.9: Übergangsfunktion der DTM für Palindrome.

Entscheidbarkeit von Sprachen leichter zu fassen als die Berechenbarkeit von Funktionen, andererseits lassen sich aber auch viele Probleme mit Hilfe von Sprachen definieren bzw. auf Sprachen zurückführen. Jedoch wird die Berechenbarkeit von Funktionen bei dem für die Berechenbarkeits- und Komplexitätstheorie grundlegendem Begriff der Reduktion eine wichtige Rolle spielen. Wir werden im Zusammenhang mit Reduktion dann genauer auf den Begriff der berechenbaren Funktion eingehen.

2.2 Programmiertechniken

In diesem Abschnitt werden wir einige einfache Programmiertechniken für Turingmaschinen kennenlernen. Viele dieser Techniken werden zu Techniken aus der Programmierung etwa mit Java ähnlich sein. Dann werden wir Mehrband Turingmaschinen als ein alternatives Rechenmodell definieren. Die Programmiertechniken werden wir in *Simulationen* nutzen, um etwa zu zeigen, dass Mehrband Turingmaschinen nicht mächtiger sind als die 1-Band Turingmaschinen, die wir bislang kennen gelernt haben.

2.2.1 Endlicher Speicher - „Im Zustand merken“

Am letzten Beispiel des vorangegangenen Abschnitts haben wir gesehen, dass man sich gelesene Buchstaben mit Hilfe des Zustandes merken kann. Geht man im Beispiel für Palindrome aus dem Zustand q_0 in den Zustand q_1, so war das gelesene Zeichen eine 0. Geht man hingegen in den Zustand q_2, so war das gelesene Zeichen eine 1. Allgemein kann man sich endlich viele verschiedene Informationen im Zustand einer Turingmaschine merken. Formal kann die Information, die die TM sich merken kann, durch Elemente $a \in A$ eines endlichen Alphabets A beschrieben werden. Hat man eine Turingmaschine M, die sich die verschiedenen Informationen aus A noch nicht merken kann, so kann man die Maschine erweitern zu einer Maschine, die sich Elemente aus A merken kann, indem die Zustandsmenge Q von M um die Zustandsmenge $Q \times A$ erweitert wird.

Beispiel: Wir wollen testen, ob bei Eingabe $w_1 \dots w_n \in \Sigma^+$ der Buchstabe w_1 in $w_2 \dots w_n$ vorkommt. Wir merken uns w_1 im Zustand. Sei $\Gamma = \Sigma \cup \{\rhd, \sqcup\}$, $Q = (\{q_0\} \times \Sigma) \cup \{q_0, q_1, q_2\}$, $q_{\text{accept}} = q_1$ und $q_{\text{reject}} = q_2$

$$
\begin{array}{llcll}
1) & \delta(q_0, \rhd) & = & (q_0, \rhd, R) & \\
2) & \delta(q_0, \sqcup) & = & (q_2, \sqcup, L) & \\
3) & \delta(q_0, a) & = & ([q_0, a], a, R) & \forall a \in \Sigma \\
4) & \delta([q_0, a], a) & = & (q_1, a, L) & \forall a \in \Sigma \\
5) & \delta([q_0, a], b) & = & ([q_0, a], b, R) & \forall a, b \in \Sigma, a \neq b \\
6) & \delta([q_0, a], \sqcup) & = & (q_2, \sqcup, L) & \\
7) & \delta([q_0, a], \rhd) & = & (q_2, \rhd, R) &
\end{array}
$$

2.2.2 Markieren von Buchstaben und Bandpositionen

Jetzt werden wir sehen, wie man einzelne Eingabebuchstaben und deren Position markieren kann, um sie später wiederfinden zu können. Sei Σ das Eingabealphabet. Wollen wir z.B. Auftreten des Buchstabens $a \in \Sigma$ markieren können, so können wir in das Bandalphabet Γ zusätzlich den Buchstaben $\hat{a}$ aufnehmen.

Beispiel: Das folgende Problem wird *Element-Distictness* genannt. Die Eingabe besteht aus Worten $w_i \in \{0,1\}^*, i = 1, \dots, l$, die durch $\sharp$ voneinander getrennt sind. Es soll entschieden werden, ob alle w_i unterschiedlich sind.

Gesucht ist also eine DTM M, die die Sprache

$$
\mathit{ElDist} = \{\sharp w_1 \sharp w_2 \sharp \cdots \sharp w_l \mid w_i \in \{0,1\}^* \text{ und } w_i \neq w_j \text{ für alle } i \neq j\}
$$

entscheidet.

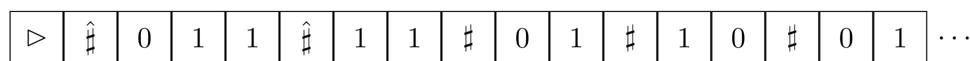
Die im Folgenden beschriebene TM entscheidet diese Sprache, indem sie zunächst w_1 mit $w_2, w_3, \cdots, w_l$ vergleicht. Dann vergleicht sie w_2 mit $w_3, \cdots, w_l$, usw. Um sich zu merken, welche w_i, w_j sie gerade miteinander vergleicht, und wie sie dann weiter vorzugehen hat, benutzt die TM markierte Buchstaben $\hat{\sharp}$.

1. Falls das äußerste linke Eingabesymbol nicht $\sharp$ ist, lehne ab, sonst ersetze $\sharp$ durch $\hat{\sharp}$.
2. Finde das nächste $\sharp$ rechts vom ersten $\sharp$ und ersetze es durch $\hat{\sharp}$. Wird kein weiteres $\sharp$ gefunden, akzeptiere.
3. Laufe zwischen den beiden Folgen w_i, w_j in $\{0,1\}^*$, die rechts von den markierten $\sharp$ stehen, hin und her, um zu entscheiden, ob die Folgen gleich sind. Wenn ja, lehne ab.
4. Entferne die Markierung auf dem rechten der beiden $\hat{\sharp}$. Markiere das $\sharp$, das am nächsten rechts von dem gerade unmarkierten $\sharp$ steht. Falls es kein $\sharp$ mehr gibt, entferne die Markierung auf dem linken markierten $\sharp$ und markiere das $\sharp$ rechts hiervon. Markiere auch das nächste rechts vom gerade markierten $\sharp$ gelegene $\sharp$. Falls kein $\sharp$ mehr vorhanden ist, akzeptiere.
5. Gehe zum dritten Schritt.

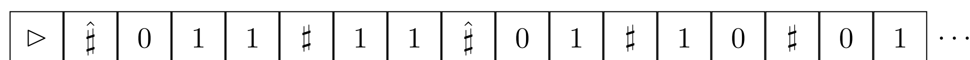
Für das Beispiel

$$l = 5, w_1 = 011, w_2 = 11, w_3 = 01, w_4 = 10, w_5 = 01$$

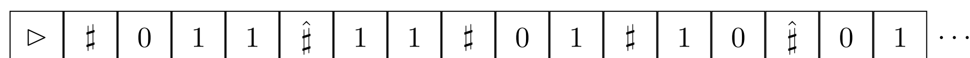
sind einige Beispiele von markierten $\sharp$ in Abbildung 2.10 angegeben.



Markierte $\sharp$ bei Vergleich von w_1 und w_2



Markierte $\sharp$ bei Vergleich von w_1 und w_3



Markierte $\sharp$ bei Vergleich von w_2 und w_5

Abbildung 2.10: Markierte Positionen bei Element-Distinctness

Um den dritten Schritt zu realisieren, ist das Merken von Buchstaben im Zustand sehr hilfreich. Um w_i und w_j miteinander zu vergleichen, merkt man sich zunächst den ersten Buchstaben von w_i, markiert und vergleicht ihn mit dem ersten Buchstaben von w_j, der ebenfalls markiert wird. Dann merkt man sich den zweite Buchstaben von w_i, markiert ihn und vergleicht ihn mit dem zweiten Buchstaben von w_j, usw. Auf eine formale Beschreibung der Turingmaschine für Element-Distinctness verzichten wir.

Die beiden Techniken, die wir gerade kennengelernt haben, werden sehr nützlich sein, um zu zeigen, dass 1-Band Turingmaschinen dieselben Sprachen entscheiden/akzeptieren können wie andere Rechenmodelle, die auf den ersten Blick vielleicht mächtiger erscheinen als 1-Band Turingmaschinen.

2.3 Mehrband Turingmaschinen und Simulationen

Wir wollen jetzt das Modell der Turingmaschine etwas erweitern. Schon in den Beispielen, die wir bislang gesehen haben, wäre es häufig nützlich gewesen, ein zweites Band zur Verfügung zu haben, um sich Information besser merken zu können. Einige der Sprachen hätte man dann sicher eleganter entscheiden können. Wir wollen untersuchen, ob man mit Turingmaschinen, die mehr als ein Band zur Verfügung haben, vielleicht mehr Sprachen entscheiden oder akzeptieren kann als mit Turingmaschinen mit nur einem Band. Mit Hilfe von *Simulationen* werden wir sehen, dass dieses nicht der Fall ist. Simulationen sind eine wichtige Technik der Theoretischen Informatik, sie werden in dieser Vorlesung eine entscheidende Rolle spielen. So kann mit Simulationen nicht nur gezeigt werden, dass 1-Band Turingmaschinen genauso mächtig sind wie Mehrband Turingmaschinen, sie werden auch im Beweis der Existenz unentscheidbarer Sprachen eine wichtige Rolle spielen.

Definition 2.7 Eine k-Band Turingmaschine (k-DTM) hat nicht nur ein Band und einen Lesekopf, sondern k Bänder mit je einem Lesekopf. Die Übergangsfunktion ist dann von der Form $\delta : Q\backslash\{q_{\text{accept}}, q_{\text{reject}}\}\times\Gamma^k \to Q\times\Gamma^k\times\{R, L, S\}^k$. Zu Beginn steht die Eingabe auf Band 1, auf allen anderen Bändern steht links das Startsymbol $\triangleright$ gefolgt von Blanks. Die Arbeitsweise einer k-DTM ist wie die Arbeitsweise einer 1-Band-DTM definiert. Allerdings erlauben wir, dass bei einer k-DTM in einem Rechenschritt einige Leseköpfe sich nicht bewegen. Dieses wird in der Übergangsfunktion durch das Symbol S gekennzeichnet.

Ist etwa

$$\delta(q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, L, R, \dots, L),$$

so bedeutet dieses, das die Maschine, falls sie sich im Zustand q_i befindet und auf dem i-ten Band ein a_i liest, im nächsten Schritt in den Zustand q_j übergeht, auf dem i-ten Band a_i durch b_i ersetzt, auf dem ersten Band den Lesekopf eine Position nach links bewegt, auf dem zweiten Band den Lesekopf eine Position nach rechts bewegt, usw.

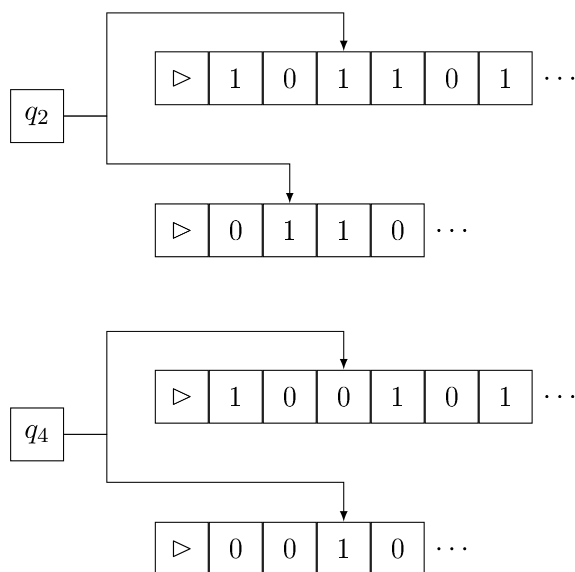


Abbildung 2.11: $\delta(q_2, 1, 1) = (q_4, 0, 1, S, R)$

Wie bei 1-Band DTMs ist die Konfiguration einer Mehrband Turingmaschine eine Momentaufnahme der Turingmaschine. Sie besteht aus dem aktuellen Zustand, dem Inhalt sämtlicher

Bänder sowie der Position aller Leseköpfe. Wir verzichten auf eine eigene Notation für Konfigurationen von Mehrband Turingmaschinen. Ein Beispiel für die Konfiguration einer Mehrband TM ist in Abbildung 2.12 dargestellt. Hier ist der Zustand q_3, die Bandinhalte sind $\triangleright$11011100 und $\triangleright$0110111 (bei Ignorieren der Blanks am Ende der jeweiligen Bänder). Die Positionen der Köpfe sind wie bei 1-Band TMs durch Pfeile gekennzeichnet.

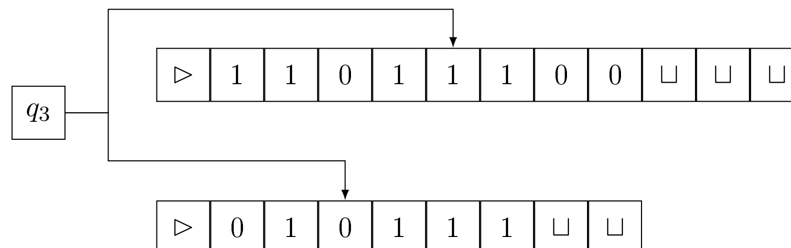


Abbildung 2.12: Konfiguration einer Mehrband Turingmaschinen

Berechnungen von Mehrband Turingmaschinen sind dann wieder definiert als Folgen von Konfigurationen. Eine akzeptierende (ablehnende) Konfiguration ist eine Konfiguration im akzeptierenden (ablehnenden) Zustand. Damit können wir wie im Fall von 1-Band Turingmaschinen definieren, wann eine k-DTM ein Wort akzeptiert bzw. ablehnt. Schließlich können wir die von einer k-DTM akzeptierte oder entschiedene Sprache wie in Definition 2.3 und in Definition 2.4 definieren.

Um nun zu zeigen, dass Mehrband Turingmaschinen nur Sprachen entscheiden oder akzeptieren können, die auch durch 1-Band Turingmaschinen entschieden oder akzeptiert werden, zeigen wir, dass jede Mehrband Turingmaschine durch eine 1-Band Turingmaschine *simuliert* werden kann. Unter einer Simulation einer Turingmaschine M durch eine Turingmaschine S verstehen wir dabei den Entwurf einer Maschine S, die das Verhalten von M nachahmt. Genauer, wir werden die DTM S so konstruieren, dass die beiden folgenden Bedingungen erfüllt sind.

1. Die DTM S kann auf ihrem Band Konfigurationen der Mehrband-DTM M speichern. Wir sagen, dass S *Kodierungen von Konfigurationen* von M speichert.

2. Für jede Konfiguration K von M und deren Nachfolgekonfiguration $\bar{K}$ kann die DTM S durch eine endliche Anzahl von Schritten die Kodierung von K durch die Kodierung der Nachfolgekonfiguration $\bar{K}$ von K ersetzen. Wir sagen, dass wir den Schritt von M, der M von der Konfiguration K in die Konfiguration $\bar{K}$ überführt, durch eine endliche Anzahl von Schritten von S *simulieren* können.

Satz 2.8 *Wird die Sprache $L \subseteq \Sigma^*$ von einer k-Band Turingmaschine M entschieden (akzeptiert), so gibt es auch eine 1-Band Turingmaschine, die L entscheidet (akzeptiert).*

Beweis: Zunächst erklären wir, wie Konfigurationen von M auf dem Band von S gespeichert werden können.

Kodierung einer Konfiguration: Das Eingabealphabet von S wird das Eingabealphabet Σ von M sein. Das Bandalphabet von S wird nicht nur die Buchstaben des Bandalphabets Γ von M enthalten, sondern auch noch einen Buchstaben $\sharp \notin \Gamma$. Um den Inhalt der Bänder von M zu speichern, schreibt nun S diese Inhalte hintereinander auf sein Band, wobei die Inhalte der einzelnen Bänder von M durch den Buchstaben $\sharp$ getrennt sind. Die Teile der Bänder, die nur noch aus Blanks bestehen, werden nicht mit gespeichert. Damit müssen für jedes Band

von M nur endlich viele Symbole gespeichert werden und die beschriebene Abspeicherung der Inhalte der Bänder von M auf dem Band von S ist möglich. Hinter den Inhalt des k-ten Bandes von M setzt S schließlich auch noch ein $\sharp$.

S muss sich aber auch die Positionen der k Leseköpfe von M merken. Hierzu benutzt S die Technik des Markierens von Positionen, die wir oben schon erklärt haben. S wird daher für jeden Buchstaben $a \in \Gamma$ einen weiteren Buchstaben $\hat{a}$ in seinem Bandalphabet besitzen. Zu jedem Zeitpunkt wird S zwischen dem Startsymbol und dem ersten $\sharp$ sowie zwischen zwei aufeinanderfolgenden $\sharp$ in genau einer Zelle höchstens einen Buchstaben der Form $\hat{a}$ speichern. Dieser markiert dann die Position des Lesekopfes von M auf dem entsprechenden Band. Steht der Lesekopf auf dem Startsymbol des entsprechenden Bandes, so steht zwischen den entsprechenden $\sharp$'s auf dem Band von S kein markierter Buchstabe. In Abbildung 2.13 haben wir die Speicherung der Inhalte und Lesekopfpositionen einer 2-Band Turingmaschine auf dem Band einer 1-Band Turingmaschine dargestellt.

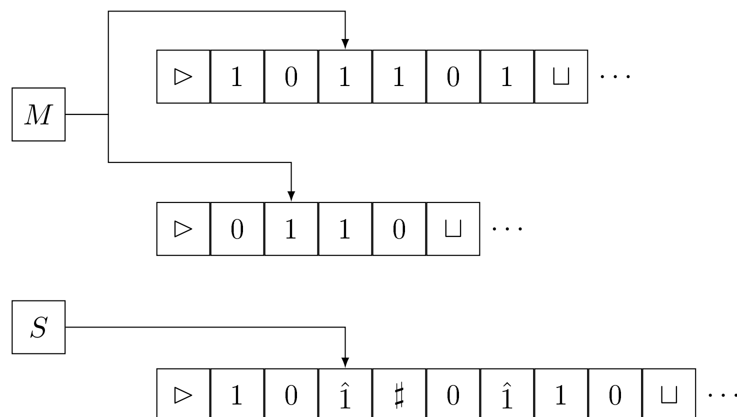


Abbildung 2.13: Speicherung von 2 Bändern durch 1 Band

Schließlich können wir uns den Zustand von M im Zustand von S merken (siehe oben).

Nun zeigen wir wie die einzelnen Schritte der Mehrband-DTM M durch eine Folge von Schritten der DTM S simuliert werden können.

Simulation eines Schrittes: Sei $w = w_1 w_2 \cdots w_n \in \Sigma^*$ die Eingabe von M. Die Simulation durch S erfolgt dann folgendermaßen.

1. Zunächst bringt S sein Band in die Form

$$\triangleright w_1 w_2 \cdots w_n \sharp\sharp\sharp \cdots \sharp,$$

   dabei haben wir genau k Symbole $\sharp$, wobei das i-te $\sharp$ das Ende des Inhalts des i-ten Bands von M kennzeichnet. Danach geht S in die Konfiguration

$$[p_1, (q_0, \triangleright, \triangleright, \ldots, \triangleright)] \triangleright w_1 w_2 \cdots w_n \sharp\sharp\sharp \cdots \sharp,$$

   um die Simulation zu starten.

2. Um einen einzelnen Schritt von M zu simulieren, durchläuft S einmal sein komplettes Band, startend bei $\triangleright$ und endend beim k-ten $\sharp$. Dabei merkt es sich im Zustand die k Buchstaben, auf denen die Leseköpfe von M zurzeit stehen. Konkret befindet sich S im Zustand $[p_i, (q, a_1, \ldots, a_k)]$, bis wieder ein markierter Buchstabe (oder $\sharp$ für den

Fall, dass $a_i = \triangleright$ ist) erreicht worden ist. Sobald ein markierter Buchstabe erreicht worden ist (welcher mit a_i übereinstimmen sollte), simuliert S die Rechnung von M auf $(q, a_1, \ldots, a_k)$ für Band i, markiert den Buchstaben, auf dem M danach in Band i stehen würde, und geht danach in den Zustand $[p_{i+1}, (q, a_1, \ldots, a_k)]$ über. Ist S am k-ten $\sharp$ angekommen, ersetzt S q durch den Nachfolgezustand von M bei Eingabe $(q, a_1, \ldots, a_k)$ und läuft wieder zurück zum Startsymbol $\triangleright$. Dabei werden die Zelleninhalte der Zellen mit markierten Buchstaben in die a_i-Werte im Zustand von S übernommen, so dass der nächste Schritt von M simuliert werden kann.

3. Sollte S bei der Simulation wie gerade beschrieben versuchen, eine Kopfbewegung von M dadurch zu simulieren, dass es ein $\sharp$ markieren will, so bedeutet dieses, dass M in diesem Schritt auf bislang ungenutzte Teile eines seiner Bänder zugreifen will (Lesekopf hat sich nach rechts bewegt) oder aber das Startsymbol eines seiner Bänder erreicht hat (Lesekopf hat sich nach links bewegt). Den ersten Fall simuliert S, indem es alle Buchstaben, beginnend mit dem $\sharp$, das markiert werden müsste, um eine Position nach rechts verschiebt. In der nun frei gewordenen Zelle links vom ersten verschobenen $\sharp$ schreibt S einen markierten Blank $\hat{\sqcup}$. Den zweiten Fall simuliert S, indem $\sharp$ nicht markiert wird, in Übereinstimmung mit der Vereinbarung, dass zwischen zwei $\sharp$ kein markiertes Symbol steht, wenn der Lesekopf des entsprechenden Bandes das Startsymbol $\triangleright$ liest.

Somit ist gezeigt, dass wir jede Mehrband Turingmaschine durch eine 1-Band Turingmaschine simulieren können. Insbesondere kann jede Sprache, die durch eine Mehrband Turingmaschine entschieden oder akzeptiert wird auch durch eine 1-Band Turingmaschine entschieden oder akzeptiert werden. □

Wie 1-Band Turingmaschinen können auch Mehrband DTMs Funktionen berechnen. Bei Mehrband DTMs gehen wir dabei immer davon aus, dass das Ergebnis der Berechnung (der Funktionswert) auf dem letzten Band der Maschine steht.

Definition 2.9 Sei $M = (Q, \Sigma, \Gamma, \delta)$ eine k-DTM. Die k-DTM M berechnet die Funktion $f : \Sigma^* \to (\Gamma \backslash \{\sqcup, \triangleright\})^*$, falls die Berechnung von M bei jeder Eingabe $w \in \Sigma^*$ in einer akzeptierenden Konfiguration hält und dabei der Bandinhalt des k-ten Bands $f(w)$ ist. Hierbei werden das Startsymbol $\triangleright$ und alle Blanks des Bandes ignoriert.

Der Beweis von Satz 2.8 liefert nun auch, dass es für jede Funktion f, die durch eine k-Band DTM berechnet wird, auch eine 1-Band DTM gibt, die f berechnet.

2.4 Die Churchsche These

Die Churchsche These (auch Church-Turing-These genannt) wurde 1936 von dem Mathematiker und Begründer der Theoretischen Informatik Alonzo Church aufgestellt. Sie lautet:
Die im intuitiven Sinne berechenbaren Funktionen und Sprachen sind genau die, die durch Turingmaschinen berechenbar sind.
Diese These ist nicht beweisbar, da „im intuitiven Sinne berechenbar“ nicht formalisiert werden kann. Es gibt aber viele Zugänge zur Formalisierung des Begriffs, die alle äquivalent sind. Zwei haben wir kennengelernt: 1-Band und Mehrband Turingmaschinen. Es gibt aber noch viele andere Zugänge zum Begriff der Berechenbarkeit. In den 30er Jahren schlug z.B. Church das λ-Kalkül und Kleene die μ-Rekursion vor, und später unter anderem Markov die Markov-Algorithmen und von Neumann die von Neumann Maschine. Es kann nicht nur gezeigt werden, dass 1-Band und Mehrband Turingmaschinen zu denselben berechenbaren Funktionen und

Sprachen führen. Mit Hilfe von Simulationen kann auch gezeigt werden, dass alle anderen jemals vorgeschlagenen Formalisierungen von Berechenbarkeit (inklusive Quantencomputern) äquivalent zur Berechenbarkeit durch eine Turingmaschine sind. Obwohl also nicht beweisbar, ist die Churchsche These gut fundiert und allgemein akzeptiert.

2.5 Eigenschaften entscheidbarer und rekursiv aufzählbarer Sprachen

Zum Abschluss dieses Abschnitts wollen wir einige wichtige Eigenschaften entscheidbarer und rekursiv aufzählbarer Sprachen herleiten. Da aus Satz 2.8 folgt, dass eine Sprache genau dann entscheidbar (rekursiv aufzählbar) ist, wenn sie von einer Mehrband-DTM entschieden (akzeptiert) werden kann, werden wir in den Beweisen Mehrband-DTMs nutzen.

Satz 2.10 *Seien L_1, L_2 entscheidbare Sprachen, dann gilt:*

(i) $\bar{L}_1$ *ist entscheidbar.*

(ii) $L_1 \cap L_2$ *ist entscheidbar.*

(iii) $L_1 \cup L_2$ *ist entscheidbar.*

Dieses sind sogenannte *Abschlusseigenschaften*: die Klasse der entscheidbaren Sprachen ist abgeschlossen unter Komplement, Durchschnitt und Vereinigung.
Beweis: Seien $M_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1), M_2 = (Q_2, \Sigma_2, \Gamma_2, \delta_2)$ 1-Band DTMs die die Sprachen L_1 bzw. L_2 entscheiden.
(i) Um eine DTM $\bar{M}$ zu erhalten, die $\bar{L}_1$ entscheidet, vertauschen wir in der Beschreibung für M_1 nur die Zustände q_{accept} und q_{reject}. Da M_1 bei jeder Eingabe halten soll (M_1 entscheidet L_1), heißt dieses, dass $\bar{M}$ ein Wort x genau dann akzeptiert, wenn M_1 dieses Wort nicht akzeptiert.
(ii) Wir konstruieren die 2-Band-DTM M, die wie folgt arbeitet:

- Kopiere den Input auf Band 2.
- Arbeite *parallel* wie M_1 auf Band 1 und wie M_2 auf Band 2.

Formaler ist M folgendermaßen definiert: Die Zustandsmenge Q von M enthält $Q_1 \times Q_2$. Weiter enthält die Menge Q Zustände, die für das Kopieren der Eingabe auf das zweite Band zuständig sind. Schließlich definieren wir für $q_1 \in Q_1, q_2 \in Q_2$, alle $a_1 \in \Sigma_1, a_2 \in \Sigma_2$ und $D_i \in \{L, R, S\}$ die Übergangsfunktion δ von M durch: $\delta([q_1, q_2], a_1, a_2) = ([p_1, p_2], b_1, b_2, D_1, D_2)$, falls $\delta_1(q_1, a_1) = (p_1, b_1, D_1)$ und $\delta_2(q_2, a_2) = (p_2, b_2, D_2)$. Weiter sind Regeln vorhanden, die dafür sorgen, dass, wenn M_1 bereits im Zustand q_{accept} gehalten hat, die Rechnung von M_2 weitergeführt wird, und umgekehrt.
M akzeptiert eine Eingabe, falls M_1 und M_2 beide die Eingabe akzeptieren, d. h. der Zustand $[q_{\text{accept}}, q_{\text{accept}}]$ erreicht wird.
(iii) Analog zu (ii). □

Satz 2.11 *Seien L_1 und L_2 rekursiv aufzählbar, dann gilt:*

(i) $L_1 \cup L_2$ *ist rekursiv aufzählbar.*

(ii) $L_1 \cap L_2$ *ist rekursiv aufzählbar.*

Beweis: genau wie $(ii), (iii)$ oben. □

Der Beweis von (i) aus Satz 2.10 kann nicht übertragen werden, da ja nun die DTM für $x \notin L_1$ nicht halten muss!
Als nächstes wollen wir untersuchen, welche Beziehungen zwischen entscheidbar und rekursiv aufzählbar bestehen.

Satz 2.12 *L ist genau dann entscheidbar, wenn L und $\bar{L}$ rekursiv aufzählbar sind.*

Beweis: „$\Rightarrow$“ :
Wir zeigen, dass sowohl L als auch $\bar{L}$ rekursiv aufzählbar sind.

1. L entscheidbar $\Rightarrow$ L rekursiv aufzählbar, denn rekursiv aufzählbar ist die schwächere Eigenschaft.

2. L entscheidbar (Satz 2.10) $\Rightarrow$ $\bar{L}$ entscheidbar $\Rightarrow$ $\bar{L}$ rekursiv aufzählbar.

„$\Leftarrow$“:
Seien M_1, M_2 die 1-Band-DTMs, die L bzw. $\bar{L}$ akzeptieren. Wir lassen M_1 und M_2 wieder parallel arbeiten, wie im Beweis von Satz 2.10 (ii). Die so beschriebene DTM M hält, sobald eine der beiden Maschinen im akzeptierenden Zustand hält. Sie akzeptiert, falls M_1 akzeptiert und verwirft, falls M_2 akzeptiert. Beachte hierbei, dass M für *jeden* Input x hält. Denn es gilt immer, dass $x \in L$ oder $x \in \bar{L}$. Damit hält bei Input x immer genau eine der beiden Turingmaschinen M_1 und M_2 im akzeptierenden Zustand q_{accept}. □

2.6 Universelle Turingmaschinen

Die bislang beschriebenen DTMs sind *special purpose computers*, d. h. sie können nur ein Problem lösen. Das entspricht der historischen Entwicklung, denn erste Anwendungen von Computern waren special-purpose-Anwendungen: Ballistikprobleme in den USA, Dechiffrieren des deutschen ENIGMA-Codes in England, Flügelvermessung der V1- und V2-Rakete in Deutschland (Zuse).

Erst später wurden programmierbare (auch universell genannte) Rechner entwickelt. Bei den programmierbaren Rechnern besteht die Eingabe aus einem Programm und einer Eingabe für das Programm. Dabei ist das Programm eine Beschreibung einer (virtuellen) special purpose Maschine.

Um programmierbare oder universelle Rechner durch Turingmaschinen zu formalisieren, führen wir zunächst eine kanonische Beschreibung für Turingmaschinen ein.

2.6.1 Beschreibung einer Turingmaschine - Gödelnummern

Wir beschränken uns auf Turingmaschinen mit Eingabealphabet $\Sigma = \{0, 1\}$ und Bandalphabet $\Gamma = \{0, 1, \rhd, \sqcup\}$. In einem Exkurs am Ende dieses Abschnitts zeigen wir, dass man damit alle Turingmaschinen erfasst, wenn andere Alphabete geeignet codiert werden. Weiter beschränken wir uns auf 1-Band Turingmaschinen. Mehrband Turingmaschinen können durch diese simuliert werden (siehe Satz 2.8).

Definition 2.13 Sei M eine 1-Band-DTM mit

$$Q = \{q_0, \ldots, q_n\}, q_{\text{accept}} = q_{n-1}, q_{\text{reject}} = q_n, \text{Startzustand } s = q_0.$$

Sei $X_1 \hat{=} 0, X_2 \hat{=} 1, X_3 \hat{=} \sqcup, X_4 \hat{=} \rhd, D_1 \hat{=} L, D_2 \hat{=} R$. Wir kodieren einen Eintrag

$$\delta(q_i, X_j) = (q_k, X_l, D_m)$$

der Funktionstabelle der Übergangsfunktion δ durch

$$0^{i+1}10^j10^{k+1}10^l10^m.$$

Wir nummerieren die Einträge der Tabelle für δ mit den Zahlen 1 bis $(n-1)|\Gamma| = 4(n-1)$ durch. Die Nummerierung erfolgt von links nach rechts und beginnt in der ersten Zeile. Es gibt zwar $n+1$ Zustände, aber für den akzeptierenden Zustand q_{accept} und den ablehnenden Zustand q_{reject} gibt es nichts zu codieren. $Code_t$ sei die Codierung des t-ten Eintrags, $1 \leq t \leq 4(n-1)$. Die **Gödelnummer** von M ist

$$\langle M \rangle = 111Code_1 11Code_2 11Code_3 \ldots 11Code_g 111,$$

wobei $g = 4(n-1)$.

Die Bezeichnung *Gödelnummer* geht auf den deutschen Mathematiker Kurt Gödel zurück und ist motiviert durch die Tatsache, dass wir die obige 0-1-Folge auch als Binärdarstellung einer Zahl auffassen können. Sie sollten sich davon überzeugen, dass aus der Gödelnummer einer Turingmaschine ihre Übergangsfunktion rekonstruiert werden kann. Hierzu überlegen Sie sich insbesondere, was zwei aufeinanderfolgende Einsen bedeuten.

Beispiel: Wir betrachten eine Turingmaschine M, die die Sprache

$$L = \{w \in \{0, 1\}^* \mid w \text{ enthält } \geq 2 \text{ Einsen}\}$$

entscheidet. Die DTM M hat Zustände $q_0, q_1, q_2 = q_{\text{accept}}, q_3 = q_{\text{reject}}$. Die Tabelle der Übergangsfunktion δ sieht folgendermaßen aus.

δ	0	1	$\sqcup$	$\rhd$
q_0	$(q_0, 0, R)$	$(q_1, 1, R)$	$(q_3, \sqcup, L)$	$(q_0, \rhd, R)$
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	$(q_3, \sqcup, L)$	$(q_3, \rhd, R)$

Da $n = 3$ müssen also $4 \cdot 2 = 8$ Einträge codiert werden. Die folgende Tabelle enthält die Codierung dieser 8 Einträge.

Nummer des Eintrags	Eintrag	Codierung
1	$\delta(q_0, 0) = (q_0, 0, R)$	0101010100
2	$\delta(q_0, 1) = (q_1, 1, R)$	0100100100100
3	$\delta(q_0, \sqcup) = (q_3, \sqcup, L)$	0100010000100010
4	$\delta(q_0, \rhd) = (q_0, \rhd, R)$	0100001010000100
5	$\delta(q_1, 0) = (q_1, 0, R)$	001010010100
6	$\delta(q_1, 1) = (q_2, 1, R)$	001001000100100
7	$\delta(q_1, \sqcup) = (q_3, \sqcup, L)$	00100010000100010
8	$\delta(q_1, \rhd) = (q_3, \rhd, R)$	0010000010000010000100

Die Gödelnummer dieser Maschine ist nun

$$\langle M \rangle = \begin{array}{l} 111010101010011010010010010011010001000010001011 \\ 010000010100000100110010100101001100100100010010011 \\ 0010001000010001011001000001000001000000100111 \end{array}$$

Als natürliche Zahl betrachtet ist die Gödelnummer dieser Maschine

638779882761580251873009425399934115415079.

Definition 2.14 Eine Turingmaschine M_0 heißt **universell**, falls für jede 1-Band-DTM M und jedes $x \in \{0,1\}^*$ gilt:

- M_0 gestartet mit $\langle M \rangle x$ hält genau dann, wenn M gestartet mit x hält.
- Falls M gestartet mit x hält, berechnet M_0 gestartet mit $\langle M \rangle x$ die gleiche Ausgabe wie M gestartet mit x. Insbesondere akzeptiert M_0 die Eingabe $\langle M \rangle x$ genau dann, wenn M die Eingabe x akzeptiert.

Satz 2.15 *Es gibt eine universelle 2-Band-DTM M_0.*

Beweis: Gegeben eine beliebige DTM M zeigen wir, wie M_0 Konfigurationen von M kodiert, und wie wir einzelne Schritte von M auf M_0 simulieren können. Zu Beginn der Simulation wird der Beginn von x bestimmt. Nach Definition der Gödelnummer zeigt das zweite Auftauchen von 111 das Ende von $\langle M \rangle$ an. Die Maschine kopiert dann zunächst x auf ihr zweites Band. Sie bewegt den Kopf des zweiten Bandes auf das erste Zeichen in x. Dann löscht sie x auf dem ersten Band und schreibt 0, die Kodierung des Startzustandes q_0 hinter $\langle M \rangle$. Dann bewegt M_0 den Kopf des ersten Bandes zurück zum linken Ende des Bandes. Von nun an folgt die Kodierung von Konfigurationen und die Simulation der Schritte den Regeln:

Kodierung einer Konfiguration Eine Konfiguration $\alpha q_i X_j \beta$ von M wird von M_0 wie folgt Kodiert:

- Auf Band 1 steht $\langle M \rangle$ und der Zustand q_i, kodiert durch 0^{i+1}.
- Auf Band 2 steht die Bandinschrift $\alpha X_j \beta$ von M.
- Der Kopf von Band 2 steht auf X_j.

Simulation eines Schritts

- Suche auf Band 1 in $\langle M \rangle$ die Zeichenreihe $110^{i+1}10^{j}1$. Diese bildet den Anfang der Kodierung des Übergangs $\delta(q_i, X_j)$. Dazu kann $j \in \{1,2,3,4\}$ im Zustand gespeichert werden, 0^{i+1} wird durch Vergleich mit der Kodierung von q_i ($\hat{=} 0^{i+1}$, auf Band 1 hinter $\langle M \rangle$) gefunden. Beachten Sie, dass q_i nicht im Zustand gespeichert werden kann. (Warum nicht?)

- Lese die dahinterstehende Zeichenreihe der Form $0^{k+1}10^{l}10^{m}$, ersetze hinter $\langle M \rangle$ die Kodierung 0^{i+1} von q_i durch 0^{k+1}, die Kodierung des neuen Zustands q_k, und speichere l, m im Zustand. Damit ist im Zustand die Information über den auszuführenden nächsten Schritt gespeichert: „überschreibe die Zelle der Kopfposition mit X_l und bewege den Kopf gemäß D_m“. Man beachte, dass der Zustandswechsel von q_i nach q_k bereits hinter $\langle M \rangle$ auf Band 1 gespeichert ist. Der Zustandswechsel muss nicht (und kann auch nicht) im Zustand gespeichert werden.

- Verändere Band 2 entsprechend dem im Zustand gespeicherten Befehl.

Ende der Simulation und Berechnung Halte, falls der Zustand $q_{n-1} = q_{\text{accept}}$ oder der Zustand $q_n = q_{\text{reject}}$ von M auf Band 1 hinter $\langle M \rangle$ gespeichert ist. Akzeptiere, falls der akzeptierende Endzustand q_{n-1} von M auf Band 1 hinter $\langle M \rangle$ steht. □

2.6.2 Eigenschaften von Turingmaschinen als entscheidbare bzw. rekursiv aufzählbare Sprachen

Da wir DTMs mit Hilfe der Gödelnummern mit Elementen aus $\{0,1\}^*$ identifizieren, können wir nun auch Eigenschaften von Turingmaschinen mit Hilfe von Sprachen definieren. Weiter können wir uns dann fragen, ob diese Sprachen entscheidbar oder rekursiv aufzählbar sind.

Beispiel 1: Die einfachste Frage in Bezug auf Turingmaschinen ist sicherlich, ob etwas eine korrekt definierte DTM ist. Als Sprache über dem Eingabealphabet $\{0,1\}$ erhalten wir dann

$$\text{Gödel} := \{w \in \{0,1\}^* \mid w \text{ ist die Gödelnummer einer DTM } M.\}$$

Lemma 2.16 *Die Sprache Gödel ist entscheidbar.*

Beweis: Zunächst überprüfen wir, ob die Eingabefolge w das richtige Format hat, d.h., die Folge beginnt und endet mit drei Einsen, zwischen zwei Teilfolgen bestehend aus jeweils zwei Einsen tauchen 4 Einsen auf, zwischen denen wiederum jeweils eine gewisse Anzahl von Nullen steht. Ist sichergestellt, dass die Folge w das richtige Format hat, müssen wir prüfen, dass alle Forderungen, die an eine DTM gestellt werden, erfüllt sind. Z.B. müssen wir überprüfen, dass beim Lesen von $\triangleright$ die Übergangsfunktion festlegt, dass der Lesekopf nach rechts wandert. Für die Folge w bedeutet dieses, dass bei jeder Teilfolge $0^{i+1}10^{j}10^{k+1}0^{l}10^{m}$ mit $j = 4$ (das gelesene Zeichen ist $\triangleright$) gelten muss, dass $m = 2$ (der Kopf bewegt sich nach rechts). Diese Eigenschaft einer Folge kann sicherlich leicht durch eine Turingmaschine überprüft werden. Man überzeugt sich, dass auch alle anderen Forderungen an eine DTM leicht anhand der Gödelnummer überprüft werden können. □

Beispiel 2: Wir wollen wissen, ob eine DTM eine gewisse Anzahl von Zuständen besitzt. Als Sprache formuliert erhalten wir

$$\text{States} := \{(\langle M\rangle, d) \mid \text{Die DTM } M \text{ besitzt mindestens } d \text{ Zustände, } d \in \mathbb{N}.\}$$

Lemma 2.17 *Die Sprache States ist entscheidbar.*

Beweis: Zunächst einmal ist klar, dass wir eine DTM konstruieren können, die alle Folgen ablehnt, die nicht Paar einer Gödelnummer und einer natürlichen Zahl sind. Ist hingegen die Eingabe ein Paar bestehend aus der Gödelnummer $\langle M\rangle$ und der natürlichen Zahl d, so können wir anhand der Gödelnummer leicht feststellen, ob M mindestens d Zustände besitzt. Hierzu müssen wir nur überprüfen, ob die Gödelnummer eine Übergangsfunktion kodiert, die mindestens aus $d-2$ Zeilen besteht. Es müssen nur mindestens $d-2$ Zeilen sein, da für die Zustände $q_{\text{accept}}, q_{\text{reject}}$ die Übergangsfunktion einer DTM nicht definiert ist. □

Beispiel 3 - Das Halteproblem: Nun kommen wir zu einem Beispiel, das nicht nur für diese Vorlesung sondern auch für die Praxis von großer Bedeutung ist. Es handelt sich hierbei um das Halteproblem, das wir als Sprache immer mit H abkürzen werden:

$$H := \{\langle M\rangle x \mid M \text{ ist DTM, die gestartet mit Eingabe } x \text{ hält.}\}.$$

In die etwas modernere Terminologie übersetzt, fragen wir uns bei gegebenem Computerprogramm und gegebener Eingabe für das Programm, ob das Programm bei dieser Eingabe hält oder in eine Endschlosschleife gerät. Ein Problem, das offensichtlich sehr praxisrelevant ist. Wir wollen uns nun überlegen, dass das Halteproblem rekursiv aufzählbar ist. Später werden wir sehen, dass das Halteproblem nicht entscheidbar ist.

Satz 2.18 *Das Halteproblem ist rekursiv aufzählbar.*

Beweis: Um zu zeigen, dass das Halteproblem rekursiv aufzählbar ist, müssen wir eine DTM $\bar{M}$ konstruieren, die die Sprache H akzeptiert. Die DTM ist ein Abbildung 2.14 beschrieben. Die Beschreibung ist in einer Form angeben, die wir von nun an immer wieder zur Beschreibung von DTMs verwenden werden.

$\bar{M}$ bei Eingabe $w \in \{0,1\}^*$

1. Falls w nicht von der Form $\langle M\rangle x$ ist, lehne ab.
2. Simuliere M mit Eingabe x.
3. Wird in 2. festgestellt, dass M die Eingabe x akzeptiert, akzeptiere $\langle M\rangle x$.

Abbildung 2.14: DTM $\bar{M}$, die H akzeptiert.

Um den ersten Schritt umzusetzen, erinnern wir uns, dass wir das Ende der Gödelnummer an der zweiten Teilfolge von drei aufeinanderfolgend Einsen erkennen können. Existiert eine solche Teilfolge nicht, ist die Eingabe nicht von der geforderten Form, und wir können ablehnen. Sonst können wir wie in Beispiel 1 überprüfen, ob die Teilfolge beginnend mit den ersten drei Einsen und endend mit den zweiten drei Einsen eine Gödelnummer ist. Den 2. Schritt bis 4. Schritt können wir mit der universellen Turingmaschine M_0 umsetzen.
Wir müssen noch zeigen, dass die DTM $\bar{M}$ die Sprache H akzeptiert. Hierzu ist für $w = \langle M\rangle x$ zweierlei zu zeigen.

i) Ist $w \in H$, so akzeptiert $\bar{M}$ die Eingabe w.

ii) Ist $w \notin H$, so akzeptiert $\bar{M}$ die Eingabe w nicht.

zu i) Ist $w \in H$, so hält M bei Eingabe x. Dann wird auch $\bar{M}$ bei Eingabe $w = \langle M \rangle x$ halten. Aus 3. und 4. der Beschreibung von $\bar{M}$ folgt, dass $\bar{M}$ dann w akzeptiert.

zu ii) Ist $w \notin H$, so hält M bei Eingabe x nicht. Damit kann auch $\bar{M}$ bei Eingabe w nicht halten. Dieses bedeutet, dass $\bar{M}$ die Eingabe w nicht akzeptiert.

□

Man beachte, dass die DTM $\bar{M}$ die Sprache H wirklich nur akzeptiert und nicht entscheidet. Denn ist die Eingabe $w = \langle M \rangle x$ für $\bar{M}$ derart, dass M bei Eingabe x nicht hält, so wird $\bar{M}$ bei Eingabe $w = \langle M \rangle x$ ebenfalls nicht halten. Wie oben schon gesagt, werden wir auch sehen, dass H nicht entscheidbar ist.

Beispiel 4: Wir betrachten die folgende Sprache

$$\text{Useful} := \left\{ (\langle M \rangle, q) \;\middle|\; \begin{array}{l} M \text{ ist DTM mit Zustand } q, \text{ und es gibt eine Eingabe} \\ w, \text{ so dass } M \text{ gestartet mit } w \text{ den Zustand } q \text{ erreicht.} \end{array} \right\}$$

Auch diese Sprache ist praktisch von Bedeutung. Dieses wird klarer, wenn wir das Problem etwas umformulieren. Gegeben ist ein Computerprogramm bestehend aus mehreren Unterprogrammen. Wir wählen eines der Unterprogramme aus, und fragen uns, ob dieses Unterprogramm jemals von irgendeiner Eingabe aufgerufen wird. Wir zeigen nun

Lemma 2.19 *Useful ist rekursiv aufzählbar.*

Beweis: Um eine DTM zu konstruieren, die Useful akzeptiert, werden wir eine nützliche Technik kennenlernen. Die Technik beruht darauf, dass wir die Elemente in $\{0,1\}^*$ aufzählen oder indizieren können. Diese Aufzählung ist gegeben durch die folgende bijektive Funktion index:

$$\begin{aligned} \text{index}\colon \{0,1\}^* &\to \mathbb{N} \\ v &\mapsto (1v)_2. \end{aligned}$$

In Worten, wir stellen der Bitfolge w zunächst eine 1 voran und erhalten dann eine neue Folge $1w$. Diese Folge interpretieren wir als die Binärdarstellung einer natürlichen Zahl. Diese Zahl wiederum ist der Index, die wir der Folge w zuordnen. Die Folge, der wir auf diese Weise den Index i zuordnen, bezeichnen wir mit w_i. Es gilt also

$$\forall w \in \{0,1\}^* : w = w_{\text{index}(w)}.$$

Betrachten wir einige Beispiele. Der leeren Folge ϵ ordnen wir den Index zu, deren Binärdarstellung $1\epsilon = 1$ ist. Dieses ist die Zahl 1. Der Folge 00 ordnen wir den Index mit der Binärdarstellung 100 zu. Dieses ist die Zahl 4 und es gilt $\text{index}(00) = 4$.

Wir ordnen auf die oben beschriebene Weise nicht nur jeder Folge einen Index zu. Umgekehrt gibt es auch zu jeder Zahl ≥ 1 eine Folge mit diesem Index. Um die Folge mit Index i zu erhalten, betrachten wir die Binärdarstellung von i, streichen die führende 1 und erhalten so die Folge mit Index i. Betrachten wir $i = 13$. Die Binärdarstellung von 13 ist 1101. Damit ist $w_{13} = 101$.

Nun können wir eine DTM E beschreiben, die die Sprache Useful akzeptiert.

E bei Eingabe $w \in \{0,1\}^*$

1. Lehne ab, falls w nicht von der Form $(\langle M \rangle, q)$ ist,
   wobei q ein Zustand von M ist.
2. Wiederhole für $i = 1, 2, 3, \ldots$ Schritte bis die DTM hält:
3. Simuliere für i Schritte die DTM M mit Eingabe $w_1, \ldots, w_i$.
4. Wird in 3. festgestellt, dass M bei einer der Eingaben nach
   höchstens i Schritten den Zustand q erreicht, akzeptiere.

Der 1. Schritt ist wieder leicht durchzuführen. Um die Simulationen im 3. Schritt durchzuführen, können wir wieder vorgehen wie bei der universellen Turingmaschinen. Diese statten wir aber jetzt noch zusätzlich mit einem Zähler aus, der jeweils sagt, wie viele Schritte bereits simuliert wurden. Im i-ten Durchlauf der Schleife in 2. wird dann jede Simulation abgebrochen, sobald der Zähler den Wert i erreicht hat.

Wir müssen nun zeigen, dass E die Sprache Useful akzeptiert. Eingaben, die nicht von der korrekten Form sind, werden im 1. Schritt abgelehnt. Ist die Eingabe für E von der Form $(\langle M \rangle, q)$, wobei q ein Zustand von M ist, M aber bei keiner Eingabe w in den Zustand q geht, wird E nicht halten. Vielmehr wird die Schleife in 2. für $i = 1, 2, 3, 4, \ldots$ durchlaufen.

Ist hingegen $(\langle M \rangle, q) \in$ Useful, so gibt es mindestens eine Folge $w_j \in \{0,1\}^*$, so dass M gestartet mit Eingabe w_j in den Zustand q geht. Erreicht nun M gestartet mit Eingabe w_j den Zustand nach k Schritten, so betrachten wir den Schleifendurchlauf im 2. Schritt von E für den Wert $i = \max\{j, k\}$. Hat E vorher noch nicht im akzeptierenden Zustand gehalten, wird nun M mit Eingabe w_j für mindestens k Schritte simuliert. Dann aber wird festgestellt, dass M bei Eingabe w_j den Zustand q erreicht. E hält dann im akzeptierenden Zustand.

Wir haben also gezeigt, dass E Eingaben, die nicht in Useful liegen, nicht akzeptiert, während alle Eingaben, die in Useful liegen auch von E akzeptiert werden. E akzeptiert somit die Sprache Useful. $\square$

Wir wollen nun noch kurz die Frage klären, warum wir die Laufzeit für die Simulationen im 3. Schritt von E sukzessive erhöhen. Warum können wir nicht nacheinander M mit Eingabe $w_1, w_2, w_3, \ldots$ simulieren, um zu sehen, ob M bei einer Eingabe jemals den Zustand q erreicht? Betrachten wir hierzu den Fall, dass M bei Eingabe w_2 den Zustand q erreicht, M aber bei Eingabe w_1 nicht hält. Simulieren wir nun M nacheinander mit allen Eingaben, ohne die Zeit der Simulation zu beschränken, werden wir nie zur Simulation von M mit Eingabe w_2 gelangen, denn bei Eingabe w_1 gerät M, und damit auch die Simulation von M, in eine Endlosschleife. $(\langle M \rangle, q)$ liegt dann zwar in Useful, würde aber von E nicht akzeptiert werden.

Exkurs: Einschränkung auf die Standardalphabete Wir haben zu Beginn dieses Abschnitts vor der Definition von Gödelnummern behauptet, dass wir uns bei DTMs auf die so genannten Standardalphabete $\Gamma = \{0, 1, \triangleright, \sqcup\}$, $\Sigma = \{0, 1\}$ beschränken können. Dieses wollen wir nun begründen. Die Einschränkung auf die Standardalphabete wird sich auch im weiteren Verlauf der Vorlesung als nützlich erweisen. Beliebige Alphabete Γ' können durch $\{0, 1, \sqcup\}$ simulieren werden, indem wir die Buchstaben $\neq \sqcup$ aus Γ' mit $1, \ldots, r$ durchnummerieren und den i-ten Buchstaben durch i Nullen gefolgt von einer 1 kodieren. Als Beispiel betrachten wir $\Gamma' = \{a, b, c, d, \triangleright, \sqcup\}$. Die Elemente aus Γ' werden dann durch $01, 001, 0001, 00001, \triangleright, \sqcup$ kodiert. Die letzte 1 fügen wir immer an, um das Ende der Kodierung einer Buchstabens erkennen zu können. Es gibt bessere Kodierungen, aber die gerade beschriebene reicht für unsere Zwecke. Von nun an werden wir daher immer annehmen $\Sigma = \{0, 1\}, \Gamma = \{0, 1, \triangleright, \sqcup\}$.

2.7 Die Existenz nicht rekursiv aufzählbarer Sprachen

Als Nächstes wollen wir nachweisen, dass es Sprachen gibt, die nicht rekursiv aufzählbar sind, d.h. es gibt keine DTM, die diese akzeptiert. Dazu gehen wir zu den Ursprüngen der Komplexitätstheorie zurück, die in der Begründung der modernen Mengenlehre liegen.

Die moderne Mengenlehre geht im Prinzip auf eine Person zurück, Georg Cantor. In seiner bahnbrechenden Forschungsarbeit im Jahre 1874 beobachtete Cantor, dass es verschiedene Arten von Unendlichkeit gibt. Davor hatte sich die Forscherwelt nie mit dieser Problematik auseinandergesetzt sondern glaubte schlicht, dass es nur eine Art von Unendlichkeit gibt. Cantor zeigte, dass das nicht wahr ist. Insbesondere zeigte er, dass die Kardinalität der reellen Zahlen größer ist als die Kardinalität der natürlichen Zahlen. Wir werden seinen Beweis in diesem Kapitel vorstellen und darauf aufbauend nachweisen, dass es nicht rekursiv aufzählbare Sprachen geben muss.

2.7.1 Eigenschaften unendlicher Mengen

Die *Kardinalität* einer Menge ist definiert als die Anzahl ihrer Elemente. Wenn S eine unendliche Menge ist, ist $|S|$ unendlich groß. In diesem Fall bezeichnet man die Kardinalität einer Menge auch als eine *transfinite Zahl* oder *transfinite Kardinalität.*

Zwei Mengen können in eine eins-zu-eins Beziehung gebracht werden (bzw. sie sind *isomorph*) genau dann wenn ihre Elemente gepaart werden können, so dass jedes Element der zwei Mengen genau einem Paar zugeordnet ist, oder mit anderen Worten, wenn es eine Bijektion von einer Menge zur anderen gibt. Falls zwei Mengen A und B isomorph sind, so schreiben wir $A \simeq B$.

Beispiel: Die Mengen $A = \{a, b, c\}$ und $B = \{1, 2, 3\}$ können offensichtlich in eine eins-zu-eins Beziehung gebracht werden, da wir die Paare $(a, 1)$, $(b, 2)$ und $(c, 3)$ formen können, oder mit anderen Worten, die Funktion $f : A \to B$ mit $f(a) = 1$, $f(b) = 2$ und $f(c) = 3$ ist eine Bijektion.

Die Kardinalität einer Menge A ist *mindestens so groß* wie die Kardinalität von B, oder $|A| \geq |B|$, genau dann wenn es eine Bijektion einer Teilmenge von A auf die Menge B gibt, oder mit anderen Worten, wenn es eine sujektive Abbildung von A nach B gibt. Wir sagen, dass A und B *dieselbe Kardinalität* haben, oder $|A| = |B|$, genau dann wenn $|A| \geq |B|$ und $|B| \geq |A|$. Es ist einfach zu überprüfen, dass „=“ eine Äquivalenzrelation und „≥“ eine Ordnung darstellt. Weiterhin gilt der folgende Satz.

Satz 2.20 *Seien A und B beliebige Mengen. Falls $A \simeq B$, dann ist $|A| = |B|$.*

Beweis: Falls $A \simeq B$, dann gibt es eine bijektive Abbildung f von A nach B und eine bijektive Abbildung g von B nach A. Da bijektive Abbildungen surjektiv sind, folgt aus f, dass $|A| \geq |B|$, und aus g, dass $|B| \geq |A|$. Somit ist $|A| = |B|$. □

Die wichtigsten Mengen sind

- $\mathbb{N} = \{1, 2, 3, 4, \ldots\}$: Menge der *natürlichen Zahlen*
- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$: Menge der *ganzen Zahlen*
- $\mathbb{Q} = \{x/y \mid x \in \mathbb{Z} \text{ und } y \in \mathbb{N}\}$: Menge der *rationalen Zahlen*
- $\mathbb{R}$: Menge der *reellen Zahlen*

Gegeben eine Zahlenmenge S, sei S_+ die Menge aller positiven Zahlen in S und S_- die Menge aller negativen Zahlen in S.
Es mag naheliegen, dass

$$|\mathbb{N}| = |\mathbb{Z}| = |\mathbb{Q}| = |\mathbb{R}| \quad \text{oder} \quad |\mathbb{N}| < |\mathbb{Z}| < |\mathbb{Q}| < |\mathbb{R}|$$

ist. Allerdings werden wir zeigen, dass das nicht wahr ist.

Satz 2.21 $|\mathbb{N}| = |\mathbb{Z}|$.

Beweis: Aufgrund der Tatsache, dass $\mathbb{N} \subseteq \mathbb{Z}$ ist, oder konkreter, die Abbildung $g : \mathbb{Z} \to \mathbb{N}$ mit $g(x) = 1 + |x|$ surjektiv ist, folgt direkt, dass auch $|\mathbb{Z}| \geq |\mathbb{N}|$ ist. Um $|\mathbb{N}| \geq |\mathbb{Z}|$ zu zeigen, betrachten wir die Abbildung $f : \mathbb{N} \to \mathbb{Z}$ mit $f(x) = (-1)^x \cdot \lfloor x/2 \rfloor$. f ist surjektiv, da für jedes $y \in \mathbb{Z}$ der Wert $x = 2 \cdot |y| + (|y| - y)/(2|y|)$ (falls $y \neq 0$ und sonst $x = 1$) die Eigenschaft hat, dass $f(x) = y$. Also ist $|\mathbb{N}| = |\mathbb{Z}|$. □

Satz 2.22 $|\mathbb{Z}| = |\mathbb{Q}|$.

Beweis: Offensichtlich ist $\mathbb{Z} \subseteq \mathbb{Q}$ und damit $|\mathbb{Q}| \geq |\mathbb{Z}|$. Es bleibt also zu zeigen, dass $|\mathbb{Z}| \geq |\mathbb{Q}|$ ist. Wir nehmen hier vereinfachend an, dass alle Brüche $x/y \neq 0$ in $\mathbb{Q}$ verschiedene Zahlen darstellen (was offensichtlich für 4/2 und 6/3 nicht gilt). Können wir für diese Vereinfachung Surjektivität nachweisen, gilt sie offensichtlich auch für alle Zahlen, die durch $\mathbb{Q}$ abgedeckt sind. Die folgende Tabelle illustriert Cantors Methode, um diese Ungleichung zu zeigen.

		Zähler				
		1	2	3	4	
	1	1/1	2/1	3/1	4/1	...
Nenner	2	1/2	2/2	3/2	4/2	...
	3	1/3	2/3	3/3	4/3	...
	4	1/4	2/4	3/4	4/4	...
			...			

Offensichtlich kann jede positive rationale Zahl in dieser Tabelle gefunden werden. Wir können diese Zahlen rigoros aufzählen, indem wir diagonal durch die Tabelle laufen: 1/1, 2/1, 1/2, 3/1, 2/2, 1/3, usw. Wenn wir nun darauf aufbauend die folgende Abbildung $g : \mathbb{Z}_+ \to \mathbb{Q}_+$ definieren, erhalten wir eine surjektive Abbildung von $\mathbb{Z}_+$ nach $\mathbb{Q}_+$.

$\mathbb{Z}_+$	1	2	3	4	5	6	7	...
$\mathbb{Q}$	1/1	2/1	1/2	3/1	2/2	1/3	4/1	...

Mit derselben Strategie können wir auch eine surjektive Abbildung h von $\mathbb{Z}_-$ nach $\mathbb{Q}_-$ erzeugen. Definieren wir nun die Abbildung $f : \mathbb{Z} \to \mathbb{Q}$ mit $f(x) = g(x)$ falls $x > 0$, $f(x) = h(x)$ falls $x < 0$ und $f(0) = 0$, dann erhalten wir eine surjektive Abbildung von $\mathbb{Z}$ nach $\mathbb{Q}$. Daher ist $|\mathbb{Z}| \geq |\mathbb{Q}|$ und damit $|\mathbb{Z}| = |\mathbb{Q}|$. □

Da die Gleichheit transitiv ist, erhalten wir aus den Sätzen 2.21 und 2.22 das folgende Resultat.

Korollar 2.23 $|\mathbb{N}| = |\mathbb{Q}|$.

Eine Verallgemeinerung der Aufzählungstechnik aus dem Beweis von Satz 2.22 liefert auch das folgende Resultat.

Satz 2.24 *Für jedes* $k \in \mathbb{N}$ *ist* $|\mathbb{N}| = |\mathbb{N}^k|$.

Im Lichte dieses Resultats mag es naheliegend sein zu glauben, dass alle unendliche Mengen dieselbe Kardinalität haben. Die nächsten zwei Sätze zeigen allerdings, dass das nicht wahr ist. Für eine beliebige Menge M ist $\mathcal{P}(M) = \{U \mid U \subseteq M\}$ die *Potenzmenge* von M.

Satz 2.25 $|\mathbb{R}| = |\mathcal{P}(\mathbb{N})|$.

Beweis: Zunächst zeigen wir, dass $|\mathbb{R}| \geq |\mathcal{P}(\mathbb{N})|$. Für jedes (binär kodierte) $x \in [0,1]$ sei $b(x)$ der Binärstring, der genau dann eine 1 in Position i enthält, wenn die i-te Ziffer rechts vom Dezimalpunkt von x eine 1 ist. Für $x = 0,625$ ist z.B. $(x)_2 = 0,101$ und daher $b(x) = 101$. Dabei gehen wir hier implizit davon aus, dass x eine eindeutige Binärkodierung hat. Das stimmt aber leider nicht ganz, da $x = 0,625$ zwei verschiedene Binärkodierungen hat, nämlich $(x)_2 = 0,101$ und $(x)_2 = 0,100\bar{1}$ (also 0,100 gefolgt von unendlich vielen 1en). Das liegt daran, dass $\sum_{i \geq k} 1/2^i = 1/2^{k-1}$ ist. Wir unterscheiden zwischen diesen Binärkodierungen, indem wir $(x)_2 = 0,101$ als die *kanonische* Binärkodierung bezeichnen. Sei nun $b(x)$ der Binärstring, der sich aus der kanonischen Binärkodierung ergibt, und $b'(x)$ der Binärstring, der sich aus der alternativen Binärkodierung ergibt. $b'(x)$ existiert natürlich nur, wenn $x > 0$ und $b(x)$ endlich ist, sonst nehmen wir an, dass $b'(x) = b(x)$. Für den Sonderfall, dass $x = 1$ ist, setzen wir $b(x) = b'(x) = \bar{1}$ (unendlich viele 1en). Für einen Binärstring $w = w_1 w_2 w_3 \ldots$ sei $S_w \subseteq \mathbb{N}$ die Menge, die genau die Zahlen $i \in \mathbb{N}$ enthält mit $w_i = 1$. Betrachte nun die Abbildung $f : [0,1] \times \{0,1\} \to \mathcal{P}(\mathbb{N})$ mit

$$f(x,i) = \begin{cases} S_{b(x)} & \text{falls } i = 0 \\ S_{b'(x)} & \text{falls } i = 1 \end{cases}$$

Wir werden zeigen, dass f surjektiv ist. Betrachte eine beliebige Menge $S \subseteq \mathcal{P}(\mathbb{N})$. Falls $|S|$ endlich ist, dann gilt für $x = \sum_{i \in S} 1/2^i \in [0,1]$, dass $S_{b(x)} = S$ bzw. $f(x,0) = S$. Falls $|S|$ unendlich ist, unterscheiden wir zwischen drei Fällen.

- Falls $S = \mathbb{N}$, dann ist $f(1,0) = f(1,1) = S$.
- Ist $|\bar{S}|$ endlich (und nichtleer), dann gilt für die größte Zahl k in $\bar{S}$, dass für alle $k' > k$, $k' \in S$. Sei $S' = \{i \in S \mid i < k\} \cup \{k\}$. Dann gilt für die reelle Zahl $x' \in [0,1]$ mit $S_{b(x')} = S'$, dass $S_{b'(x')} = S$ und damit $f(x',1) = S$.
- Falls auch $|\bar{S}|$ unendlich ist, dann gilt für $x = \sum_{i \in S} 1/2^i \in [0,1]$, dass $S_{b(x)} = S_{b'(x)} = S$, d.h. $f(x,0) = f(x,1) = S$.

f ist also in der Tat surjektiv. Das bedeutet, dass auch $g : [-1,1] \to \mathcal{P}(\mathbb{N})$ mit $g(x) = f(|x|,0)$ falls $x < 0$ und sonst $g(x) = f(x,1)$ surjektiv ist. Da $[-1,1] \subseteq \mathbb{R}$, folgt somit, dass $|\mathbb{R}| \geq |\mathcal{P}(\mathbb{N})|$.
Als Nächstes zeigen wir, dass $|\mathcal{P}(\mathbb{N})| \geq |\mathbb{R}|$ ist. Um eine surjektive Abbildung von $\mathcal{P}(\mathbb{N})$ nach $\mathbb{R}$ zu erhalten, transformieren wir jede Menge $S \in \mathcal{P}(\mathbb{N})$ zunächst in einen Binärstring $w(S)$ mit der Eigenschaft, dass die i-te Ziffer von $w(S)$ genau dann 1 ist, wenn $i \in S$ ist. Danach transformieren wir den Binärstring w in eine Zahl x_w mit der Eigenschaft, dass w_1 das Vorzeichen von x kodiert (0 ist + und 1 ist −), alle w_i mit geradem i Bit $i/2$ von x links vom Dezimalpunkt und alle w_i mit ungeradem $i > 1$ Bit $(i-1)/2$ von x rechts vom Dezimalpunkt repräsentiert. Zum Beispiel entspricht $w = 10111$ der Zahl $(-10,11)_2 = -2,75$. (Allerdings können wir mit unendlich langen Binärstrings w auch Zahlen erzeugen, die keine

reellen Zahlen sind, da diese potentiell unendlich viele Vorkommastellen enthalten können.) Formal betrachten wir also die Abbildung f mit $f(S) = x_{w(S)}$ für alle $S \in \mathcal{P}(\mathbb{N})$. Da es für jede reelle Zahl x einen (potentiell unendlichen) Binärstring w gibt mit $x_w = x$ und für jeden (potentiell unendlichen) Binärstring w eine Menge $S \in \mathcal{P}(\mathbb{N})$ gibt mit $w(S) = w$, ist f eine surjektive Abbildung auf $\mathbb{R}$. □

Weiterhin gilt der folgende Satz.

Satz 2.26 $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})|$.

Beweis: Wir zeigen diese Aussage durch einen Widerspruchsbeweis. Angenommen, es gäbe eine surjektive Abbildung $f : \mathbb{N} \to \mathcal{P}(\mathbb{N})$. Betrachte eine feste Abbildung f mit dieser Eigenschaft, und sei f repräsentiert durch die linkste Spalte in der folgenden Tabelle (d.h., x wird abgebildet auf $f(x) = S_x$).

Ist die Zahl in der Menge?	natürliche Zahlen				
	1	2	3	4	
S_1	**ja**	nein	ja	nein	...
S_2	nein	**ja**	nein	ja	...
S_3	ja	ja	**nein**	nein	...
			...		

Da jede Zeile i von "ja" und "nein" Antworten die Menge S_i eindeutig bestimmt, können wir jede Zeile von "ja" und "nein" Antworten als eine eindeutige Kodierung von S_i ansehen.
Betrachte nun die Diagonale, die durch die fettgedruckten 'ja" und "nein" Antworten geformt wird. Drehen wir jede "ja" Antwort in eine "nein" Antwort und jede "nein" Antwort in eine "ja" Antwort, erhalten wir eine Antwortkette, die offensichtlich in keiner Zeile enthalten sein kann, da sie sich von jeder Zeile i an der Position i unterscheidet. Das aber bedeutet, dass die Menge, die durch die umgedrehten Antworten kodiert wird, keiner Zahl i zugewiesen wurde. Das widerlegt unsere Behauptung, dass f eine surjektive Abbildung ist, und daher kann es keine surjektive Abbildung von $\mathbb{N}$ nach $\mathcal{P}(\mathbb{N})$ geben. Darum ist $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})|$. □

Die Methode in diesem Beweis nennt man auch *Diagonalisierung*, und wir werden diese später nochmal in einem anderen Kontext behandeln.
Aus den Sätzen 2.25 und 2.26 ergibt sich zusammen mit den vorigen Sätzen das folgende Bild:

$$|\mathbb{N}| = |\mathbb{Z}| = |\mathbb{Q}| < |\mathbb{R}| .$$

Man mag nun annehmen, dass $|\mathbb{N}| < |\bigcup_{i\geq 0} \mathbb{N}^i|$, da $\bigcup_{i\geq 0} \mathbb{N}^i = \mathcal{P}(\mathbb{N})$, aber diese Gleichung ist falsch, da $\bigcup_{i\geq 0} \mathbb{N}^i$ nur Mengen *endlicher* Kardinalität enthält. In der Tat kann das folgende Resultat gezeigt werden.

Satz 2.27 $|\mathbb{N}| = |\bigcup_{i\geq 0} \mathbb{N}^i|$

Beweis: Der Beweis ist eine Übung. □

Weiterhin mag man sich fragen, ob es eine Menge S gibt mit $|\mathbb{N}| < |S| < |\mathbb{R}|$. Das ist in der Tat das erste Problem in David Hilberts berühmter Liste ungelöster Probleme, die er 1900 auf dem Internationalen Mathematiker-Kongress in Paris vorgestellt hat. Die sogenannte *Kontinuumshypothese* behauptet, dass es keine solche Menge S gibt. Gödel hat 1938 gezeigt, dass die Kontinuumshypothese innerhalb der Axiome der Zermelo-Fraenkel (ZF) Mengentheorie nicht

widerlegt werden kann, und 1963 hat Paul Cohen schließlich gezeigt, dass die Kontinuumshypothese auch nicht innerhalb der ZF Axiome bewiesen werden kann. Mit anderen Worten, die Kontinuumshypothese ist *unentscheidbar* innerhalb der ZF Axiomatik.

Satz 2.26 kann wie folgt verallgemeinert werden (was eine Übung ist).

Satz 2.28 *Für jede Menge A gilt, dass $|A| < |\mathcal{P}(A)|$.*

Satz 2.28 impliziert, dass es eine unendliche Folge transfiniter Zahlen gibt. Diese Zahlen werden mit dem hebräischen Buchstaben Aleph (als Zeichen, $\aleph$) bezeichnet. $|\mathbb{N}|$ ist definiert als $\aleph_0$ und $|\mathbb{R}|$ ist definiert als $\aleph_1$. $|\mathbb{N}|$ repräsentiert die kleinste transfinite Zahl, da gezeigt werden kann, dass jede Menge unendlicher Größe mindestens so groß sein muss wie $\mathbb{N}$.

2.7.2 Paradoxien in der Mengentheorie

Cantor publizierte zwischen 1879 und 1884 noch sechs weitere Abhandlungen über die Mengenlehre. Diese Arbeiten erschienen in den *Mathematische Annalen*, und es war zu dieser Zeit eine mutige Entscheidung des Herausgebers, diese Werke zu publizieren, da es eine wachsende Opposition gegenüber Cantors Ideen gab. Eine führende Figur in dieser Opposition war Kronecker, der damals sehr einflussreich in der Mathematikwelt war. Allerdings hatte wohl auch Cantor selbst Probleme mit seiner neuen Theorie. Es wird angenommen, dass er der erste war, der Paradoxien in der Mengenlehre entdeckte, obwohl das erste dokumentierte Paradox von 1897 ist, publiziert von Cesare Burali-Forti. 1899 entdeckte Cantor ein weiteres Paradox, das sich aus der Menge aller Mengen ergibt. Was ist die Kardinalität der Menge aller Mengen? Offensichtlich muss das die größtmögliche Kardinalität sein. Aber wir haben im vorigen Kapitel gezeigt, dass die Potenzmenge einer Menge immer eine größere Kardinalität als die Menge selbst haben muss. Es sah also so aus, als ob die Kritik von Kronecker zumindest teilweise berechtigt war. Das "ultimative" Paradox wurde von Russel 1902 (und unabhängig davon von Zermelo) gefunden. Es betrachtet die Menge

$$A = \{X \mid X \text{ ist kein Element von } X\} .$$

Russel fragte dann: *Ist A ein Element von A?* Sowohl die Annahme, dass A ein Element von A ist, als auch die Annahme, dass A kein Element von A ist, führen zu einem Widerspruch. Die Mengendefinition an sich scheint also ein Paradox zu ergeben. Was läuft hier schief?

Zunächst einmal nehmen wir, wenn wir A definieren, implizit an, dass A existiert! D.h. das Paradox beweist einfach nur durch Widerspruch, dass A nicht existieren kann! Aber wie kann das sein, da es kein Problem mit dem Komplement von A zu geben scheint? Dieses ist nämlich

$$B = \{X \mid X \text{ ist ein Element von } X\} .$$

Ist die Mengentheorie also nicht gegenüber dem Komplement abgeschlossen? Um das Problem zu lösen, spezifizieren wir einen Wertebereich für X, z.B.

$$B = \{X \in \mathbb{N} \mid X \text{ ist ein Element von } X\} .$$

In diesem Fall ist offensichtlich $B = \emptyset$. Also ist

$$A = \{X \in \mathbb{N} \mid X \text{ ist kein Element von } X\} = \mathbb{N} .$$

Wenn wir das Paradox nochmal überprüfen, dann ergibt sich:

- $A \in A \overset{def.}{\Rightarrow} A \notin A$: Dieses Argument gilt nicht mehr, da A kein Element von A ist (bzw. $\mathbb{N} \notin \mathbb{N}$).

- $A \notin A \stackrel{def.}{\Rightarrow} A \in A$: $A \notin A$ ist wahr, aber da $A \notin \mathbb{N}$, kann daraus nicht geschlossen werden, dass $A \in A$ (denn nur die Elemente $X \in \mathbb{N}$ können Elemente von A sein!).

Sobald wir also den Wertebereich von X spezifizieren, löst sich das Paradox in Wohlgefallen auf. Das zeigt auf, dass A nie Teil des Wertebereichs sein kann, aus dem die Xe gewählt sind (ersetzen wir $\mathbb{N}$ durch jeden anderen Wertebereich, erhalten wir dasselbe Resultat wie für $\mathbb{N}$). Wenn also X eine beliebige Menge sein kann, dann kann A nicht existieren!

Einige Mathematiker konnten sich nicht damit anfreunden, dass es möglich ist, Mengen zu definieren, die nicht existieren. Das verursachte auf der einen Seite Versuche, die Mengenlehre so zu axiomatisieren, dass Paradoxe nicht möglich sind, und auf der anderen Seite die Ablehnung von Beweisen, die auf Widersprüchen beruhen, was die mathematische Gruppe der "Konstruktivisten" hervorbrachte. Der Vesuch, eine geeignete Axiomatisierung der Mengentheorie zu finden schlug im Übrigen fehl, und es ist heute bekannt, warum der Versuch scheitern musste. Obwohl Paradoxien seltsam erscheinen mögen, wenn man zum ersten Mal mit ihnen zu tun hat, so sind sie doch nicht auf die Mengentheorie beschränkt. Definiere, z.B., x als die größte natürliche Zahl. Offensichtlich kann x nicht existieren, da $x + 1$ auch eine natürliche Zahl ist. Dementsprechend macht es auch keinen Sinn, die Menge aller Mengen zu definieren, obwohl es hier weniger offensichtlich ist.

2.7.3 Konsequenzen für die Berechenbarkeit

Im Folgenden sei $\mathcal{L}$ die Menge aller Sprachen und $\mathcal{L}_{re}$ die Menge aller rekursiv aufzählbaren Sprachen. Eine fundamentale Frage ist, ob $\mathcal{L}_{re} = \mathcal{L}$ ist oder nicht. Mithilfe unserer Resultate aus der Mengenlehre werden wir zeigen, dass diese Frage leicht zu beantworten ist.

Wir starten mit einer festen Darstellung von $\mathcal{L}$. Offensichtlich kann jedes Entscheidungsproblem als eine Sprache $L \subseteq \{0,1\}^*$ dargestellt werden (L enthält die Binärkodierungen aller Eingaben, die akzeptiert werden sollen). D.h. die Menge aller Entscheidungsprobleme kann durch $\mathcal{P}(\{0,1\}^*)$ beschrieben werden. Weiterhin haben wir gesehen, dass jede DTM M eine eindeutige Gödelnummer $\langle M \rangle$ besitzt. Sei $\mathcal{M} \subseteq \{0,1\}^*$ die Menge aller Worte in $\{0,1\}^*$, die Gödelnummern von DTMs darstellen. Dann ist die Abbildung $f : \mathcal{M} \to \mathcal{L}_{re}$ mit $f(\langle M \rangle) = L(M)$ offensichtlich surjektiv. Also ist $|\mathcal{M}| \geq |\mathcal{L}_{re}|$. Weiterhin ist $|\{0,1\}^*| \geq |\mathcal{M}|$, da $\mathcal{M} \subseteq \{0,1\}^*$ ist. Aufgrund von Satz 2.28 gilt, dass $|\mathcal{P}(\{0,1\}^*)| > |\{0,1\}^*|$. Durch Kombination dieser Ungleichungen erhalten wir $|2^{\{0,1\}^*}| > |\mathcal{L}_{re}|$ und daher, dass $|\mathcal{L}| > |\mathcal{L}_{re}|$. Also kann die Menge der rekursiv aufzählbarne Sprachen nicht alle möglichen Sprachen abdecken. Oder mit anderen Worten:

Satz 2.29 *Es gibt eine Sprache, die nicht rekursiv aufzählbar ist.*

Gibt es auch Sprachen L, für die weder L noch $\bar{L}$ rekursiv aufzählbar sind? Der nächste Satz zeigt, dass das der Fall ist.

Satz 2.30 *Es gibt eine Sprache L, für die weder L noch $\bar{L}$ rekursiv aufzählbar sind.*

Beweis: Um den Satz zu beweisen verwenden wir die Cantorsche Diagonalisierungsmethode. Sei $\mathcal{M} \subseteq \{0,1\}^*$ die Menge aller Gödelnummern von Turingmaschinen und $\mathcal{L}_{rec} = \{L \in 2^{\{0,1\}^*} \mid L \in \mathcal{L}_{re}$ oder $\bar{L} \in \mathcal{L}_{re}\}$. In Worten, $\mathcal{L}_{rec}$ enthält die Menge aller Sprachen L, für die entweder L oder $\bar{L}$ rekursiv aufzählbar sind. Offensichtlich ist die Abbildung $f : \mathcal{M} \times \{0,1\} \to \mathcal{L}_{rec}$ mit $f(\langle M \rangle c) = L(M)$ falls $c = 0$ und $\overline{L(M)}$ sonst surjektiv. Also ist $|\mathcal{M} \times \{0,1\}| \geq |\mathcal{L}_{rec}|$. Weiterhin ist $|\{0,1\}^*| \geq |\mathcal{M} \times \{0,1\}|$, da $\mathcal{M} \times \{0,1\} \subseteq \{0,1\}^*$ ist. Da $|\mathcal{P}(\{0,1\}^*)| > |\{0,1\}^*|$, schließen wir darauf, dass $|\mathcal{P}(\{0,1\}^*)| > |\mathcal{L}_{rec}|$. Mit anderen Worten, es kann keine surjektive

Abbildung von $\mathcal{L}_{rec}$ nach $\mathcal{L}$ geben. Es muss also eine Sprache L geben, für die weder L noch $\bar{L}$ rekursiv aufzählbar sind. ◻

2.7.4 Orakelrechnungen

Natürlich könnte man sich fragen, was passieren würde, wenn ein bestimmtes nicht rekursiv aufzählbares Problem entscheidbar wäre. Wären dann alle Sprachen rekursiv aufzählbar? Wie wir sehen werden, ist das nicht der Fall.

Sei $O \subseteq \{0,1\}^*$ eine beliebige Sprache (was nicht rekursiv aufzählbare Sprachen einschließt). Eine *Turingmaschine mit Orakel O* ist eine DTM mit drei speziellen Zuständen: $q_?$, q_y und q_n. Der Zustand $q_?$ wird verwendet um zu fragen, ob das Wort rechts vom Kopf auf dem Band in O ist. Das Orakel für O wird dann $q_?$ entweder auf q_y oder q_n setzen, abhängig davon, ob das Wort in O ist oder nicht. Die Rechnung der DTM wird danach normal forgesetzt, bis zum nächsten Mal ein Zustand $q_?$ erreicht wird, für den das Orakel die nächste Frage beantwortet.

Falls O entscheidbar ist, kann das Orakel natürlich durch eine andere DTM simuliert werden. In diesem Fall ist die Menge der rekursiv aufzählbaren Sprachen mit Orakel O identisch zur ursprünglichen Menge der rekursiv aufzählbaren Sprachen. Falls aber O nicht entscheidbar ist, könnte es möglich sein, eine Turingmaschine mit Orakel O zu bauen, die eine nicht rekursiv aufzählbare Sprache akzeptiert. Wir bezeichnen eine Turingmaschine M mit Orakel O als M^O. Eine Sprache L heißt *rekursiv aufzählbar bezüglich O* falls $L = L(M^O)$ für eine Turingmaschine M^O gilt. Sei $\mathcal{L}_{re}^O$ die Menge aller Sprachen, die rekursiv aufzählbar bezüglich O sind. Dann gilt der folgende Satz.

Satz 2.31 *Für jedes Orakel O ist $\mathcal{L}_{re}^O \subset \mathcal{L}$.*

Beweis: Offensichtlich ist $\mathcal{L}_{re}^O \subseteq \mathcal{L}$. Es bleibt also zu zeigen, dass es eine Sprache gibt, die nicht in $\mathcal{L}_{re}^O$ ist.

Betrachte ein festes Orakel O. Wir erweitern das Alphabet für die Gödelnummer einer Turingmaschine von $\{0,1\}$ auf $\{0,1,q_?,q_y,q_n\}$. Jede Turingmaschine wird dabei in der Standardweise kodiert, bis auf die Zustände $q_?$, q_y, and q_n, für die neue Symbole benutzt werden. Sei $\mathcal{M}^O \subseteq \{0,1,q_?,q_y,q_n\}^*$ die Menge aller möglichen Kodierungen von Turingmaschinen, die das Orakel O benutzen oder nicht. Offensichtlich ist die Funktion $f : \mathcal{M}^O \to \mathcal{L}_{re}^O$ mit $f(\langle M^O \rangle) = L(M^O)$ surjektiv. Also ist $|\mathcal{M}^O| \geq |\mathcal{L}_{re}^O|$. Da $\mathcal{M}^O \subseteq \{0,1,q_?,q_y,q_n\}^*$, gilt weiterhin, dass $|\{0,1,q_?,q_y,q_n\}^*| \geq |\mathcal{M}^O|$. Weil $\{0,1,q_?,q_y,q_n\}^* \simeq \{000,001,100,101,110\}^* \subseteq \{0,1\}^*$ ist, gilt außerdem, dass $|\{0,1\}^*| \geq |\{0,1,q_?,q_y,q_n\}^*|$. Wegen $|\mathcal{P}(\{0,1\}^*)| > |\{0,1\}^*|$ folgt daraus, dass $|\mathcal{P}(\{0,1\}^*)| > |\mathcal{L}_{re}^O|$, und daher muss es Sprachen geben, die nicht in $\mathcal{L}_{re}^O$ sind. ◻

Satz 2.31 kann auf mehrere Orakel erweitert werden. In der Tat reicht jede endliche und sogar jede abzählbar unendliche Menge von Orakeln nicht aus, um alle Sprachen zu akzeptieren.

2.8 Eine nicht rekursiv aufzählbare Sprache

In diesem Abschnitt werden wir eine erste Sprache kennenlernen, die nicht rekursiv aufzählbar ist. Hierzu benötigen wir noch einige einfache Vorbereitungen. Auf Seite 25 haben wir die Funktion index kennengelernt, die jeder Folge $w \in \{0,1\}^*$ eine natürliche Zahl, oder einen Index, zuordnet. Wie wir gesehen haben, ist die Funktion index bijektiv, so dass wir mit der inversen Funktion jeder natürlichen Zahl $i \in \mathbb{N}$ eine Folge w_i zuordnen können. Nun wollen wir auch Turingmaschinen so aufzählen, dass jeder DTM eine Zahl $i \in \mathbb{N}$ entspricht, und umgekehrt jeder Zahl $i \in \mathbb{N}$ auch eine DTM zugeordnet wird.

Wir sagen, dass *die Zahl* $i \in \mathbb{N}$ *eine Gödelnummer ist*, wenn die Binärdarstellung $bin(i)$ von i die Gödelnummer einer DTM M ist. Schließlich benötigen wir noch die spezielle DTM M^{reject}. Diese ist so definiert, dass M^{reject} jede Eingabe sofort ablehnt. Die von M^{reject} akzeptierte Sprache ist daher die leere Menge. Nun definieren wir für alle $i \in \mathbb{N}$

$$M_i = \begin{cases} M, & \text{falls die Binärdarstellung } bin(i) \text{ die Gödelnummer der DTM } M \text{ ist,} \\ & \text{d.h., } \text{bin}(i) = \langle M \rangle. \\ M^{\text{reject}}, & \text{falls } \text{bin}(i) \text{ keine Gödelnummer ist} \end{cases}$$

Wie angekündigt, entspricht auf diese Weise jeder natürlichen Zahl i eine DTM, und umgekehrt erhält jede DTM M einen Index, nämlich ihre Gödelnummer aufgefasst als natürliche Zahl.

Nun können wir das Verhalten aller DTMs auf allen möglichen Eingaben aus $\{0,1\}^*$ in einer (unendlich großen) Tabelle zusammenfassen. Wir haben diese in Tabelle 2.1 angedeutet. Einträge auf der Diagonalen sind fett markiert. Diese spielen in Kürze eine wichtige Rolle.

	M_1	M_2	M_3	$\cdots \quad \cdots$	M_7	$\cdots \quad \cdots$	M_i	$\cdots \quad \cdots$
w_1	**na**	na	na		na		na	
w_2	na	**na**	na		na		na	
w_3	na	na	**na**		na		na	
$\vdots$								
w_7	na	na	na		**na**		a	
$\vdots$								
w_i	na	na	na		na		**a**	
$\vdots$								

Tabelle 2.1: Tabelle für Aktzeptanz/Nichtakzeptanz von DTMs.

Die Zeilen der Tabelle sind mit den Eingaben aus $\{0,1\}^*$ gelabelt und zwar in der Reihenfolge $w_1, w_2, \ldots$. Die Spalten der Tabelle sind mit den DTMs gelabelt und zwar in der Reihenfolge $M_1, M_2, \ldots$. Jeder Eintrag der Tabelle ist entweder a oder na. Ein Eintrag a in der Zeile w_i und Spalte M_j bedeutet, dass die DTM M_j die Eingabe w_i akzeptiert. Steht in der Zeile w_i und der Spalte M_j der Eintrag na, so bedeutet dies, dass die DTM M_j den Eintrag w_i nicht akzeptiert, d.h., entweder lehnt M_j die Eingabe w_i ab oder die DTM M_j hält bei Eingabe w_i nicht. Die von der DTM M_j akzeptierte Sprache besteht somit aus allen w_i, so dass in der Zeile w_i und der Spalte M_j der Eintrag a steht.

Betrachten wir einige Beispiele. Die DTM M_1 ist M^{reject}, denn 1 ist keine Gödelnummer. M^{reject} lehnt alle Eingaben w_i ab, daher sind alle Einträge in der ersten Spalte der Tabelle na. Da auch 2 und 3 keine Gödelnummern sind, stehen auch in der zweiten und dritten Spalte der Tabelle nur na's. Dasselbe gilt für M_7 und die siebte Spalte.

Schauen wir uns als nächstes die Spalte für M_i mit

$$i = 638779882761580251873009425399934115415079$$

an. Wir haben auf Seite 22 gesehen, dass diese Zahl eine Gödelnummer ist. Genauer, i ist Gödelnummer einer DTM, die alle $w_i \in \{0,1\}^*$ akzeptiert, die mindestens zwei Einsen enthalten. Nun ist $w_1 = \epsilon$, daher lehnt M_i die Eingabe w_1 ab. Somit ist der Eintrag in der Zeile w_1 und Spalte M_i der Tabelle *na*. Es gilt $w_2 = 0$ und $w_3 = 1$ und beide Folgen enthalten weniger als zwei Einsen. Daher sind die Einträge in der Spalte M_i und den Zeilen w_2, w_3 ebenfalls *na*. Dagegen ist $w_7 = 11$ und somit wird w_7 von M_i akzeptiert. Daher steht in der Zeile w_7 und der Spalte M_i der Eintrag *a*.

Entscheidend für unsere erste nicht rekursiv aufzählbare Sprache sind die Einträge auf der Diagonalen der Tabelle, also die Einträge für Zeile w_j und Spalte M_j. Schauen wir uns diesen Eintrag für das oben erwähnte i an. Auf Seite 22 haben wir die Binärdarstellung von i bereits notiert. Die Folge w_i erhalten wir, indem wir die erste 1 in der Binärdarstellung von i streichen. Damit ist

$$\begin{aligned} w_i \quad = \quad & 110101010100110100100100100110100010000010001011 \\ & 01000001010000010011001010010100110010010001001 0011 \\ & 0010001000001000101100100000100000100000100111 \end{aligned}$$

Diese Folge enthält sicherlich zwei Einsen. Daher akzeptiert M_i die Eingabe w_i. Entsprechend ist der Eintrag in Zeile w_i und Spalte M_i unserer Tabelle *a*.

Nun können wir die Sprache Diag definieren, von der wir zeigen werden, dass sie nicht rekursiv aufzählbar ist.

Definition 2.32 Diag $:= \{w \in \{0,1\}^* \mid \text{DTM } M_{\text{index}(w)} \text{ akzeptiert } w \text{ nicht}\}$.

Mit anderen Worten, die Sprache Diag enthält genau die Elemente $w_i \in \{0,1\}^*$, so dass in Zeile w_i und Spalte M_i unserer Tabelle der Eintrag *na* steht. Der Name Diag erklärt sich daraus, dass wir die Werte auf der Diagonalen der Tabelle komplementiert haben, um für jede Folge in $\{0,1\}^*$ zu sagen, ob sie in Diag enthalten ist oder nicht enthalten ist. Wir haben oben gesehen, dass M_1 die Folge $\epsilon = w_1$ ablehnt. Daher ist $\epsilon \in$ Diag. Genauso wird $0 = w_2$ von M_2 abgelehnt und daher ist $0 \in$ Diag. betrachten wir aber unser i von oben und erinnern wir uns, dass M_i die Eingabe w_i akzeptiert, so sehen wir, dass $w_i \notin$ Diag.

Satz 2.33 *Die Sprache Diag ist nicht rekursiv aufzählbar.*

Beweis: Wir zeigen, dass für alle DTMs M gilt: $L(M) \neq$ Diag. Sei nun DTM M beliebig und $j \in \mathbb{N}$ mit $\text{bin}(j) = \langle M \rangle$. Also gilt $M = M_j$. Weiter sei $w \in \{0,1\}^*$ mit $\text{index}(w) = j$. Solch eine Folge w existiert, da index bijektiv ist. Nach Definition von Diag gilt nun

$$\begin{aligned} w \in L(M_j) &\Leftrightarrow M_{\text{index}(w)} \text{ akzeptiert } w \\ &\Leftrightarrow w \notin \text{Diag}. \end{aligned}$$

Damit gilt für alle DTMs M

$$L(M) \neq \text{Diag}$$

und Diag ist nicht rekursiv aufzählbar. □

Wie Sie sicherlich bemerkt haben, entspricht die Technik, die wir zur Konstruktion von Diag angewandt haben, und die Argumentation, die wir im Beweis von Satz 2.33 benutzt haben, dem *Cantorschen Diagonalisierungsverfahren.* Allerdings liefert uns auch Satz 2.33 keine praktisch

interessante Sprache, die nicht rekursiv aufzählbar ist. Wir werden aber sehen, dass wir mit Hilfe der Sprache Diag des Satzes 2.33, einigen allgemeinen Eigenschaften der Begriffe rekursiv aufzählbar und entscheidbar, sowie den sogenannten Reduktionen auch praktisch relevante Sprachen identifizieren können, die nicht rekursiv aufzählbar sind. Dieselben Techniken werden es uns auch ermöglichen, von einigen Sprachen, die rekursiv aufzählbar sind, zu zeigen, dass sie nicht entscheidbar sind. Hierzu gehören u.a. das Halteproblem (siehe Seite 24) und die Sprache Useful (siehe Seite 25).

2.9 Reduktionen

Der Begriff der Reduktion soll formal fassen, was es bedeutet, dass ein Problem A nicht schwerer ist als ein Problem B. Nun ist das Problem A sicherlich nicht schwerer als das Problem B, wenn wir aus einem Algorithmus, der das Problem B löst, einen Algorithmus konstruieren können, der das Problem A löst. Reduktionen sind eine besonders einfache Methode aus einem Algorithmus für das Problem B einen Algorithmus für das Problem A zu erhalten.

Wir wollen dieses an einem einfachen Beispiel demonstrieren. Wir betrachten zum einen das Problem des Multiplizierens zweier natürlicher Zahlen. Für die Eingabe $(a, b) \in \mathbb{Z}^2$ soll also das Ergebnis $a \cdot b$ sein. Das zweite Problem ist das Quadrieren einer ganzen Zahl. Für Eingabe $a \in \mathbb{Z}$ soll also die Ausgabe a^2 sein. Wir sagen, dass das Quadrieren auf das Multiplizieren reduzierbar ist, denn $a^2 = a \cdot a$. Das Quadrieren einer Zahl ist nichts anderes als das Multiplizieren dieser Zahl mit sich selbst. Insbesondere kann jedes Programm zur Multiplikation zweier Zahlen auch zur Berechnung des Quadrats einer Zahl benutzt werden.

Die Art, wie wir in diesem Beispiel aus einem Programm P_M für das Multiplizieren ein Programm P_Q für das Quadrieren erhalten, ist sehr einfach. Um a^2 zu berechnen, rufen wir P_M mit der Eingabe (a, a) auf. Das Programm P_Q für das Quadrieren erhalten wir also aus dem Programm P_M für das Multiplizieren, indem wir vor P_M noch ein Programm ausführen, das bei Eingabe $a \in \mathbb{Z}$ für P_Q eine geeignete Eingabe für P_M (und zwar (a, a)) berechnet. Dann wird P_M mit dieser geeignet gewählten Eingabe aufgerufen.

Bevor wir zu einer formalen Definition von Reduktionen kommen, wollen wir dieses Vorgehen anhand zweier einfacher Sprachen und DTMs statt Programmen noch einmal skizzieren. Wir betrachten zum einen die Sprache P der Palindrome über dem Eingabealphabet $\{0, 1\}$

$$P := \{w \in \{0, 1\}^* \mid w \text{ ist ein Palindrom}\}.$$

Um die zweite Sprache zu definieren, bezeichnen wir für zwei beliebige Bitfolgen $a = (a_1, \ldots, a_n), b = (b_1, \ldots, b_n)$ gleicher Länge mit $a \oplus b$ das bitweise Exklusive-Oder der einzelnen Bits von a und b

$$a \oplus b := (a_1 \oplus b_1, \ldots, a_n \oplus b_n).$$

Dann ist

$$\text{XOR} := \left\{ (a, b, c) \in \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^* \;\middle|\; \begin{array}{l} a, b, c \text{ haben die gleiche} \\ \text{Länge und } a \oplus b = c. \end{array} \right\}.$$

Sei nun M_{XOR} eine beliebige DTM, die die Sprache XOR entscheidet. Weiter sei M_f eine DTM, die die folgende Funktion f berechnet

$$\begin{aligned} f: \quad \{0, 1\}^* &\to \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^* \\ w &\mapsto (w, w^R, 0^{|w|}). \end{aligned}$$

Hierbei bezeichnet w^R die Folge w von rechts nach links geschrieben und $|w|$ ist die Länge von w.

Aus M_f und M_{XOR} erhalten wir nun die folgende DTM M_P, die die Sprache P entscheidet.

Graphisch kann man sich die DTM M_P wie in Abbildung 2.15 vorstellen.

M_P bei Eingabe $w \in \{0,1\}^*$:

1. Berechne mit M_f das Tripel $(w, w^R, 0^{|w|})$.
2. Simuliere M_{XOR} mit Eingabe $f(w)$.
3. Falls M_{XOR} die Eingabe $f(w)$ akzeptiert, akzeptiere w.
4. Falls M_{XOR} die Eingabe $f(w)$ ablehnt, lehne w ab.

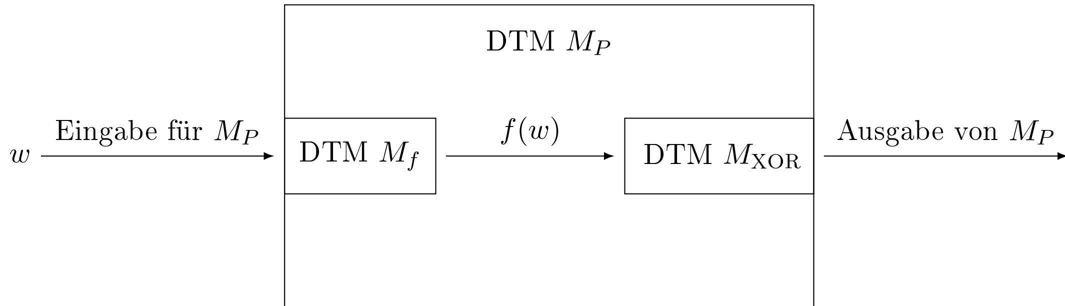


Abbildung 2.15: Graphische Darstellung der DTM M_P.

Es gilt

$$w \in P \Leftrightarrow f(w) = (w, w^R, 0^{|w|}) \in \text{XOR}.$$

Denn liegt w in P, so ist $w = w^R$ und $w \oplus w^R = 0^{|w|}$. Ist andererseits $w \oplus w^R = 0^{|w|}$, so gilt $w = w^R$ und $w \in P$. Daher entscheidet die DTM M_P die Sprache P. Wir sagen, dass wir P *mittels der Funktion f auf XOR reduziert oder zurückgeführt haben.*

Definition 2.34 $L' \subseteq \{0,1\}^*$ heißt reduzierbar auf $L \subseteq \{0,1\}^*$, falls es eine Funktion $f : \{0,1\}^* \to \{0,1\}^*$ gibt mit

1. Für alle $w \in \{0,1\}^*$ gilt:
$$w \in L' \Leftrightarrow f(w) \in L.$$

2. f ist berechenbar, d.h., es gibt eine DTM M_f, die die Funktion f berechnet.

Wir nennen die Funktion f auch eine Reduktion von L' auf L. Ist L' auf L reduzierbar, so schreiben wir $L' \leq L$. Wollen wir die Funktion f noch besonders erwähnen, sagen wir L' ist auf L mittels f reduzierbar.

Damit f eine Reduktion von L' auf L ist, müssen beide Bedingungen der Definition erfüllt sein. Allerdings sind bei unseren Anwendungen die beiden Bedingungen meistens nicht gleich schwer zu erfüllen. Um eine Reduktion einer Sprache L' auf eine Sprache L zu finden, ist der entscheidende Schritte fast immer, eine Funktion f zu finden, die der ersten Bedingung genügt. Haben wir erst einmal eine solche Funktion gefunden, ist diese in aller Regel auch berechenbar.
Wir sehen nun, dass die Funktion f, die wir benutzt haben, um aus einer DTM für die Sprache XOR eine DTM für die Sprache P zu konstruieren, eine Reduktion von P auf die Sprache XOR darstellt.
Graphisch kann die Forderung $w \in L' \Leftrightarrow f(w) \in L$ wie in Abbildung 2.16 dargestellt werden.
Wir sagten, dass Reduktionen in spezieller Weise formalisieren, dass ein Problem oder eine Sprache nicht schwerer ist als ein anderes Problem oder eine andere Sprache. Dass unsere Definition einer Reduktion diese Forderung erfüllt, zeigt das folgende fundamentale Lemma.

Lemma 2.35 $L', L \subseteq \{0,1\}^*$ *seien Sprachen mit* $L' \leq L$ *mittels* f.

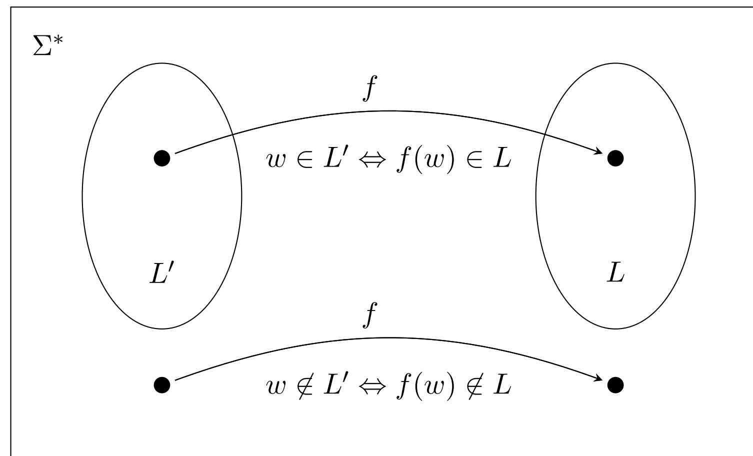


Abbildung 2.16: Graphische Darstellung einer Reduktion f.

1. *Ist L entscheidbar, so ist auch L' entscheidbar.*
2. *Ist L rekursiv aufzählbar, so ist auch L' rekursiv aufzählbar.*

Beweis: Wir beweisen nur die erste Aussage des Lemmas. Die zweite folgt sehr ähnlich. Sei M eine DTM, die L entscheidet. Weiter sei M_f eine DTM, die f berechnet. Ein solche DTM muss es geben, denn f ist berechenbar. Die DTM M' arbeite nun wie folgt:

M' bei Eingabe $w \in \{0,1\}^*$:

1. Simuliere M_f mit Eingabe w und berechne so $f(w)$.
2. Simuliere M mit Eingabe $f(w)$.
3. Falls M die Eingabe $f(w)$ akzeptiert, akzeptiere w.
4. Falls M die Eingabe $f(w)$ ablehnt, lehne w ab.

Ersetzen wir in Abbildung 2.15 M_{XOR} durch M und M_P durch M', so erhalten wir eine graphische Darstellung der Arbeitsweise von M'.
Wir zeigen nun, dass M' die Sprache L' entscheidet. Hierzu müssen wir zeigen

- M' hält bei jeder Eingabe.
- Ist $w \in L'$, so akzeptiert M' die Eingabe w.
- Ist $w \notin L'$, so lehnt M' die Eingabe w ab.

Nun hält M' bei jeder Eingabe, denn M_f und M halten bei jeder Eingabe. Da M die Sprache L entscheidet, gilt außerdem

$$\begin{array}{lcl} z \in L & \Rightarrow & M \text{ akzeptiert } z \in \{0,1\}^*. \\ z \notin L & \Rightarrow & M \text{ lehnt } z \in \{0,1\}^* \text{ ab.} \end{array}$$

Mit dieser Beobachtung, mit $w \in L' \Leftrightarrow f(w) \in L$ und mit der Definition von M' erhalten wir

$$\begin{aligned} w \in L' &\Rightarrow f(w) \in L \\ &\Rightarrow M \text{ akzeptiert } f(w) \in \{0,1\}^* \\ &\Rightarrow M' \text{ akzeptiert } w \in \{0,1\}^* \end{aligned}$$

$$\begin{aligned} w \notin L' &\Rightarrow f(w) \notin L \\ &\Rightarrow M \text{ lehnt } f(w) \in \{0,1\}^* \text{ ab} \\ &\Rightarrow M' \text{ lehnt } w \in \{0,1\}^* \text{ ab.} \end{aligned}$$

□

Bevor wir Reduktionen benutzen, um zu zeigen, dass Sprachen unentscheidbar sind, wollen wir noch zwei Beispiele von Reduktionen kennenlernen, die wir später bei Unentscheidbarkeitsbeweisen benutzen.
Die Sprache

$$A = \{\langle M \rangle x \mid M \text{ ist DTM, die die Eingabe } x \text{ akzeptiert}\}$$

wird das *Akzeptanzproblem* genannt. Im Gegensatz zum Halteproblem muss die DTM M die Eingabe x akzeptieren und nicht nur bei Eingabe x halten, damit $\langle M \rangle x \in A$. Zur Erinnerung hier noch einmal die Definition des Halteproblems (Seite 24)

$$H := \{\langle M \rangle x \mid M \text{ ist DTM, die gestartet mit Eingabe } x \text{ hält.}\}.$$

Wir wollen zeigen, dass sich H auf A reduzieren lässt. Wir müssen also eine berechenbare Funktion f konstruieren, so dass für alle $w \in \{0,1\}^*$ gilt $w \in H \Leftrightarrow f(w) \in A$. Um die Funktion f zu konstruieren, unterscheiden wir zwei Typen von Worten in $\{0,1\}^*$: Worte w, die die Form $w = \langle M \rangle x$ für eine DTM M haben, und Worte w, die sich nicht auf diese Weise schreiben lassen.
Sei zunächst $w = \langle M \rangle x$ für eine DTM M. Zu M konstruieren wir eine DTM $\bar{M}$, für die gilt

$$\langle M \rangle x \in H \Leftrightarrow \langle \bar{M} \rangle x \in A.$$

Wir werden dann $f(w) = f(\langle M \rangle x) = \langle \bar{M} \rangle x$ setzen. Hier ist die Beschreibung der DTM $\bar{M}$.

$\bar{M}$ bei Eingabe $z \in \{0,1\}^*$:

1. Simuliere M mit Eingabe z
2. Falls M Eingabe z akzeptiert, akzeptiere z.
3. Falls M Eingabe z ablehnt, akzeptiere z.

Wie gewünscht, gilt für jede DTM M und die entsprechende DTM $\bar{M}$ die Äquivalenz $\langle M \rangle x \in H \Leftrightarrow \langle \bar{M} \rangle x \in A$. Dieses beweisen wir, indem wir

i) $\langle M \rangle x \in H \Rightarrow \langle \bar{M} \rangle x \in A$.

ii) $\langle \bar{M} x \rangle \in A \Rightarrow \langle M \rangle x \in H$.

beweisen.

zu i): Ist $\langle M \rangle x \in H$, so hält M bei Eingabe x. Nach Definition von $\bar{M}$ wird dann die DTM $\bar{M}$ die Eingabe x akzeptieren. Also gilt $\langle \bar{M} \rangle x \in A$.

zu ii): Ist $\langle \bar{M} \rangle x \in A$, so akzeptiert die DTM $\bar{M}$ die Eingabe x. Dann aber muss M bei Eingabe x halten. Also gilt $\langle M \rangle x \in H$.

Nun betrachten wir die Worte w, die sich nicht als $\langle M \rangle x$ schreiben lassen. All diese Worte liegen nicht in H. Daher darf auch $f(w)$ nicht in A liegen. Da aber in A kein Wort liegt, das sich nicht als $\langle M \rangle x$ für eine DTM M schreiben lässt, können wir $f(w) = w$ wählen.
Insgesamt ist nun die Funktion f, durch die H auf A reduziert werden soll, folgendermaßen definiert.

$$f(w) = \begin{cases} w & \text{falls } w \text{ nicht von der Form } \langle M \rangle x \text{ für eine DTM } M \text{ ist.} \\ \\ \langle \bar{M} \rangle x & \text{falls } w = \langle M \rangle x \text{ für eine DTM } M, \\ & \text{wobei } \bar{M} \text{ sich wie oben beschrieben aus } M \text{ ergibt.} \end{cases}$$

Um $H \leq A$ mittels f zu beweisen, müssen wir zeigen

(i) $w \in H \Leftrightarrow f(w) \in A$.

(ii) f, wie oben definiert, ist berechenbar.

Obwohl wir (i) eigentlich schon bewiesen haben, fassen wir die Argumentation noch einmal zusammen. Wir müssen $w \in H \Rightarrow f(w) \in A$ und $w \notin H \Rightarrow f(w) \notin A$ zeigen. Es gilt:

$$\begin{aligned} w \in H \quad & \Rightarrow \quad w = \langle M \rangle x, \text{ wobei die DTM } M \text{ bei Eingabe } x \text{ hält.} \\ & \overset{a)}{\Rightarrow} \quad f(w) = \langle \bar{M} \rangle x, \text{ wobei } \bar{M} \text{ die Eingabe } x \text{ akzeptiert} \\ & \Rightarrow \quad f(w) = \langle \bar{M} \rangle x \in A. \end{aligned}$$

$$\begin{aligned} w \notin H \quad & \Rightarrow \quad w = \langle M \rangle x, \text{ wobei die DTM } M \text{ nicht bei Eingabe } x \text{ hält oder} \\ & \qquad w \text{ nicht von der Form } \langle M \rangle x \text{ für eine DTM } M \text{ ist.} \\ & \overset{b)}{\Rightarrow} \quad f(w) = \langle \bar{M} \rangle x, \text{ wobei } \bar{M} \text{ die Eingabe } x \text{ nicht akzeptiert oder} \\ & \qquad f(w) = w, \text{ wobei } w \text{ nicht von der Form } \langle M \rangle x \text{ für eine DTM } M \text{ ist} \\ & \Rightarrow \quad f(w) \notin A. \end{aligned}$$

Hier haben wir bei *a)* $\langle M \rangle x \in H \Rightarrow \langle \bar{M} \rangle x \in A$ benutzt. Bei *b)* wird $\langle M \rangle x \notin H \Rightarrow \langle \bar{M} \rangle x \notin A$ gebraucht.
Zu (ii) betrachten wir die folgende grobe Skizze einer DTM M_f.

M_f bei Eingabe $w \in \{0,1\}^*$:

1. Überprüfe, ob $w = \langle M \rangle x$ für eine DTM M und $x \in \{0,1\}^*$.
2. Falls dies nicht der Fall ist, gib w aus.
3. Falls $w = \langle M \rangle x$, konstruiere aus M die DTM $\bar{M}$, indem q_{reject} durch q_{accept} ersetzt wird.
4. Berechne $\langle \bar{M} \rangle$ und gib $\langle \bar{M} \rangle x$ aus.

Im 1. Schritt suchen wir in w zunächst die zweite Folge von 111 in w. Existiert eine solche nicht, ist die Ausgabe von M_f die Eingabe w selber. Dann überprüfen wir, ob w mit 111 beginnt und die Folge zwischen den beiden Folgen 111 eine Gödelnummer ist. Ist eine dieser Eigenschaften verletzt, ist die Ausgabe wieder w. Sonst gehen wir zum 3. Schritt. Zur Überprüfung ob zwischen den beiden 111-Folgen eine Gödelnummer steht, können wir die DTM für die Sprache Gödel (siehe Seite 23) benutzen.

Auch zu den Schritten 3 und 4 muss noch einiges gesagt werden. Wir können die DTM $\bar{M}$ aus M erhalten, indem wir in der Übergangsfunktion für M alle Übergänge in den Zustand q_{reject} durch Übergänge in den Zustand q_{accept} ersetzen. Sei die Anzahl der Zustände von M genau $n+1$. Dann ist $q_{\text{accept}} = q_{n-1}$ und $q_{\text{reject}} = q_n$. Für die Gödelnummern von $M, \bar{M}$ heißt die Änderung von q_{accept} in q_{reject}, dass wir $\langle \bar{M} \rangle$ aus $\langle M \rangle$ erhalten, indem alle Codierung 0^{n+1} von q_{reject} in Codierungen 0^n von q_{accept} ändern. Dieses kann sicherlich auf einer DTM durchgeführt werden. Damit ist gezeigt, dass alle Schritte der oben skizzierten DTM M_f auch wirklich auf einer DTM durchgeführt werden können. Wir schließen, dass f berechenbar ist. Insgesamt haben wir gezeigt

Lemma 2.36 *Das Halteproblem lässt sich auf das Akzeptanzproblem reduzieren:*

$$H \leq A.$$

Die Art und Weise, wie wir dieses Lemma bewiesen haben, ist typisch für Reduktionsbeweise. Wir haben nicht unmittelbar die Funktion f angegeben, die die Reduktion ausführen soll. Vielmehr haben wir uns zunächst überlegt, wie wir aus einer DTM M eine DTM $\bar{M}$ konstruieren können, so dass bei jeder Eingabe $x \in \{0,1\}^*$ die DTM M bei Eingabe x genau dann hält, wenn $\bar{M}$ die Eingabe x akzeptiert. Erst dann, und mit Hilfe der Konstruktion von $\bar{M}$ haben wir die Funktion f definiert. In der Definition von f haben wir dann auch die Fälle betrachtet, wo das Wort w nicht von der Form $\langle M \rangle x$ ist. Dieses Testen, ob ein Wort das richtige Format hat, ist bei fast allen Reduktionen notwendig und auch bei fast allen Reduktionen einfach umzusetzen. Genauso einfach ist es bei den meisten Reduktionen zu zeigen, dass die Funktion f berechenbar ist. Wir gehen daher in den meisten Beweisen nur kurz auf die Berechenbarkeit von f ein.
Als nächstes konstruieren wir eine Reduktion des Akzeptanzproblems A auf die Sprache Useful (siehe Seite 25). Um die Reduktion zu konstruieren, konzentrieren wir uns zunächst auf Worte $w \in \{0,1\}^*$, die aus einer Gödelnummer $\langle M \rangle$ und einer Folge $x \in \{0,1\}^*$ bestehen. Wir müssen uns überlegen, wie wir aus $\langle M \rangle x$ ein Paar $(\langle M_x \rangle, q)$ bestehend aus einer DTM M_x und einem Zustand dieser DTM konstruieren können, so dass M die Eingabe x genau dann akzeptiert, wenn es eine Eingabe z für M_x gibt, bei der M_x in den Zustand q geht. Wir werden nun die naheliegende Wahl $q = q_{\text{accept}}$ treffen. Die DTM M_x werden wir so konstruieren, dass sie bei jeder Eingabe $z \in \{0,1\}^*$ das Verhalten von M bei Eingabe x simuliert. Etwas genauer wird M_x folgendermaßen konstruiert

M_x bei Eingabe $z \in \{0,1\}^*$:

1. Lösche z vom Band und schreibe x auf das Band.
2. Simuliere M mit Eingabe x.

Nun gilt

$$\langle M \rangle x \in A \Leftrightarrow (\langle M_x \rangle, q_{\text{accept}}) \in \text{Useful}.$$

Hierzu müssen

i) $\langle M \rangle x \in A \Rightarrow (\langle M_x \rangle, q_{\text{accept}}) \in \text{Useful}$

ii) $(\langle M_x \rangle, q_{\text{accept}}) \in \text{Useful} \Rightarrow \langle M \rangle x \in A$

gezeigt werden.

zu i): Akzeptiert M das Wort x, so wird M_x alle $z \in \{0,1\}^*$ akzeptieren. Damit gibt es eine Eingabe für M_x, bei der M_x den Zustand q_{accept} erreicht und $(\langle M_x \rangle, q_{\text{accept}}) \in$ Useful.

zu ii): Erreicht M_x bei irgendeiner Eingabe den Zustand q_{accept}, so muss M bei Eingabe x den Zustand q_{accept} erreichen, denn M_x verhält sich bei beliebiger Eingabe wie M bei Eingabe x. Damit liegt dann $\langle M \rangle x$ in der Sprache A.

Jetzt können wir die Reduktion f des Akzeptanzproblems auf die Sprache Useful definieren. Es gilt

$$f(w) = \begin{cases} (\langle M^{\text{reject}} \rangle, q_{\text{accept}}), & \text{falls } w \text{ nicht von der Form } \langle M \rangle x \text{ ist} \\ (\langle M_x \rangle, q_{\text{accept}}), & \text{falls } w \text{ von der Form } \langle M \rangle x \text{ für eine DTM } M \text{ ist.} \end{cases}$$

Hierbei ist M^{reject} die DTM, die bei jeder Eingabe in den ablehnenden Zustand q_{reject} geht. Aus dieser Setzung und der Äquivalenz $\langle M \rangle x \in A \Leftrightarrow (\langle M_x \rangle, q_{\text{accept}}) \in$ Useful, die wir oben schon bewiesen haben, folgt nun für alle $w \in \{0,1\}^*$

$$w \in A \Leftrightarrow f(w) \in \text{Useful}.$$

Damit f eine Reduktion von A auf Useful ist, müssen wir abschließend noch zeigen, dass die Funktion f berechenbar ist. Wir wissen aber bereits, dass wir mit einer DTM entscheiden können, ob eine Bitfolge eine Gödelnummer ist. Weiter kann für eine DTM M und eine Folge $x \in \{0,1\}^*$ die DTM M_x und die Gödelnummer $\langle M_x \rangle$ wie folgt berechnet werden. Zu den Zuständen von M fügen wir noch Zustände hinzu, die bei beliebiger Eingabe z diese vom Band löschen und den Lesekopf anschließend an den Beginn des Bandes zurückführen. Dann gibt es Zustände, die x auf das Band schreiben. Ist $x = x_1 \ldots x_n$ so gibt es für jedes Symbol x_i einen Zustand $s_i, i = 1, \ldots, n$. Ist der aktuelle Zustand $s_i, i \leq n-1$, so wird x_i auf das Band geschrieben und der Lesekopf nach rechts bewegt. Dann wird in den Zustand s_{i+1} gegangen. Bei s_n schreiben wir x_n aufs Band, gehen an den Beginn des Bandes zurück und wechseln schließlich in den Startzustand der Maschine M. Die so konstruierte Maschine ist M_x. Diese DTM M_x kann also recht einfach konstruiert werden. Insbesondere kann auch die Gödelnummer $\langle M_x \rangle$ von M_x auf einer DTM berechnet werden. Damit ist f berechenbar und wir haben gezeigt

Lemma 2.37 *Das Akzeptanzproblem kann auf die Sprache Useful reduziert werden.*

2.10 Nichtentscheidbarkeit und Reduktionen

Wir hatten gesagt, dass Reduktionen uns helfen werden, von Sprachen zu zeigen, dass sie unentscheidbar sind. Dieses beruht auf dem folgenden Korollar zu Lemma 2.35

Korollar 2.38 $L', L \subseteq \{0,1\}^*$ *seien Sprachen mit* $L' \leq L$ *mittels* f.

1. *Ist* L' *unentscheidbar, so ist auch* L *unentscheidbar.*
2. *Ist* L' *nicht rekursiv aufzählbar, so ist auch* L *nicht rekursiv aufzählbar.*

Beweis: Die beiden Aussagen des Korollars sind jeweils die Kontraposition der entsprechenden Aussage von Lemma 2.35. □

Man beachte die Rolle der Sprachen in den Aussagen dieses Korollars. Wollen wir von einer Sprache L mit Hilfe dieses Korollars zeigen, dass sie nicht entscheidbar ist, so benötigen wir eine Sprache L', von der wir bereits wissen, dass sie unentscheidbar ist. *Dann muss die Sprache* L' *auf* L *reduziert werden*, um schließen zu können, dass L nicht entscheidbar ist. Als erstes zeigen wir

Satz 2.39 *Das Halteproblem H ist nicht entscheidbar.*

Beweis: Wäre H entscheidbar, so wäre nach Satz 2.12 das Komplement $\bar{H}$ von H rekursiv aufzählbar. Wir konstruieren nun eine Reduktion der Sprache Diag auf $\bar{H}$. Dann ist nach Korollar 2.38 die Sprache $\bar{H}$ nicht rekursiv aufzählbar, und es folgt, dass H nicht entscheidbar ist. Hier ist zunächst eine ausführlichere Beschreibung der Sprache $\bar{H}$

$$\bar{H} := \left\{ w \in \{0,1\}^* \;\middle|\; \begin{array}{l} w \text{ ist nicht von der Form } \langle M \rangle x \text{ für eine DTM } M \text{ oder} \\ w = \langle M \rangle x \text{ für DTM } M \text{, die bei Eingabe } x \text{ nicht hält} \end{array} \right\}.$$

Für jede DTM M bezeichne M^∞ die DTM, die in eine Endlosschleife geht, wenn M in den ablehnenden Zustand q_{reject} geht. In allen anderen Situationen verhält sich M^∞ wie M. Die Endlosschleife von M^∞ können wir etwa realisieren, indem M^∞ seinen Lesekopf immer weiter nach rechts über sein Eingabeband bewegt. M^∞ akzeptiert dieselben Eingaben wie M. Bei allen Eingaben jedoch, die M nicht akzeptiert, hält die DTM M^∞ nicht. Für jede DTM M gilt damit $L(M) = L(M^\infty)$.

Um die Reduktion von Diag auf $\bar{H}$ zu definieren, benutzen wir die Funktion index die jeder Folge $w \in \{0,1\}^*$ einen Index $\text{index}(w)$ zuordnet (siehe Seite 25). Dann ist die Reduktion f definiert durch

$$\begin{aligned} f : \{0,1\}^* &\to \{0,1\}^* \\ w &\mapsto \langle M^\infty_{\text{index}(w)} \rangle w. \end{aligned}$$

Es muss gezeigt werden

$$w \in \text{Diag} \Leftrightarrow f(w) = \langle M^\infty_{\text{index}(w)} \rangle w \in \bar{H}.$$

Sei also $w \in \{0,1\}^*$ beliebig. Wir zeigen zunächst $w \in \text{Diag} \Rightarrow f(w) = \langle M^\infty_{\text{index}(w)} \rangle w \in \bar{H}$. Ist $w \in \text{Diag}$, so akzeptiert nach Definition der Sprache Diag die DTM $M_{\text{index}(w)}$ die Eingabe w nicht. Nach Definition von $M^\infty_{\text{index}(w)}$ geht diese DTM dann bei Eingabe w in eine Endlosschleife. $M^\infty_{\text{index}(w)}$ hält also bei Eingabe w nicht und daher gilt $\langle M^\infty_{\text{index}(w)} \rangle w \in \bar{H}$.

Jetzt zeigen wir $f(w) = \langle M^\infty_{\text{index}(w)} \rangle w \in \bar{H} \Rightarrow w \in \text{Diag}$. Gilt $\langle M^\infty_{\text{index}(w)} \rangle w \in \bar{H}$, so wird $M^\infty_{\text{index}(w)}$ bei Eingabe w nicht halten. Dann aber akzeptiert $M_{\text{index}(w)}$ die Eingabe w nicht. Denn wir hatten oben argumentiert, dass für eine beliebige DTM M, die beiden DTMs M und M^∞ dieselben Eingaben akzeptieren. Akzeptiert $M_{\text{index}(w)}$ die Eingabe w nicht, so gilt nach Definition der Sprache Diag wie zu zeigen ist, dass $w \in \text{Diag}$.

Schließlich muss noch gezeigt werden, dass die Funktion f berechenbar ist. Betrachten wir nun die folgende DTM M_f.

M_f bei Eingabe $w \in \{0,1\}^*$:
1. Berechne $\langle M_{\text{index}(w)} \rangle$.
2. Berechne $\langle M^\infty_{\text{index}(w)} \rangle$ und gebe $\langle M^\infty_{\text{index}(w)} \rangle w$ aus.

Der Index $\text{index}(w)$, und damit die Gödelnummer von $\langle M_{\text{index}(w)} \rangle$, im ersten Schritt kann leicht berechnet werden, da $w) = (1w)_2$ (siehe Seite 25). Die Modifikationen an $M_{\text{index}(w)}$ und $\langle M_{\text{index}(w)} \rangle$ im zweiten Schritt, um $M^\infty_{\text{index}(w)}$ und $\langle M^\infty_{\text{index}(w)} \rangle$ zu berechnen, können dann ebenfalls leicht berechnet werden. □

Der Beweis liefert uns

Korollar 2.40 *Die Sprache $\bar{H}$ ist nicht rekursiv aufzählbar.*

Ausserdem erhalten wir

Korollar 2.41 *(i) Die Klasse der rekursiv aufzählbaren Sprachen ist von der Klasse der entscheidbaren Sprachen verschieden.*

(ii) Die Klasse der rekursiv aufzählbaren Sprachen ist nicht gegen Komplementbildung abgeschlossen.

Beweis:

(i) Nach Satz 2.39 ist das Halteproblem H nicht entscheidbar. Nach Satz 2.18 ist H jedoch rekursiv aufzählbar. H liegt also in der Klasse der rekursiv aufzählbaren Sprachen, nicht jedoch in der Klasse der entscheidbaren Sprachen.

(ii) Nach Satz 2.18 ist H rekursiv aufzählbar. Nach Korollar 2.40 ist jedoch das Komplement von H nicht rekursiv aufzählbar.

□

Wir können unsere Reduktionen des letzten Abschnitts benutzen, um weitere Unentscheidbarkeitsresultate zu erhalten.

Satz 2.42 *Das Akzeptanzproblem A und die Sprache Useful sind nicht entscheidbar.*

Beweis: Aus Korollar 2.38, Satz 2.39 und Lemma 2.36 folgt, dass das Akzeptanzproblem nicht entscheidbar ist. Dann folgt aus Korollar 2.38 und Lemma 2.37, dass auch die Sprache Useful nicht entscheidbar ist. □

Als letzte Anwendung von Reduktionen wollen wir zeigen, dass auch das sogenannte *Halteproblem mit leerem Band* H_0 nicht entscheidbar ist. Hier zunächst die Definition dieser Sprache.

$$H_0 = \{\langle M\rangle \mid M \text{ ist DTM, die gestartet mit dem leeren Wort } \epsilon \text{ als Eingabe hält}\}$$

Satz 2.43 *Das Halteproblem mit leerem Band H_0 ist nicht entscheidbar.*

Beweis: Wir zeigen, dass H auf H_0 reduziert werden kann. Dann folgt aus Satz 2.39 und Korollar 2.38 die Behauptung des Satzes.
Für die Reduktion von H auf H_0 benötigen wir die folgende Konstruktion einer DTM M_x aus einer DTM M und einer Folge x (siehe auch Seite 42)

M_x bei Eingabe $z \in \{0,1\}^*$:

1. Lösche z vom Band und schreibe x auf das Band.
2. Simuliere M mit Eingabe x.

Für alle Eingaben $z \in \{0,1\}^*$ verhält sich M_x also wie M bei Eingabe x. Insbesondere verhält sich M_x bei Eingabe des leeren Wortes ϵ wie M bei Eingabe x. Also

$$M \text{ hält bei Eingabe } x \Leftrightarrow M_x \text{ hält bei Eingabe } \epsilon \tag{2.1}$$

Nun definieren wir f durch

$$f(w) = \begin{cases} w & \text{falls } w \text{ nicht von der Form } \langle M\rangle x \text{ für eine DTM } M \text{ ist.} \\ \langle M_x\rangle & \text{falls } w = \langle M\rangle x \text{ für eine DTM } M \text{ ist} \end{cases}$$

Hier ist $\langle M_x \rangle$ die Gödelnummer der DTM M_x. Aus (2.1) folgt

$$w \in H \Leftrightarrow f(w) \in H_0.$$

Wir müssen uns noch überlegen, dass f berechenbar ist. Dies folgt aber aus der Tatsache, dass M_x aus M und x leicht konstruierbar ist. □

Dieser Satz zeigt, dass das Halteproblem selbst dann noch unentscheidbar bleibt, wenn wir immer nur für eine feste Eingabe wissen wollen, ob eine DTM M bei dieser Eingabe hält. Für den Satz hatten wir die feste Eingabe ϵ gewählt. Wir hätten aber auch jede andere beliebige, aber feste Eingabe wählen können.

2.11 Weitere unentscheidbare Sprachen

Es gibt noch viele andere interessante unentscheidbare Sprachen. Hierzu gehören:

$\{\langle M \rangle \mid M$ hält für jede Eingabe$\}$ *Totalitätsproblem*

$\{\langle M \rangle \mid M$ hält für endlich viele Eingaben$\} \rightarrow$ *Endlichkeitsproblem*

$\{(\langle M \rangle, \langle M' \rangle) \mid M$ und M' *akzeptieren die gleiche Sprache*$\} \rightarrow$ *Äquivalenzproblem*

Wir betrachten hier nur das Totalitätsproblem und zeigen, dass es nicht rekursiv aufzählbar ist. Der Beweis des folgenden Satzes ist etwas komplexer und trickreicher, als die Beweise, die wir bislang gesehen haben.

Satz 2.44 *Das Totalitätsproblem ist nicht rekursiv aufzählbar.*

Beweis: Wir zeigen, dass das Komplement des Halteproblems auf das Totalitätsproblem reduziert werden kann. Dann folgt der Satz aus Korollar 2.40 und Korollar 2.38. Im Weiteren bezeichnen wir das Totalitätsproblem mit Total.
Für die Reduktionen benötigen wir die folgende Konstruktionen einer DTM $M^{(x)}$ aus einer DTM M und einer Folge $x \in \{0,1\}^*$.

$M^{(x)}$ bei Eingabe $z \in \{0,1\}^*$:

1. Berechne $|z|$.
2. Simuliere M mit Eingabe x für $|z|$ Schritte.
3. Falls M während der $|z|$ Schritte hält, gehe in eine Endlosschleife.
4. Sonst akzeptiere z.

Nun gilt für jede DTM M und Folge $x \in \{0,1\}^*$

$$M \text{ hält bei Eingabe } x \text{ nicht} \Leftrightarrow M^{(x)} \text{ hält bei jeder Eingabe } z \in \{0,1\}^* \qquad (2.2)$$

Nun definieren wir die Funktion f durch

$$f(w) := \begin{cases} \langle M^{\text{reject}} \rangle & \text{falls } w \text{ nicht von der Form } \langle M \rangle x \text{ für eine} \\ & \text{DTM } M \text{ ist,} \\ \langle M^{(x)} \rangle & \text{falls } w = \langle M \rangle x \text{ für eine DTM } M \text{ ist} \end{cases}$$

Hierbei ist M^{reject} die DTM, die bei jeder Eingabe sofort in den Zustand q_{reject} geht, insbesondere als bei jeder Eingabe hält. Damit und aus (2.2) gilt für alle $w \in \{0,1\}^*$

$$w \in \bar{H} \Leftrightarrow f(w) \in \text{Total}.$$

Da f auch berechenbar ist, ist f die gewünschte Reduktion von $\bar{H}$ auf Total. □

Die Unentscheidbarkeit des Endlichkeitsproblems und des Äquivalenzproblems sind ein direktes Korollar des folgenden Satzes von Rice. Gleiches gilt für viele weitere Probleme, die Aussagen über das Verhalten von Turingmaschinen oder über Eigenschaften der von ihnen berechneten Funktion machen,

Satz 2.45 (Satz von Rice) *Sei $\mathcal{R}$ die Menge aller (partiellen) berechenbaren Funktionen (d. h. Funktionen der Form $f : U \to \{0,1\}^*$ für eine Teilmenge $U \subseteq \{0,1\}^*$, für die es eine DTM gibt, die f berechnet) und $S \subseteq \mathcal{R}$. Dann ist die Sprache*

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

nicht entscheidbar genau dann wenn S eine nichttriviale Teilmenge von $\mathcal{R}$ ist, d. h. $S \neq \mathcal{R}$ und $S \neq \emptyset$.

Beweis: Wir zeigen zunächst, dass wenn $S = \mathcal{R}$ oder $S \neq \emptyset$ ist, $L(S)$ entscheidbar ist. Angenommen, $S = \mathcal{R}$. Dann ist $L(S)$ die Menge aller Gödelnummern von Turingmaschinen und damit gleich der Sprache Gödel, die wir in Kapitel 2.6.2 kennengelernt haben. Also ist wegen Lemma 2.16 $L(S)$ entscheidbar. Falls $S = \emptyset$, dann ist $L(S)$ leer und damit offensichtlich auch entscheidbar, da hier eine DTM ausreicht, die alle Eingaben ablehnt.
Als Nächstes zeigen wir, dass wenn S eine nichttriviale Teilmenge von $\mathcal{R}$ ist, dann $L(S)$ nicht entscheidbar ist. Sei f_{nd} die überall nicht definierte Funktion (d. h. U im Satz von Rice ist leer). Zunächst nehmen wir an, dass $f_{nd} \in S$. Wir zeigen dann, dass $H_0 \leq \overline{L(S)}$.
Sei g eine beliebige Funktion in $\mathcal{R} \setminus S$. Sei $\tilde{M}$ eine DTM, die g berechnet, und M_{nd} eine DTM, die f_{nd} berechnet. Wir betrachten nun die folgende DTM $N(M)$.

$N(M)$ bei Eingabe $y \in \{0,1\}^*$:

1. Simuliere M gestartet mit ϵ.
2. Falls M hält, simuliere $\tilde{M}$ gestartet mit y

Weiterhin definieren wir die Funktion

$$f(x) := \begin{cases} \langle N(M) \rangle & \text{falls } x = \langle M \rangle \text{ für eine DTM } M \\ \langle M_{nd} \rangle & \text{sonst} \end{cases}$$

Offensichtlich ist f berechenbar. Wir zeigen nun: $x \in H_0 \;\Leftrightarrow\; f(x) \in \overline{L(S)}$.
$\Rightarrow$: Falls $x \in H_0$, dann ist $x = \langle M \rangle$ für eine DTM M und M hält bei Eingabe ϵ. In diesem Fall ist $f(x) = \langle N(M) \rangle$ für eine DTM $N(M)$, die g berechnet. Da $g \notin S$, ist damit $\langle N(M) \rangle \notin L(S)$.
$\Leftarrow$: Falls $x \notin H_0$, unterscheiden wir zwischen zwei Fällen. Ist x nicht von der Form $x = \langle M \rangle$ für eine DTM M, dann ist $f(x) = \langle M_{nd} \rangle$. Da $f_{nd} \in S$, ist damit $f(x) \in L(S)$. Ist $x = \langle M \rangle$ für eine DTM M, die bei Eingabe ϵ nicht hält, dann ist $f(x) = \langle N(M) \rangle$ für eine DTM $N(M)$, die für keine Eingabe hält und damit die Funktion f_{nd} berechnet. Daraus folgt, dass $f(x) \in L(S)$.
Wir haben also gezeigt, dass $H_0 \leq \overline{L(S)}$. Da H_0 nicht entscheidbar ist, ist damit auch $\overline{L(S)}$ nicht entscheidbar und daher $L(S)$ nicht entscheidbar.

Schließlich müssen wir noch den Fall betrachten, dass $f_{nd} \notin S$. Hier können wir zeigen, dass $H_0 \leq L(S)$. Der Beweis ist eine Übung. □

Beispiel 1: Wenn wir als S die Menge aller totalen berechenbaren Funktionen wählen, liefert uns der Satz von Rice die Unentscheidbarkeit des Totalitätsproblems.

Beispiel 2: Sei f eine berechenbare Funktion. Sei $S = \{f\}$. Diese Menge erfüllt die Voraussetzungen des Satzes von Rice. Also ist die Sprache

$$L(f) := \{\langle M \rangle \mid M \text{ berechnet } f\}$$

nicht entscheidbar. Soll also eine DTM M (oder ein Programm) entworfen werden, dass die Funktion f entscheidet, so ist es nicht möglich, zu verifizieren, ob die DTM M das Gewünschte leistet, nämlich f bei allen Eingaben berechnet.

Kapitel 3

Zeitkomplexität und die Klasse P

In diesem Kapitel werden wir formal die Zeitkomplexität eines Algorithmus und einer Turingmaschine definieren. Wir werden dann die Klasse P der in Polynomialzeit entscheidbaren Sprachen definieren.

3.1 Motivation, Überblick und einführende Beispiele

Ein Problem, das entscheidbar und damit prinzipiell auf einem Computer lösbar ist, kann in der Praxis nur schwer oder fast gar nicht lösbar sein. Benötigt jeder Algorithmus zur Lösung eines Problems sehr viel Zeit und/oder sehr viel Speicher, so ist ein Problem praktisch auf einem Computer nicht lösbar. Sind wir also an der praktischen und nicht nur der prinzipiellen Lösbarkeit eines Problems interessiert, müssen wir untersuchen, ob es zeit- und speichereffiziente Algorithmen für ein Problem gibt. Ziel der Komplexitätstheorie ist es daher, Probleme entsprechend ihrer zeit- oder speichereffizienten Lösbarkeit zu klassifizieren.

Für eine solche Klassifizierung benötigen wir zunächst einmal ein geeignetes Maß für den Zeit- und Speicherbedarf eines Problems. Hierzu werden wir wieder Turingmaschinen nutzen, die uns ein geeignetes Maß liefern werden. Dann werden wir definieren, was unter einer effizienten Lösung eines Problems zu verstehen ist. Das führt auf die Klasse P aller Probleme, die sich in Polynomialzeit lösen lassen. Wir werden sehen, dass die Klasse P unabhängig ist vom Rechnermodell, das wir zugrunde legen.

Wir werden dann feststellen, dass es viele interessante Probleme gibt, von denen nicht bekannt ist, ob sie in P liegen oder nicht. Unter diesen gibt es einige, die eine wesentliche Eigenschaft gemeinsam haben. Bei diesen Problemen ist es zwar möglicherweise schwierig, eine Lösung zu finden, aber es ist relativ einfach zu überprüfen, ob eine gefundene Lösung korrekt ist. Um diese Eigenschaft präzise zu fassen, führen wir die Klasse NP und die *nichtdeterministischen Turingmaschinen* ein. Schließlich kommen wir dann zum Begriff der *NP-Vollständigkeit*. Dieser Begriff charakterisiert die schwierigsten Probleme in der Klasse NP. Um dabei die Schwierigkeit von Problemen miteinander vergleichen zu können, kommen wir zurück zum Begriff der *Reduktionen*, den wir schon im Zusammenhang mit der Berechenbarkeit kennengelernt haben. Wir werden den Begriff der Reduktion zu *Polynomialzeitreduktionen* verfeinern. Es wird sich herausstellen, dass die NP-vollständigen Probleme, obwohl aus den verschiedensten Bereichen kommend, in einem wohldefinierten Sinn alle gleich schwer sind. Insbesondere hat ein beliebiges der NP-vollständigen Probleme genau dann eine effiziente Lösung, wenn *alle* NP-vollständigen Probleme eine effiziente Lösung besitzen.

Unter den NP-vollständigen Problemen befinden sich viele praktisch relevante. Die Theorie der NP-Vollständigkeit sagt uns, dass diese Probleme aller Voraussicht nach nicht effizient

gelöst werden können. Da diese Probleme aber in der Praxis immer wieder auftauchen, müssen sie trotzdem irgendwie gelöst werden. Dieses wird uns zu *Heuristiken* und sogenannten *Approximationsalgorithmen* führen.

Zum Abschluss dieses Überblicks wollen wir uns einige Beispiele anschauen, um einen Eindruck zu bekommen, wann Probleme noch effizient lösbar sind. Wir beginnen mit Beispielen, die aus der Vorlesung "Datenstrukturen und Algorithmen" bekannt sind.

Beispiel 1 Gegeben sind n natürliche Zahlen $a_1, a_2, \ldots, a_n$. Gesucht ist das (oder ein) minimales Element unter diesen n Zahlen. In "Datenstrukturen und Algorithmen" haben Sie gelernt, dass dieses Problem durch einen Algorithmus gelöst werden kann, der genau $n-1$ Vergleiche durchführt. In der $\mathcal{O}$-Terminologie, die ebenfalls aus "Datenstrukturen und Algorithmen" bekannt ist (kurze Wiederholung unten), benötigt der Algorithmus $\mathcal{O}(n)$ Vergleiche. Dieses ist sicherlich eine effiziente Lösung zur Bestimmung des Minimums von n Zahlen.

Beispiel 2 Wiederum sind n natürliche Zahlen $a_1, \ldots, a_n$ gegeben. Diesmal sollen die Zahlen aufsteigend sortiert werden. Eine Möglichkeit, dieses Problem zu lösen, ist es, $n-1$-mal ein Minimum auszurechnen. Dieses führt zu einem Algorithmus der $\mathcal{O}(n^2)$ Vergleiche benötigt. Sie haben aber im zweiten Semester bereits gehört, dass Algorithmen wie Quicksort oder Heapsort nur $\mathcal{O}(n \log(n))$ Vergleiche benötigen, um n Zahlen zu sortieren. Auch dieses sind effiziente Lösungen des Sortier-Problems

Beispiel 3 Das folgende Problem nennt man das *Problem des Handlungsreisenden*, abgekürzt *TSP* nach dem englischen Titel *Traveling Salesman Problem.* Hierbei sind n Städte $s_1, \ldots, s_n$ gegeben, sowie Distanzen $d_{ij}, 1 \leq i < j \leq n$, zwischen je zwei Städten. Ein Handlungsreisender möchte beginnend und endend in s_1 alle Städte genau einmal besuchen. Dabei möchte er eine möglichst geringe Strecke zurücklegen. Anders formuliert, der Handlungsreisende möchte eine *Rundreise mit minimaler Länge* durch alle Städte s_i finden.

Wie kann eine solche Rundreise minimaler Länge bestimmt werden? Hier ist eine einfache Lösung. Für alle möglichen Rundreisen bestimme man ihre Länge und wähle dann unter diesen eine Rundreise minimaler Länge aus. Um den Aufwand, den dieser Ansatz erfordert, abzuschätzen, müssen wir zunächst bestimmen, wie viele mögliche Rundreisen es gibt. Wir sagten, der Handlungsreisende möchte in s_1 seine Reise beginnen und beenden. Für die zweite Stadt auf der Rundreise gibt es nun $n-1$ Möglichkeiten, ist die zweite Stadt festgelegt, gibt es für die dritte Stadt der Rundreise noch $n-2$ Möglichkeiten. Allgemein, sind die ersten $i-1$ Städte der Rundreise festgelegt, gibt es für die i-te Stadt auf der Rundreise noch $n-i+1$ Möglichkeiten. Insgesamt gibt es damit $(n-1)(n-2)\cdots 2 \cdot 1 = (n-1)!$ viele verschiedene Rundreisen. Für all diese Rundreisen muss die Länge berechnet werden, die sich als Summe von n Distanzen ergibt. Dann muss unter den $(n-1)!$ vielen Längen der Rundreisen eine minimaler Länge bestimmt werden. Der Aufwand den dieser Ansatz benötigt ist sicherlich mindestens $(n-1)!$. Schon für relativ wenige Städte ist dieser Ansatz nicht mehr praktisch durchführbar. Nun hatten wir aber ja schon beim Sortieren gesehen, dass der erste Ansatz nicht unbedingt der beste sein muss. Also, gibt es auch beim Problem des Handlungsreisenden eine bessere Lösung als alle Rundreisen durchzuprobieren? Die Antwort ist, dass es vermutlich keinen wesentlich besseren Ansatz geben kann. Im Laufe der Vorlesung werden wir sehen, was sich hinter den Worten "vermutlich" und "wesentlich besser" genauer verbirgt.

Beispiel 4 Dieses Problem wird das Rucksackproblem genannt. Gegeben sind n Gegenstände, die wir mit $1, 2, \ldots, n$ nummerieren. Der i-te Gegenstand hat Gewicht g_i und Wert w_i. Ein Dieb möchte eine Teilmenge der Gegenstände mit möglichst großem Gesamtwert mitnehmen,

kann aber in seinem Rucksack nur Gegenstände vom Gesamtgewicht G transportieren. Welche Teilmenge S der Gegenstände $1, \dots, n$ sollte er mitnehmen? In Formeln, gesucht ist eine Teilmenge $S \subseteq \{1, \dots, n\}$, so dass $\sum_{i \in S} g_i \leq G$ ist und $\sum_{i \in S} w_i$ möglichst groß ist.

Ein erster Ansatz besteht wie beim Problem des Handlungsreisenden wiederum darin, alle möglichen Lösungen durchzuprobieren und die beste zu nehmen. In diesem Fall bedeutet das, für alle Teilmengen S von $\{1, \dots, n\}$ das Gesamtgewicht und den Gesamtwert der Gegenstände in der Teilmenge zu berechnen. Unter den Teilmengen, deren Gesamtgewicht G nicht überschreitet, wird dann eine ausgewählt, deren Gesamtwert möglichst groß ist.

Der Aufwand für diesen Ansatz ist mindestens so groß wie die Anzahl der Teilmengen S von $\{1, \dots, n\}$. Hiervon gibt es genau 2^n viele. Damit ist der Ansatz schon für $n = 100$ nicht mehr durchführbar. Wiederum können wir fragen, ob es nicht einen besseren Ansatz zur Lösung des Problems gibt. Beim Rucksackproblem gibt es viele andere Ansätze, aber wirklich effizient ist keiner. Stattdessen werden wir sehen, dass das Rucksackproblem und das Problem des Handlungsreisenden in gewisser Hinsicht gleich schwer sind. Beide Probleme sind nämlich NP-*hart*, und damit mindestens so schwer wie ein beliebiges NP-vollständiges Problem.

In den folgenden Tabellen haben wir einmal die Laufzeiten für unsere vier Beispielprobleme miteinander verglichen. Wir haben auch den Sortieralgorithmus mit Laufzeit $\mathcal{O}(n^2)$ aufgenommen. In der ersten Tabelle haben wir angenommen, dass unser Computer 1000 Operationen pro Sekunde ausführen kann. Dies mag wenig erscheinen, aber selbst wenn wir 10^7 Operationen pro Sekunde annehmen würden, sähen die letzten Zeilen der Tabelle ähnlich aus. Wichtig sind nicht so sehr die absoluten Zahlen, sondern die Beobachtung, dass bei Laufzeiten wie 2^n oder $n!$ zusätzliche Rechenzeit nur wenig an der Größe der noch zu lösenden Eingabeinstanzen ändert. Gleiches gilt natürlich für zusätzliche Rechenleistung statt zusätzlicher Rechenzeit. Um diesen Punkt deutlicher zu machen, haben wir in der zweiten Tabelle dargestellt, was eine Erhöhung der Rechenzeit (oder Rechenleistung) um den Faktor 10 für die Größe der noch zu lösende Eingabeinstanzen bedeutet.

$T(n)$	Maximale Größe von n bei vorgegebener Rechenzeit von			
	0,01 Sekunden	1 Sekunde	1 Minute	1 Stunde
n	10	1 000	60 000	3 600 000
$n \log n$	4	140	4 893	204 094
n^2	3	31	244	1 897
2^n	3	9	15	21
$n!$	3	6	8	9

$T(n)$	Maximale Eingabelänge vor	nach Erhöhung der Rechengeschwindigkeit	Bemerkungen
n	m	$10 \cdot m$	
$n \log n$	m	(fast)$10 \cdot m$	
n^2	m	$3.16 \cdot m$	$10^{1/2} \approx 3.16$
2^n	m	$m + 3.3$	$\log 10 \approx 3.3$
$n!$	m	$\approx m$	

Unser Vergleich der Laufzeiten der vier Beispielprobleme ist natürlich nicht ganz zulässig. In den ersten beiden Beispielen haben wir die Anzahl der Vergleiche gezählt, die ein Algorithmus zur Lösung des Minimums- bzw. des Sortierproblems braucht. Beim Problem des Handlungsreisenden und beim Rucksackproblem hingegen waren es auch arithmetische Operationen, die wir bei einer genauen Analyse der Algorithmen hätten zählen müssen. Um die Effizienz von

Problemlösungen und Algorithmen miteinander vergleichen zu können, brauchen wir ein einheitliches Kostenmaß für Algorithmen. Glücklicherweise kennen wir schon Turingmaschinen, die uns ein solches Kostenmaß liefern werden. Mit Hilfe von Turingmaschinen werden wir auch formalisieren können, was es für zwei Probleme heißt, im wesentlichen gleich schwer zu sein.

3.2 $\mathcal{O}$-Notation

Hier sind noch einmal kurz die wichtigsten Notationen zur Größenordnung zusammengefasst.

1. $f = \mathcal{O}(g) \Leftrightarrow \exists c > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) \leq c \cdot g(n)$.
   Man sagt: f wächst asymptotisch höchstens so schnell wie g.

2. $f = \Omega(g) \Leftrightarrow g = \mathcal{O}(f)$.
   Man sagt: f wächst mindestens so schnell wie g.

3. $f = \Theta(g) \Leftrightarrow g = \mathcal{O}(f)$ und $f = \mathcal{O}(g)$.
   Man sagt: f und g wachsen asymptotisch gleich schnell.

4. $f = o(g) \Leftrightarrow \forall c > 0\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : f(n) < c \cdot g(n)$.
   Man sagt: f wächst asymptotisch langsamer g.

5. $f = \omega(g) \Leftrightarrow g = o(f)$.
   Man sagt: f wächst asymptotisch schneller als g.

3.3 Zeitkomplexität einer Turingmaschine

Um die Laufzeit von Algorithmen und Turingmaschinen vergleichen zu können, führen wir als einheitliches Kostenmaß die Rechenschritte einer Turingmaschine ein (zu den Details von Turingmaschinen siehe Abschnitt 2.1). Wir werden jetzt wieder Turingmaschinen mit beliebigem Eingabe- und Bandalphabet betrachten.

Definition 3.1 Sei $M = (Q, \Sigma, \Gamma, \delta)$ eine DTM, die bei jeder Eingabe hält.

- Für $w \in \Sigma^*$ ist $\mathrm{T}_M(w)$ definiert als die Anzahl der Rechenschritte von M bei Eingabe w.
- Für $n \in \mathbb{N}$ ist
$$\mathrm{T}_M(n) := \max\{\mathrm{T}_M(w) \mid w \in \Sigma^{\leq n}\}.$$
- Die Funktion $\mathrm{T}_M : \mathbb{N} \to \mathbb{N}$ heißt die *Zeitkomplexität* oder *Laufzeit* der DTM M.

Man beachte, dass wir die Zeitkomplexität nur für DTMs definiert haben, die immer halten. Eine DTM, die bei einigen Eingaben nicht hält, hat keine Zeitkomplexität oder Zeitkomplexität ∞. Zur Erinnerung: $\Sigma^{\leq n}$ enthält alle Wörter aus Σ^*, die aus höchstens n Zeichen von Σ bestehen. $\mathrm{T}_M(n)$ ist also definiert als die maximale Anzahl der Rechenschritte der DTM M bei einer Eingabe der Länge *höchstens* n. Damit ist sichergestellt, dass die Zeitkomplexität oder Laufzeit einer DTM stets eine monoton wachsende Funktion ist.
Wir sagen auch, dass die DTM M *Laufzeit oder Zeitkomplexität* $\mathcal{O}(f(n))$ hat, wenn gilt $\mathrm{T}_M(n) = \mathcal{O}(f(n))$.

Beispiel: Als erstes Beispiel wollen wir uns eine DTM anschauen, die die Sprache $L := \{0^n1^n \mid n \geq 1\}$ entscheidet. Eine solche DTM hatten wir schon in Abschnitt 2.1 auf Seite 8 kennengelernt. Dabei hatten wir die DTM detailliert beschrieben. Wir werden nun die Laufzeit dieser DTM analysieren. Dabei werden wir allerdings nicht die exakte formale Beschreibung der DTM unserer Analyse zugrunde legen. Vielmehr werden wir eine informelle Beschreibung der DTM benutzen. Diese Beschreibung sollte einerseits präzise genug sein, um eine genaue Analyse zu erlauben. Andererseits sollte die Beschreibung Implementierungsdetails, die für die Analyse unwichtig sind, unterdrücken. Wir werden im weiteren Verlauf der Vorlesung ausschließlich mit solchen Beschreibungen von DTMs arbeiten und auf die formale Beschreibung stets verzichten. Hier ist nun die Beschreibung einer DTM M_1, die $L := \{0^n1^n \mid n \geq 1\}$ entscheidet.

M_1 bei Eingabe $w \in \{0,1\}^*$:

1. Durchlaufe die Eingabe. Falls eine 0 nach einer 1 auftaucht, lehne ab. Sonst gehe zum Beginn des Bandes zurück.
2. Wiederhole den folgenden Schritt, solange noch eine 0 *und* eine 1 auf dem Band steht.
3. Durchlaufe das Band und streiche die erste 0 und letzte 1 auf dem Band. Gehe zum Beginn des Bandes zurück.
4. Falls noch eine 0, aber keine 1 oder noch eine 1, aber keine 0 mehr auf dem Band steht, lehne ab. Sonst akzeptiere.

Wie angekündigt fehlen in dieser Beschreibung viele Details. Wir haben nicht gesagt, wie Symbole gestrichen werden sollen. Eine Möglichkeit ist es, Eingabesymbole durch einen Blank zu ersetzen. Im 4. Schritt haben wir überhaupt nicht gesagt, wie das angegebene Ziel zu erreichen ist. Dieses werden wir in Zukunft häufig in Fällen machen, wo klar sein sollte, wie das angegebene Ziel zu erreichen ist. Im Fall der DTM M_1 etwa muss im 4. Schritt noch einmal über die Symbole auf dem Band gegangen werden.

Zunächst einmal überlegen wir uns, dass die DTM M_1 die Sprache L_1 wirklich entscheidet. Dazu wird im 1. Schritt zunächst überprüft, dass die Eingabe von der Form 0^k1^m ist. Die restlichen Schritte stellen dann sicher, dass nur Eingaben, bei denen zusätzlich $k = m$ gilt, akzeptiert werden.

Um die Laufzeit dieser DTM zu analysieren, werden wir die Laufzeit des 1. und 4. Schrittes getrennt analysieren. Die Laufzeit des 2. und 3. Schrittes analysieren wir zusammen.

Nehmen wir an, die Eingabe w habe Länge n. Zur Durchführung des 1. Schrittes muss einmal über die komplette Eingabe der Länge n gegangen werden. Wird in diesem Schritt nicht abgelehnt, muss dann noch an den Beginn des Bandes zurückgegangen werden. Beides zusammen kann mit $2n = \mathcal{O}(n)$ Rechenschritten erreicht werden.

Nun die Analyse vom 2. und 3. Schritt zusammen. Da bei jedem Durchlauf des 3. Schritts genau zwei Symbole gestrichen werden, stellt die Abbruchbedingung des 2. Schritts sicher, dass der 3. Schritt höchstens $n/2$ mal durchlaufen wird. Jeder Durchlauf des 3. Schritts benötigt wie der erste Schritt höchstens $2n$ Rechenschritte. Daher benötigen die Schritt 2 und 3 gemeinsam höchstens $n^2 = \mathcal{O}(n^2)$ viele Rechenschritte.

Wie oben bereits gesagt, muss zur Realisierung des 4. Schrittes noch einmal über die Symbole auf dem Band der DTM M_1 gegangen werden, um festzustellen, ob noch 0en oder 1en auf dem Band verblieben sind. Dieses erfordert wiederum $n = \mathcal{O}(n)$ Rechenschritte. Insgesamt benötigt die DTM M_1 also bei einer Eingabe der Länge n höchstens $\mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$

viele Rechenschritte. Wir sagen, dass die Sprache L zu den in quadratischer Zeit entscheidbaren Sprachen gehört.

Definition 3.2 Sei $t : \mathbb{N} \to \mathbb{N}$ eine monoton wachsende Funktion. Die Klasse DTIME$(t(n))$ ist definiert als

$$\text{DTIME}(t(n)) := \left\{ L \,\middle|\, \begin{array}{l} L \text{ ist eine Sprache, die von einer DTM mit Laufzeit} \\ \mathcal{O}(t(n)) \text{ entschieden wird.} \end{array} \right\}.$$

In dieser Definition haben wir uns auf Sprachen beschränkt. Die Einschränkung auf Sprachen geschieht wie in Kapitel 2 ausschließlich zur Vereinfachung. Wir sagen von Problemen anderer Art, etwa Optimierungsproblem wie dem Rucksackproblem und dem Problem des Handlungsreisenden im einführenden Kapitel, dass sie in Zeit $\mathcal{O}(t(n))$ lösbar sind, wenn es eine DTM mit Laufzeit $\mathcal{O}(t(n))$ gibt, die das Problem löst.
Die DTM M_1 zeigt nun $L \in \text{DTIME}(n^2)$. Wie aber die nächste DTM zeigt, gibt es eine DTM geringerer Zeitkomplexität, die L ebenfalls entscheidet.

M_2 bei Eingabe $w \in \{0,1\}^*$:

1. Durchlaufe die Eingabe. Falls eine 0 nach einer 1 auftaucht, lehne ab. Sonst gehe zum Beginn des Bandes zurück.
2. Wiederhole die folgenden beiden Schritte, solange noch eine 0 *und* eine 1 auf dem Band steht.
3. Durchlaufe das Band und stelle dabei fest, ob die Anzahl von 0en und und die Anzahl der 1en auf dem Band beide gerade oder beide ungerade sind. Ist dieses nicht der Fall, lehne ab, sonst gehe zum Beginn des Bandes zurück.
4. Durchlaufe das Band und streiche jede zweite 0 und jede zweite 1, beginne dabei mit der ersten 0 bzw. 1.
5. Falls noch eine 0, aber keine 1 oder noch eine 1, aber keine 0 mehr auf dem Band steht, lehne ab. Sonst akzeptiere.

Wiederum müssen wir uns als erstes überlegen, dass die DTM M_2 die Sprache L entscheidet. Wie bei DTM M_1 stellt der 1. Schritt sicher, dass die Eingabe von der Form $0^k 1^m$ ist. Nun stellt der 3. Schritt sicher, dass vor jedem Durchlauf des 4. Schritts, die Anzahl der 0en und die Anzahl der 1en, die auf dem Band verblieben sind, entweder beide gerade oder beide ungerade sind. Nehmen wir nun an, dass die Eingabe akzeptiert wird. Der 5. Schritt stellt dabei sicher, dass keine 0en oder 1en nach den ersten vier Schritten auf dem Band verblieben sind. Setzen wir nun gerade := 0 und ungerade := 1, so überlegt man sich, dass die Folge von gerader/ungerader Anzahl von verbliebenen 0en bzw. 1en gerade die Binärdarstellung der Anzahl von 0en und 1en in der Eingabe ist, beginnend jeweils mit dem untersten Bit. Als Beispiel betrachten wir die Eingabe $0^{13}1^{13}$. Als Folge von verbliebenen 0/1en auf dem Band vor Durchlauf vom 4. Schritt erhalten wir $13, 6, 3, 1$. In ungerade/gerade bedeutet dieses ungerade, gerade, ungerade, ungerade. Als Folge von 0/1en erhalten wir somit 1011. Dieses ist, gelesen von links nach rechts, die Binärdarstellung von $13 = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 + 1 \cdot 8$. Insgesamt stellt die DTM M_2 also sicher, dass die Binärdarstellungen der Anzahl von 0en und der Anzahl von 1en in der Eingabe übereinstimmt und außerdem alle Nullen vor den Einsen stehen. M_2 entscheidet somit L.

Nun die Analyse der Laufzeit von M_2. Der 1. Schritt von M_2 benötigt wie der 1. Schritt von M_1 $2n$ Rechenschritte. Der 5. Schritt von M_2 benötigt wie der 4. Schritt von M_1 ebenfalls genau $2n$ Rechenschritte. Nun die Analyse der Schritte 2, 3 und 4. Im 4. Schritt werden jeweils mindestens die Hälfte aller verbliebenen 0en und 1en gestrichen. Damit stellt die Abbruchbedingung des 2. Schritts sicher, dass die Schritte 3 und 4 nur $1 + \log(n)$-mal durchlaufen werden. Der Logarithmus ist hier, wie auch im Rest der Vorlesung jeweils zur Basis 2. Der 3. und 4. Schritt benötigen jeweils $2n$ Rechenschritte. Damit erhalten wir für die Schritte 2-4 insgesamt $\mathcal{O}(n \log(n))$ Rechenschritte. Die Zeitkomplexität der DTM M_2 ist somit $\mathcal{O}(n \log(n))$ und $L \in \text{DTIME}(n \log(n))$.

Geht es vielleicht noch besser? Gibt es eine DTM mit Laufzeit $o(n \log(n))$, die L entscheidet? Diese Frage muss verneint werden. Es kann nämlich gezeigt werden, dass jede Sprache, die durch eine DTM mit Laufzeit $o(n \log(n))$ entschieden werden kann, *regulär* ist und L nicht regulär ist. Die DTM M_2 ist also in gewisser Weise die optimale (1-Band) DTM für L.

Nimmt man hingegen eine 2-Band DTM, so kann man L mit einer DTM der Zeitkomplexität $\mathcal{O}(n)$ entscheiden. Hierbei ist ein Rechenschritt einer Mehrband DTM wiederum die einmalige Anwendung der Übergangsfunktion der DTM (zu Details von Mehrband Turingmaschinen siehe Abschnitt 2.3). Die Laufzeit oder Zeitkomplexität einer Mehrband DTM ist dann wie bei einer 1-Band DTM erklärt. Hier ist jetzt eine 2-Band DTM mit Laufzeit $\mathcal{O}(n)$, die L entscheidet.

M_3 bei Eingabe $w \in \{0,1\}^*$:

1. Durchlaufe die Eingabe. Falls eine 0 nach einer 1 auftaucht, lehne ab. Sonst gehe zum Beginn des Bandes zurück.
2. Durchlaufe das Band bis zur ersten 1. Kopiere dabei die 0en auf das zweite Band.
3. Gehe auf Band 1 bis ans Ende der Eingabe. Für jede 1 streiche eine 0 auf Band 2. Steht dabei zu irgendeinem Zeitpunkt keine 0 mehr auf Band 2 zur Verfügung, lehne ab.
4. Ist keine 0 mehr auf Band 2, wenn das Ende der Eingabe auf Band 1 erreicht ist, akzeptiere, sonst lehne ab.

Es sollte klar sein, dass die DTM M_3 die Sprache L entscheidet. Jeder einzelne Schritt der DTM M_3 erfordert $\mathcal{O}(n)$ Rechenschritte. Damit ist M_3 eine 2-Band DTM mit Laufzeit $\mathcal{O}(n)$. Mehrband DTMs können also *beweisbar* schneller Sprachen entscheiden als 1-Band DTMs, da jede 1-Band DTM für L eine Laufzeit von $\Omega(n \log(n))$ hat. Mit einer Mehrband DTM kann aber L in Zeit $\mathcal{O}(n)$ entschieden werden. Allerdings gilt

Satz 3.3 *Sei $t : \mathbb{N} \to \mathbb{N}$ eine monoton wachsende Funktion mit $t(n) \geq n$ für alle n. Jede Mehrband DTM mit Laufzeit $t(n)$ kann durch eine 1-Band DTM mit Laufzeit $\mathcal{O}(t(n)^2)$ simuliert werden.*

Zum Beweis dieses Satzes muss man nur zu unserer Simulation einer Mehrband DTM durch eine 1-Band DTM im Beweis von Satz 2.8 auf Seite 16 zurückgehen. Analysiert man die Laufzeiten der zu simulierenden Mehrband DTM und der simulierenden 1-Band DTM, so erhält man Satz 3.3. Der Satz sagt, dass eine Mehrband DTM für eine Sprache höchstens quadratisch schneller sein kann als jede 1-Band DTM für dieselbe Sprache.

3.4 Die Klasse P

Satz 3.3 zeigt, dass die folgende Definition eines effizient lösbaren Problems von dem zugrunde gelegten Rechenmodell (1-Band DTM, Mehrband DTM) weitgehend unabhängig ist.

Definition 3.4 Die Klasse P ist definiert als

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(n^k).$$

In Worten: P ist die Klasse aller Sprachen L, für die es ein festes, aber beliebiges k und eine (1-Band-)DTM M mit Laufzeit $\mathcal{O}(n^k)$ gibt, die L entscheidet. P ist die Klasse der Sprachen, die in *polynomieller Zeit* entschieden werden können.

Wie oben schon angedeutet, folgt aus Satz 3.3, dass sich die Klasse P nicht ändert, wenn wir in Definition 3.4 Mehrband Turingmaschinen als Berechnungsmodell zugrunde legen. Üblicherweise wird sogar davon ausgegangen, dass jedes "vernünftige" Rechenmodell zur selben Klasse von Sprachen führt, die in polynomieller Zeit gelöst werden können. Allerdings kann es sein, dass sich diese Vermutung als falsch herausstellt, da noch nicht klar ist, ob Quantencomputer Probleme in polynomieller Zeit lösen können, die klassische Computer nicht in polynomieller Zeit lösen können. Ein Kandidat dafür wäre z.B. das Primfaktorisierungsproblem, welches von einem Quantencomputer in polynomieller Zeit gelöst werden kann, aber für das es noch unklar ist, ob das auch für klassische Computer möglich ist.
Die Klasse P stellt für uns die Klasse der effizient entscheidbaren Sprachen dar. Allgemeiner werden wir jedes Problem als effizient lösbar bezeichnen, das sich durch eine DTM mit Laufzeit $\mathcal{O}(n^k)$ lösen lässt.
Für unsere Wahl von P als der Klasse der effizient lösbaren Probleme gibt es im wesentlichen drei Gründe.

1. P ist eine mathematisch robuste Klasse.
2. Probleme, die nicht in P liegen, lassen sich in der Praxis nicht effizient lösen. Probleme, die in P liegen, lassen sich häufig auch in der Praxis effizient lösen.
3. Die Klasse P führt zu einer Theorie, die sowohl praktisch als auch theoretisch interessante und relevante Ergebnisse liefert.

Zum ersten Grund: Die Klasse der effizient lösbaren Probleme sollte vom Rechnermodell weitgehend unabhängig sein. Wie wir gesehen haben, ist dieses zumindest für Turingmaschinen (sowie alle klassischen Rechnermodelle) der Fall. Weiter sollten wir DTMs, die effizient sind, miteinander kombinieren können, um immer noch eine effiziente DTM zu erhalten. Bei unserer Definition einer effizienten DTM als einer DTM, deren Laufzeit durch ein Polynom beschränkt ist, ist diese Forderung erfüllt.
Zum zweiten Grund: Für die meisten Probleme, die nicht in P liegen, oder von denen wir nicht wissen, ob sie in P liegen, haben die besten bekannten Algorithmen häufig eine Laufzeit der Form 2^n. In den Tabellen des vorangegangenen Kapitels haben wir gesehen, wie eine solche Laufzeit sich etwa zu einer Laufzeit n^2 verhält. Insbesondere haben wir gesehen, dass mit wachsender Eingabegröße Probleme, deren Lösung Laufzeit 2^n benötigt, sehr viel schneller nicht mehr lösbar sind als Probleme, deren Lösung Laufzeit n^2 benötigt.
Aber in P lassen wir auch Sprachen zu, die von einer DTM mit Laufzeit n^{10000} entschieden werden. Ein Algorithmus dieser Laufzeit ist in der Praxis sicherlich nicht brauchbar. Die Behauptung, eine Sprache, die von einer DTM mit Laufzeit n^{10000} entschieden wird, sei effizient

lösbar, scheint kaum gerechtfertigt zu sein. In der Praxis ist es aber häufig so, dass für Sprachen, von denen bekannt ist, dass sie in P liegen, bald auch DTMs oder Algorithmen bekannt sind, die diese Sprachen entscheiden und deren Laufzeit durch ein kleines Polynom wie n^2 oder n^3 beschränkt ist. Zusammenfassend kann man sagen, dass Probleme, die nicht in P liegen, auch in der Praxis nicht effizient gelöst werden können. Außerdem können Probleme, die in P liegen, meistens auch in der Praxis effizient gelöst werden.

Den dritten oben angeführten Grund für die Definition von P auszuführen, ist Ziel und Inhalt des nächsten Teils der Vorlesung.

Als nächstes wollen wir uns Beispiele für Sprachen in P anschauen. Hier, wie auch später, wollen wir die Sprachen halbformal beschreiben. Da wir aber die Zeitkomplexität als Funktion der Eingabegröße definiert haben, müssen wir zunächst diskutieren, wie wir Eingaben kodieren wollen. Solche Kodierungen sollten es einerseits ermöglichen, die wichtigsten Eigenschaften der Eingabe effizient zu ermitteln, andererseits sollte eine solche Kodierung aber auch möglichst kurz sein. Glücklicherweise werden wir uns in dieser Vorlesung hauptsächlich mit Problemen über Graphen und Zahlen beschäftigen, so dass wir uns hier auf Kodierungen dieser Objekte beschränken werden.

Eine sinnvolle Kodierung einer Zahl ist etwa die Binärdarstellung der Zahl. Aber auch die b-näre Darstellung einer Zahl ist sinnvoll, wobei b irgendeine Basis ist. Einzige Ausnahme ist die unäre Darstellung einer Zahl, also die Darstellung zur Basis 1. Hierbei wird etwa 5 dargestellt durch 11111. Die b-näre Darstellung einer Zahl n hat für $b > 1$ die Länge $\mathcal{O}(\log(n))$, wobei wir hier, wie auch in Zukunft, mit $\log(\cdot)$ stets den Logarithmus zur Basis bezeichnen. Die unäre Darstellung der Zahl n hat Länge n, ist also exponentiell viel länger als die b-näre Darstellung für $b \neq 1$. Die unäre Darstellung einer Zahl ist unnötig lang.

Für Graphen betrachten wir zwei Kodierungen, die Kodierungen von Graphen durch eine Liste der Knoten und Kanten und die Kodierung eines Graphen durch eine Adjazenzmatrix. Ist $G = (V, E)$ ein Graph mit Knotenmenge V und Kantenmenge E, so nehmen wir stets an, dass $V = \{1, \dots, n\}$ für ein $n \in \mathbb{N}$[1]. Die Knotenliste können wir dann einfach durch n beschreiben, genauer durch eine sinnvolle Kodierung von n (siehe oben). Ist G ein ungerichteter Graph, so besteht die Kantenliste aus einer Folge von 2-elementigen Mengen $\{i, j\}, 1 \leq i, j \leq n, i \neq j$. Dabei sind i, j jeweils wieder geeignet kodiert. Bei gerichteten Graphen besteht die Kantenliste aus geordneten Paaren $(i, j), 1 \leq i, j \leq n, i \neq j$, wobei i, j geeignet kodiert sind.

Die Adjazenzmatrix A des Graphen ist eine $n \times n$ Matrix mit Einträgen $A_{ij}, 1 \leq i, j \leq n$. Ist G ein gerichteter Graph, so ist $A_{ij} = 1$, wenn die Kante (i, j) in E enthalten ist, sonst ist $A_{ij} = 0$. Bei einem ungerichteten Graphen sind $A_{ij} = A_{ji} = 1$, falls die Kante $\{i, j\}$ in E enthalten ist, sonst ist $A_{ij} = A_{ji} = 0$. Für das Beispiel des gerichteten Graphen in Abbildung 3.1 sehen die beiden möglichen Kodierungen des Graphen folgendermaßen aus.

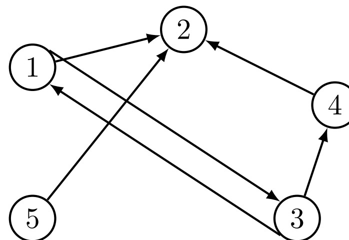


Abbildung 3.1: Ein gerichteter Graph

[1]In der Graphentheorie steht n üblicherweise für die Anzahl der Knoten und m für die Anzahl der Kanten. Das sollte nicht verwechselt werden mit der Eingabegröße n.

Die Knotenliste ist durch eine Kodierung von 5 gegeben. Die Kantenliste besteht aus

$$(1,2),(1,3),(3,1),(3,4),(4,2),(5,2).$$

Hierbei müssen wir eigentlich noch die Knoten kodieren (z.B. binär).
Die Adjazenzmatrix A des Graphen aus Abbildung 3.1 ist

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Bei beiden Kodierungen von Graphen gilt, dass die Länge der Kodierung polynomiell in der Anzahl der Knoten des kodierten Graphen ist. Wollen wir daher von Algorithmen oder DTMs mit Graphen als Eingabe zeigen, dass sie polynomielle Laufzeit besitzen, so genügt es zu zeigen, dass die Laufzeit polynomiell in der Anzahl der Knoten des Graphen ist.

Wir werden uns im allgemeinen nicht auf eine konkrete Kodierung festlegen, stattdessen werden wir die Klammern $\langle\cdot\rangle$ benutzen, um eine sinnvolle Kodierung eines oder mehrerer Objekte zu bezeichnen. Ist x eine ganze Zahl, so bedeutet also $\langle x\rangle$ eine b-näre Kodierung von x. Für $x, y \in \mathbb{Z}$ ist $\langle x, y\rangle$ eine sinnvolle Kodierung des Tupels bestehend aus x und y. Für einen Graphen G und einen Knoten s aus der Knotenmenge V von G ist $\langle G, s\rangle$ eine Kodierung des Graphen G und des Knotens s. Die Länge einer Kodierung bezeichnen wir durch $|\cdot|$. Ist $x \in \mathbb{N}$, so bezeichnet also $|x|$ die Länge der Kodierung $\langle x\rangle$ von x. Nach dem oben Gesagten gilt stets $|x| = \mathcal{O}(\log(x))$.

Nach diesen vorbereitenden Bemerkungen nun unser erstes Beispiel. Gegeben ist ein gerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$. Wir sollen entscheiden, ob es in G einen gerichteten Weg von s nach t gibt. Als Sprache ausgedrückt, suchen wir eine DTM, die die Sprache

$$\text{Path} := \left\{ \langle G, s, t\rangle \;\middle|\; \begin{array}{l} G = (V, E) \text{ ist ein gerichteter Graph mit } s, t \in V \\ \text{und einem gerichteten Weg von } s \text{ nach } t. \end{array} \right\}$$

entscheidet. Betrachten wir als Beispiel den Graphen G in Abbildung 3.1. Dann ist $\langle G, 1, 4\rangle$ in Path enthalten. Ein gerichteter Weg von 1 nach 4 führt zunächst über die Kante $(1,3)$ nach 3 und dann über die Kante $(3,4)$ nach 4. Dagegen ist $\langle G, 1, 5\rangle$ nicht in Path enthalten. Es führt überhaupt keine der gerichteten Kante in 5 hinein. Daher kann es auch keinen Weg von 1 nach 5 geben. Wir zeigen nun

Satz 3.5 *Die Sprache Path liegt in der Klasse* P.

Beweis: Hier zunächst eine DTM, die die Sprache Path entscheidet. Diese DTM implementiert eine *Breitensuche* startend im Knoten s.

M bei Eingabe $\langle G, s, t\rangle$:

1. Markiere den Knoten s.
2. Wiederhole den folgenden Schritt, bis keine Knoten markiert werden.
3. Durchlaufe alle Kanten (a, b) des Graphen G. Ist a markiert und b nicht markiert, markiere b.
4. Ist t markiert, akzeptiere, sonst lehne ab.

Wie in Zukunft immer, müssen wir zunächst die *Korrektheit* dieser DTM zeigen. Mit der Länge eines Weges bezeichnen wir nun die Anzahl der Kanten auf dem Weg. Dann gilt, dass beim ersten Durchlauf vom 3. Schritt der DTM M alle Knoten v markiert werden, die von s aus durch einen Weg der Länge 1 erreicht werden. Beim zweiten Durchlauf werden dann die Knoten markiert, die auf einem Weg der Länge 2, aber nicht der Länge 1 von s aus erreicht werden können. Allgemein werden im k-ten Durchlauf vom 3. Schritt diejenigen Knoten markiert, die von s aus auf einem Weg der Länge k nicht jedoch auf einem Weg kürzerer Länge erreicht werden. Damit wird insbesondere auch t markiert, wenn t von s aus auf einem gerichteten Weg erreicht werden kann. Die Ausgabe im 4. Schritt ist also korrekt und die DTM M entscheidet Path.

Jetzt müssen wir die Laufzeit analysieren und zeigen, dass sie polynomiell ist. Wie oben bereits gesagt, genügt es zu zeigen, dass die Laufzeit polynomiell in der Anzahl n der Knoten des Eingabegraphen G ist.

Der 1. und 4. Schritt werden jeweils nur einmal durchlaufen. Der 3. Schritt kann höchstens n-mal durchlaufen werden. In jedem Durchlauf des 3. Schritts, abgesehen vom letzten, wird mindestens ein zusätzlicher Knoten markiert. Spätestens nach $n-1$ Durchläufen kann beim nächsten Durchlauf kein zusätzlicher Knoten markiert werden. Also wird der 3. Schritt höchstens n-mal durchlaufen.

Der 1. und 4. Schritt können in polynomieller Zeit durchgeführt werden. Aber auch der 3. Schritt erfordert pro Durchlauf nur polynomielle Zeit. Wir müssen ja nur einmal über die gesamte Kantenmenge gehen. Diese aber hat Größe maximal $n(n-1)$. Damit ist die Laufzeit von M polynomiell. □

Man beachte, dass eine *brute-force* oder *erschöpfende Suche* über alle möglichen Wege nicht genügt, um Path $\in$ P zu zeigen. Wenn es einen Weg von s nach t gibt, so gibt es zwar auch einen Weg der Länge höchstens $m-1$ von s nach t. Aber in einem gerichteten Graphen mit n Knoten kann es mehr als $(n-1)!$ Pfade der Länge höchstens $n-1$ geben. $(n-1)!$ ist aber nicht mehr polynomiell in n.

Das nächste Problem, das wir uns anschauen wollen, kommt aus der Zahlentheorie. Zwei Zahlen $x, y \in \mathbb{N}$ heißen *relativ prim* oder *teilerfremd*, wenn keine Zahl $d \in \mathbb{N}, d \geq 2$, existiert, die sowohl x als auch y teilt. Bezeichnen wir mit $\text{ggT}(x, y)$ den *größten gemeinsamen Teiler* von x und y, so sind x, y genau dann teilerfremd, wenn $\text{ggT}(x, y) = 1$ gilt. Gegeben zwei Zahlen x, y, sollen wir nun entscheiden, ob die beiden Zahlen relativ prim sind. Anders ausgedrückt, wir sollen die Sprache

$$\text{RelPrim} := \{\langle x, y\rangle \mid x, y \in \mathbb{N} \text{ sind relativ prim.}\}$$

entscheiden. Es gilt

Satz 3.6 *Die Sprache RelPrim liegt in der Klasse* P.

Beweis: Zunächst beachte man, dass nicht einfach für alle Zahlen $d \leq x$ überprüft werden kann, ob sie sowohl x als auch y teilen. Die Anzahl der Zahlen $d \leq x$ ist nicht polynomiell in der Darstellungsgröße von x. Stattdessen benutzen wir als Hilfsmittel den *Euklidischen Algorithmus* (siehe die DTM E).

Mit Hilfe der DTM E kann dann eine DTM R formuliert werden, die die Sprache RelPrim entscheidet.

Wenn wir zeigen können, dass die Ausgabe von E genau $\text{ggT}(x, y)$ ist, so folgt die Korrektheit von R. Weiter ist nun die Ausgabe der DTM E der größte gemeinsame Teiler $\text{ggT}(x, y)$ von x und y, wenn wir die folgende Gleichung für alle $x, y \in \mathbb{N}$ zeigen können

$$\text{ggT}(x, y) = \text{ggT}(x \bmod y, y). \tag{3.1}$$

E bei Eingabe $\langle x, y\rangle$:

1. Wiederhole die folgenden beiden Schritte bis $y = 0$.
2. $\quad x \leftarrow x \mod y$
3. $\quad$ Vertausche x und y.
4. Ausgabe x.

R bei Eingabe $\langle x, y\rangle$:

1. Simuliere E mit Eingabe $\langle x, y\rangle$.
2. Ist die Ausgabe von E bei Eingabe $\langle x, y\rangle$ 1, akzeptiere, sonst lehne ab.

Um diese Gleichung zu zeigen, setzen wir $z := x \bmod y$. Nach Definition von Division mit Rest kann x geschrieben werden als $x = k \cdot y + z$, wobei $k \in \mathbb{N}$. Damit sehen wir, dass jeder gemeinsame Teiler von y und z auch x teilen muss. Andererseits folgt aus $z = x - ky$, dass jeder gemeinsame Teiler von x und y auch z teilen muss. Diese beiden Beobachtungen zusammen beweisen nun Gleichung (3.1).

Die Laufzeit von R ist polynomiell, wenn die Laufzeit von E polynomiell ist. Um die Laufzeit von E zu analysieren, beobachten wir zunächst, dass jeder einzelne Schritt von E nur polynomielle Laufzeit benötigt. Nicht ganz offensichtlich ist dies beim 2. Schritt. Aber alle arithmetischen Operationen, inklusive der Division mit Rest, benötigen nur polynomielle Laufzeit. Es bleibt noch zu zeigen, dass der 2. und 3. Schritt nur polynomiell häufig durchlaufen werden. Können wir nun zeigen, dass mit Ausnahme eventuell des ersten Durchlaufs jeder Durchlauf des 2. Schritts die Größe von x halbiert, so können der 2. und 3. Schritt nur $\mathcal{O}(\max\{\log(x), \log(y)\})$ häufig durchlaufen werden.

Nach jedem Durchlauf des 2. Schritts gilt $x < y$. Im 3. Schritt werden x und y vertauscht. Daher gilt nach dem 3. Schritt $x > y$. Um zu zeigen, dass nach jedem Durchlauf des 2. Schritts x halbiert worden ist, unterscheiden wir die Fälle

- $x/2 \geq y$ vor Ausführung des 2. Schritts
- $x/2 < y$ vor Ausführung des 2. Schritts.

Im Fall $x/2 \geq y$ gilt nach Ausführung des 2. Schritts $x \mod y < y \leq x/2$ wie gefordert. Im Fall $x/2 < y$ gilt $x \mod y = x - y < x/2$, ebenfalls wie gefordert. Damit hat E und dann auch R polynomielle Laufzeit. □

3.5 Die Klasse NP

In den beiden Beispielen des vorangegangenen Abschnitts war es wichtig, dass wir jeweils eine brute-force Suche, ein komplettes Durchsuchen aller Lösungsmöglichkeiten, verhindern. Andernfalls hätten wir weder von Path noch von RelPrim zeigen können, dass sie in P liegen. Nun gibt es aber viele interessante und in der Praxis nützliche Problem, von denen bislang noch nicht gezeigt werden konnte, dass zu ihrer Lösung eine brute-force Suche vermieden werden kann. Für diese Probleme ist also nicht bekannt, ob sie in P liegen. Ein Grund hierfür könnte sein, dass es DTMs polynomieller Laufzeit für diese Probleme nicht gibt. Um dieses zu beweisen, muss man etwa zeigen, dass jede DTM, die ein Problem Π löst, Laufzeit $\Omega(2^n)$ hat. Eine Aussage dieser Form wird *untere Schranke* genannt. Leider kann man derzeit von keinem der im letzten Abschnitt genannten Problem eine untere Schranke von der Form $\omega(n^2)$ zeigen. Wir werden jedoch sehen, dass für eine große Klasse von Problemen, für die wir weder polynomielle DTMs noch aussagekräftige untere Schranken haben, die Komplexität der Probleme in dieser Klasse eng miteinander verknüpft ist. Können wir nämlich bei einem Problem aus dieser Klasse eine brute-force Suche vermeiden und eine DTM polynomieller Laufzeit konstruieren, so gilt dieses auch für alle anderen Probleme in der Klasse. Um dieses genau zu verstehen, benötigen wir den Begriff eines *Verifizierers*, die Klasse NP und schließlich den Begriff der NP-*Vollständigkeit*.

3.5.1 Verifizierer und die Klasse NP

Zunächst die Definition eines Verifizierers. Hierzu gehen wir zurück zum Rucksackproblem, dass wir in der Einleitung kennengelernt haben. Wir werden dieses Problem jetzt mit RS_{opt} bezeichnen. Hier noch einmal die Definition. Gegeben sind n Gegenstände, die wir mit $1, 2, \dots, n$ durchnummerieren. Der i-te Gegenstand hat Gewicht g_i und Wert w_i. Ein Dieb möchte eine Teilmenge der Gegenstände mit möglichst großem Gesamtwert mitnehmen, kann aber in seinem Rucksack nur Gegenstände vom Gesamtgewicht g transportieren. Welche Teilmenge S der Gegenstände $1, \dots, n$ sollte er mitnehmen? In Formeln, gesucht ist eine Teilmenge $S \subseteq \{1, \dots, n\}$, so dass $\sum_{i \in S} g_i \leq g$ ist und $\sum_{i \in S} w_i$ möglichst groß ist. RS_{opt} passt nicht ganz in unsere Terminologie. Es ist ein sogenanntes *Optimierungsproblem* (daher opt) und kann nicht als Sprache formuliert werden. Wir ändern das Problem etwas ab. Das neue Problem nennen wir RS_{ent}. Gegeben sind wie bei RS_{opt} Gegenstände $1, \dots, n$, Gewichte g_i und Werte w_i. Zusätzlich zu dem zulässigen Gesamtgewicht g ist aber auch noch eine Mindestwert w spezifiziert. Entschieden werden soll, ob es eine Teilmenge S der Gegenstände gibt, die Gesamtgewicht höchstens g und Gesamtwert mindestens w hat. In Formeln, entschieden werden soll, ob es ein S mit

$$\sum_{i \in S} g_i \leq g$$

$$\sum_{i \in S} w_i \geq w$$

gibt. Dieses nennt man ein *Entscheidungsproblem*. Es kann auch als Sprache formuliert werden. Hierzu setzen wir $G := \{g_1, g_2, \dots, g_n\}$ als die Menge der Gewichte der Gegenstände und $W := \{w_1, w_2, \dots, w_n\}$ als die Menge der Werte der Gegenstände. Dann ist

$$\text{RS}_{\text{ent}} := \left\{ \langle G, W, g, w \rangle \;\middle|\; \begin{array}{l} \text{es existiert ein } S \subseteq \{1, \dots, n\} \text{ mit } \sum_{i \in S} g_i \leq g \\ \text{und } \sum_{i \in S} w_i \geq w \end{array} \right\}.$$

Auf den ersten Blick scheint RS_{ent} einfacher zu sein als RS_{opt}. Wir werden jedoch später in einem kleinen Exkurs sehen, dass dieser Eindruck täuscht. Wir werden zeigen, dass wir RS_{opt} effizient lösen können, sobald wir RS_{ent} effizient lösen können.

Man kann RS_{ent} mit Hilfe einer brute-force Suche lösen. D.h., es wird für jede Teilmenge $S \subseteq \{1, \ldots, n\}$ überprüft, ob sie Gesamtgewicht höchstens g und Gesamtwert mindestens w hat. Dieses führt allerdings zu keinem polynomiellen Algorithmus, denn die Anzahl der möglichen Teilmengen ist 2^n.

Man kennt zur Zeit keinen polynomiellen Algorithmus für RS_{ent}. Aber RS_{ent} hat eine andere interessante Eigenschaft. Diese nennt man *polynomielle Überprüfbarkeit* oder *polynomielle Verifizierbarkeit*. Denn obwohl wir keinen polynomiellen Algorithmus kennen, der uns eine Menge S mit $\sum_{i \in S} g_i \leq g$ und $\sum_{i \in S} w_i \geq w$ berechnet, so können wir doch von jeder Menge S in polynomieller Zeit überprüfen, ob sie eine Lösung mit den gewünschten Eigenschaften ist. Hierzu müssen wir nur die Summe der Gewichte und die Summe der Werte der Gegenstände in S berechnen. Anders ausgedrückt, hat jemand irgendwie eine Lösung S, so kann diese Person auch Sie davon überzeugen, dass es eine Lösung S gibt. Ihnen muss nur die Lösung S genannt werden. Danach können Sie dann in polynomieller Zeit nachprüfen, dass S wirklich eine Lösung ist. Dieses führt auf die folgende Definition.

Definition 3.7 Sei L eine Sprache.

- Eine DTM V heißt *Verifizierer* für L, falls

$$L = \{w \mid \text{es gibt ein Wort } c\text{, so dass } V \text{ die Eingabe } \langle w, c \rangle \text{ akzeptiert}\}.$$

- Ein Verifizierer V für L heißt *polynomieller Verifizierer* für L, wenn es ein $k \in N$ gibt, so dass für alle $w \in L$ ein c mit $|c| \leq |w|^k$ existiert und V die Eingabe $\langle w, c \rangle$ akzeptiert. Weiter muss die Laufzeit von V bei jeder Eingabe $\langle w, c \rangle$ polynomiell in der Länge $|w|$ von w sein.

- Existiert ein polynomieller Verifizierer für eine Sprache L, so heißt L *polynomiell verifizierbar.*

Ein Wort c, für das $\langle w, c \rangle$ von V akzeptiert wird, heißt *Zertifikat (engl. certificate)* oder *Zeuge* für $w \in L$.

Damit ein w in L enthalten ist, wird nur ein c benötigt, so dass V die Eingabe $\langle w, c \rangle$ akzeptiert. Umgekehrt, gibt es ein solches c, so muss w auch in L liegen. Die Definition verlangt *nicht*, dass es auch für $w \notin L$ ein Zertifikat gibt, dass dieses belegt. Wie wir sehen werden, gibt es vermutlich so etwas häufig nicht.

Bei einem polynomiellen Verifizierer muss die Länge $|c|$ eines Zertifikats c für $w \in L$ polynomiell in der Länge $|w|$ des Wortes w sein.

Den Verifizierer für RS_{ent}, den wir oben skizziert hatten, wollen wir nun etwas genauer beschreiben, und uns dann überzeugen, dass er ein polynomieller Verifizierer für RS_{ent} ist.

V_1 bei Eingabe $\langle G, W, g, w, S \rangle$:

1. Teste, ob $\langle S \rangle$ die Kodierung einer Teilmenge von $\{1, \ldots, n\}$ ist.
   Ist dieses nicht der Fall, lehne ab.
2. Teste, ob $\displaystyle\sum_{i \in S} g_i \leq g$ und $\displaystyle\sum_{i \in S} w_i \geq w$. Falls ja, akzeptiere.
   Sonst lehne ab.

Durch den 2. Schritt ist sichergestellt, dass es nur für Elemente $\langle G, W, g, w\rangle \in \mathrm{RS}_{\mathrm{ent}}$ eine Teilmenge S geben kann, so dass V die 5-Tupel $\langle G, W, g, w, S\rangle$ akzeptiert. Somit ist V_1 ein Verifizierer für $\mathrm{RS}_{\mathrm{ent}}$.

Um zu zeigen, dass wir diesen Verifizierer auf einer DTM so umsetzen können, dass die Laufzeit polynomiell ist, nehmen wir an, dass eine Teilmenge S der Zahlen zwischen 1 und n durch eine Bitfolge der Länge n realisiert ist. Ist das i-te Bit dieser Folge 1, so bedeutet dieses $i \in S$. Entsprechend bedeutet eine 0 an der i-ten Stelle $i \notin S$. Mit einer solchen Codierung einer Teilmenge S kann der erste Schritt des Verifizierers V_1 in linearer Zeit umgesetzt werden.

Um den zweiten Schritt zu analysieren, benutzen wir, dass zwei natürliche Zahlen mit jeweils höchstens b Bits in Zeit $\mathcal{O}(b^2)$ addiert werden können. Um die beiden Ungleichungen in 2. zu testen, müssen wir höchsten n-mal zwei Zahlen addieren. Die Zahlen müssen dabei jeweils weniger als $\log g$, bzw. $\log w$ Bits besitzen, denn andernfalls kann der Verifizierer ablehnen. Damit kann der zweite Schritt in Zeit $\mathcal{O}(n \max\{\log^2 g, \log^2 w\})$ umgesetzt werden. Dieses ist, wie auch die Laufzeit für den ersten Schritt, polynomiell in $\langle G, W, g, w\rangle$. Damit ist V_1 ein polynomieller Verifizierer für $\mathrm{RS}_{\mathrm{ent}}$.

Als zweites Beispiel betrachten wir das Problem des Handlungsreisenden. Wie wir im einleitenden Abschnitt dieses Kapitels gesehen haben, ist das Problem des Handlungsreisenden ein Optimierungsproblem, aber wie beim Rucksackproblem können wir auch hier ein Entscheidungsproblem formulieren. Genauer sei eine Landkarte mit n Städten gegeben, die die Namen $1, \ldots, n$ tragen. Die Entfernungen zwischen den Städten sind in einer $n \times n$-Matrix $\Delta = (d_{ij})$ zusammengefasst, wobei $d_{ij} \in \mathbb{N}$, $d_{ii} = 0$ und $d_{ij} = d_{ji}$ gilt.

Das Problem des Handlungsreisenden (*Travelling Salesman Problem, TSP*) ist die Sprache

$$\mathrm{TSP}_{\mathrm{ent}} := \left\{ (\Delta, L) \;\middle|\; \begin{array}{l} L \in \mathbb{N}, \text{es gibt eine Rundreise durch alle } n \text{ Städte der} \\ \text{Länge} \le L \end{array} \right\}.$$

Es ist nicht bekannt, ob die Sprache $\mathrm{TSP}_{\mathrm{ent}}$ in P liegt. Es existiert jedoch ein einfacher Verifizierer für die Sprache $\mathrm{TSP}_{\mathrm{ent}}$. Als Zeugen werden wir Rundreisen nehmen. Dabei beschreiben wir eine Rundreise durch die n Städte $s_1, \ldots, s_n$ durch eine Permutation auf den Zahlen $1, \ldots, n$. Der Permutation π entspricht dabei die Rundreise, die die Städte in der Reihenfolge $s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(n)}, s_{\pi(1)}$ besucht. Eine Permutation π können wir durch die Folge $\pi(1), \pi(2), \ldots, \pi(n)$ beschrieben. Die Länge dieser Beschreibung ist $\mathcal{O}(n \log(n))$. Hier nun der Verifizierer für $\mathrm{TSP}_{\mathrm{ent}}$.

V_2 bei Eingabe $\langle \Delta, L, \pi\rangle$:

1. Teste, ob $\langle \pi\rangle$ die Kodierung einer Permutation der Zahlen zwischen 1 und n ist. Ist dieses nicht der Fall, lehne ab.
2. Teste, ob $\sum_{i=1}^{n} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)} \le L$. Falls ja, akzeptiere. Sonst lehne ab.

Zeigen wir zunächst, dass dieses ein Verifizierer für die Sprache $\mathrm{TSP}_{\mathrm{ent}}$ ist. Eine Eingabe $\langle \Delta, L, \pi\rangle$ wird genau dann von V_2 akzeptiert, wenn

$$\sum_{i=1}^{n} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)} \le L$$

gilt. Nun ist aber die Summe $\sum_{i=1}^{n} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)} \le L$ genau die Länge der durch die Permutation π beschriebenen Rundreise. Wird $\langle \Delta, L, \pi\rangle$ akzeptiert, so existiert nach 2. in V_2 das Paar $\langle \Delta, L\rangle$ in $\mathrm{TSP}_{\mathrm{ent}}$. Ist andererseits $\langle \Delta, L\rangle \in \mathrm{TSP}_{\mathrm{ent}}$, so existiert auch eine Rundreise π

mit Länge höchstens L. Es existiert dann also ein π, so dass V_2 das Triple $\langle \Delta, L, \pi \rangle$ akzeptiert. Damit ist V_2 ein Verifizierer für TSP_{ent}.

Nun betrachten wir die Laufzeit von V_2. Wir nehmen an, dass alle Zahlen binär dargestellt sind. Außerdem sei $D := \sum_{i,j=1}^{n} d_{ij}$. Damit ist D eine obere Schranke für die Länge jeder Rundreise. Wir können annehmen, dass $L \leq D$. Dann ist die Eingabegröße von $\langle \Delta, L \rangle$ mindestens $\log(D)$. Um zu testen, ob $\langle \pi \rangle$ eine Permutation der Zahlen von 1 bis n beschreibt, müssen wir prüfen, ob $\langle \pi \rangle$ n Zahlen der Größe höchstens n kodiert und diese Zahlen alle unterschiedlich sind. Für letzteres können wir z.B. eine DTM für die Sprache Element-Distinctness nutzen (siehe Seite 14). Der 1. Schritt kann damit in polynomieller Zeit umgesetzt werden.

Um die Laufzeit des zweiten Schritts zu analysieren, können wir wie in der Analyse des Verifizierers für RS_{ent} argumentieren und erhalten $\mathcal{O}(n \log^2(D))$ als Abschätzung für die Laufzeit des 2. Schritts von V_2. Insgesamt haben wir gezeigt, dass V_2 ein polynomieller Verifizierer für die Sprache TSP_{ent} ist.

Nach diesen Beispielen kommen wir zur Definition der Klasse NP.

Definition 3.8 NP ist die Klasse aller Sprachen, die polynomiell verifizierbar sind.

NP steht für *nichtdeterministische Polynomialzeit.* Dieser Name wird sich in Kürze klären. NP steht *nicht* für "nicht polynomiell".

Sei L eine Sprache in P. Weiter sei M eine DTM, die L in polynomieller Zeit entscheidet. Wir können M auch als einen Verifizierer für L auffassen, der bei Eingabe w keine Hilfe c benötigt. Daher gilt

Satz 3.9 $\text{P} \subseteq \text{NP}$.

Dieser Satz erlaubt noch zwei Möglichkeiten für das Verhältnis von P und NP. Entweder sind P und NP identisch oder P ist echt in NP enthalten. Die Antwort auf diese Frage ist nicht bekannt.

Offenes Problem Ist $\text{P} = \text{NP}$ oder $\text{P} \subset \text{NP}$.

Dieses Problem ist als das $\text{P} - \text{NP}$-*Problem* bekannt. Es ist das wichtigste offene Problem der theoretischen Informatik und eines der wichtigsten Probleme der Mathematik.

3.5.2 Nichtdeterministische Turingmaschinen und die Klasse NP

Nun wollen wir eine andere Charakterisierung von NP geben, die auch den Namen NP erklärt. Hierzu benötigen wir zunächst *nichtdeterministische Turingmaschinen (NTM)*. NTMs sind keine realistischen Rechnermodelle, sie sind aber als mathematisches Modell von großem Nutzen. Befindet sich eine DTM im Zustand q und liest das Symbol a, so ist der nächste Schritt der DTM durch die Zuordnung $\delta(q,a) = (q', b, D)$ der Zustandsfunktion eindeutig festgelegt. Eine NTM dagegen hat Wahlmöglichkeiten. Dies bedeutet, dass die NTM bei gegebenem Zustand q und gelesenem Symbol a unter verschiedenen möglichen Schritten einen auswählen darf. Wir formalisieren dieses, indem die Übergangsfunktion δ einer NTM einem Paar (q,a) nicht mehr ein einzelnes Tripel (q', b, D) sondern eine endliche Menge $\{(q_1, b_1, D_1), \ldots, (q_k, b_k, D_k)\}$ von Tripeln zuordnet. Um dieses zu formalisieren, benötigen wir den Begriff der Potenzmenge einer Menge. Für eine beliebige Menge S ist die *Potenzmenge* $\mathcal{P}(S)$ von S die Menge aller Teilmengen von S. Eine 1-Band NTM können wir nun wie folgt definieren.

Definition 3.10 Eine nichtdeterministische *Turingmaschine* (NTM) wird durch ein 4-Tupel $M = (Q, \Sigma, \Gamma, \delta)$ beschrieben. Dabei sind Q, Σ, Γ endliche, nicht-leere Mengen und

1. Q ist die *Zustandsmenge*. Diese enthält die ausgezeichneten Zustände $q_0, q_{\text{accept}}, q_{\text{reject}}$, wobei $q_{\text{accept}} \neq q_{\text{reject}}$ gelten muss. q_0 ist der *Startzustand*, q_{accept} ist der *akzeptierende Zustand* und q_{reject} ist der *ablehnende Zustand*.

2. Σ ist das *Eingabealphabet*, wobei das *Blank* $\sqcup$ und das *Startsymbol* $\rhd$ nicht in Σ liegen.

3. Γ ist das *Bandalphabet*, wobei $\Sigma \subset \Gamma$ und $\sqcup, \rhd \in \Gamma$.

4. $\delta : Q \backslash \{q_{\text{accept}}, q_{\text{reject}}\} \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{R, L\})$ ist die *Übergangsfunktion*. Für $q_{\text{accept}} \times \Gamma, q_{\text{reject}} \times \Gamma$ ist δ also *nicht* definiert.

5. Für das Symbol $\rhd \in \Gamma$ und alle Zustände $q \in Q$ gilt $\delta(q, \rhd) \subseteq \mathcal{P}(Q \times \{\rhd\} \times \{R\})$. Außerdem muss für alle $q \in Q$, $a \in \Gamma, a \neq \rhd$, gelten: $\delta(q,a) \subseteq \mathcal{P}(Q \times \Gamma \backslash \{\rhd\} \times \{L, R\})$. Mit anderen Worten, $\rhd$ kann nur dann als zweite Koordinate eines Elementes von $\delta(q,a)$ auftauchen, wenn auch $a = \rhd$.

Die Zuordnung $\delta(q,a) = S$ mit $S \in \mathcal{P}(Q \times \Gamma \times \{R, L\})$ bedeutet dabei Folgendes. Befindet sich die NTM N im Zustand q und liest das Symbol a, so kann sich die NTM N aussuchen, welchen der durch die Tripel in S beschriebenen Rechenschritte sie ausführt. Ist also $\delta(q,a) = \{(q', a', L), (q'', a'', R)\}, q, q', q'' \in Q, a, a', a'' \in \Gamma$, so kann die NTM N, befindet sie sich im Zustand q und liest das Symbol a, im nächsten Schritt *entweder* das Symbol a durch a' ersetzen, in den Zustand q' wechseln und den Kopf eine Zelle nach links bewegen, *oder* die NTM N kann das Symbol a durch a'' ersetzen, in den Zustand q'' wechseln und den Kopf eine Zelle nach rechts bewegen.

Allerdings gelten beim Lesen des Startsymbols $\rhd$ für eine NTM für alle möglichen Schritte dieselben Einschränkungen wie für eine DTM (vergleiche Definition 2.1).

Betrachten wir ein erstes Beispiel N_1 einer NTM. Die NTM N_1 hat $Q = \{q_0, q_1, q_2, q_3\}$, wobei q_0 der Startzustand ist und $q_2 = q_{\text{accept}}$, sowie $q_3 = q_{\text{reject}}$. Weiter ist $\Sigma = \{0, 1\}$ und $\Gamma = \{0, 1, \sqcup, \rhd\}$. Die Übergangsfunktion ist wie in Abbildung 3.2 definiert.

Wir können die Begriffe von Konfiguration, Nachfolgekonfiguration, Berechnung und akzeptierender sowie ablehnender Konfiguration nun von DTMs (siehe Abschnitt 2.1) auf NTMs übertragen. Zu einer Konfiguration $K = \alpha q a \beta, \alpha, \beta \in \Gamma^*, a \in \Gamma, q \in Q$, kann es jedoch anders als bei einer DTM verschiedene Nachfolgekonfigurationen geben. Die Anzahl möglicher Nachfolgekonfigurationen ist dabei abhängig von der Größe von $\delta(q,a)$. Entsprechend kann bei einer

δ	0	1	$\rhd$	$\sqcup$
q_0	$\{(q_0, 0, R), (q_3, 0, R)\}$	$\{(q_1, 1, R)\}$	$\{(q_0, \rhd, R)\}$	$\{(q_3, \sqcup, R)\}$
q_1	$\{(q_0, 0, R), (q_3, 0, R)$	$\{(q_2, 1, L)\}$	$\{(q_3, \rhd, R)\}$	$\{(q_3, \sqcup, R)\}$

Abbildung 3.2: Erstes Beispiel der Übergangsfunktion der NTM N_1.

festen Eingabe $w \in \Sigma^*$ eine NTM N_1 viele verschiedene Berechnungen ausführen, je nachdem welche Elemente aus $\delta(q, a)$ in den einzelnen Rechenschritten von der NTM jeweils ausgewählt werden. D.h., abhängig von den Elementen, die aus den Mengen $\delta(q, a)$ gewählt werden, durchläuft die NTM N_1 bei einer Eingabe w unterschiedliche Folgen von Konfigurationen.

Betrachten wir die NTM N_1 bei Eingabe $w = 011$. Aus der Startkonfiguration $K_1 = q_0 \rhd 011$ kann die NTM nur in die Konfiguration $K_2 = \rhd q_0 011$ gelangen, da die Menge $\delta(q_0, \rhd)$ nur das Element $(q_0, \rhd, R)$ enthält. $\delta(q_0, 0)$ enthält jetzt zwei Elemente. Wird aus dieser Menge das Element $(q_3, 0, R)$ ausgewählt, so gelangt die NTM N_1 in die Konfiguration $\rhd 0 q_3 11$. Dies ist eine ablehnende Konfiguration und die Berechnung der NTM N_1 ist beendet. Wird jedoch aus der Menge $\delta(q_0, 0)$ das Element $(q_0, 0, R)$ ausgewählt, gelangt N_1 aus der Konfiguration K_2 in die Konfiguration $K_3 = \rhd 0 q_0 11$. Die Menge $\delta(q_0, 1)$ enthält nur das Element $(q_1, 1, R)$, daher gelangt N_1 nun in die Konfiguration $K_4 = \rhd 01 q_1 1$. Die Menge $\delta(q_1, 1)$ enthält nur das Element $(q_2, 1, L)$, daher gelangt N_1 als nächstes in die Konfiguration $K_5 = \rhd 0 q_2 11$. Dieses ist eine akzeptierende Konfiguration und die Berechnung von N_1 stoppt. Bei Eingabe von 011 gibt es sowohl eine Berechnung von N_0, die in einer akzeptierenden Konfiguration endet, als auch eine Berechnung die in einer ablehnenden Konfiguration endet.

Die verschiedenen Berechnungen einer NTM bei Eingabe eines Worts w können gut in einem Baum, dem sogenannten *Berechnungsbaum* von N bei Eingabe w, dargestellt werden. Für die NTM N_1 und die Eingabe $w = 011$ ist der Berechnungsbaum in Abbildung 3.3 dargestellt.

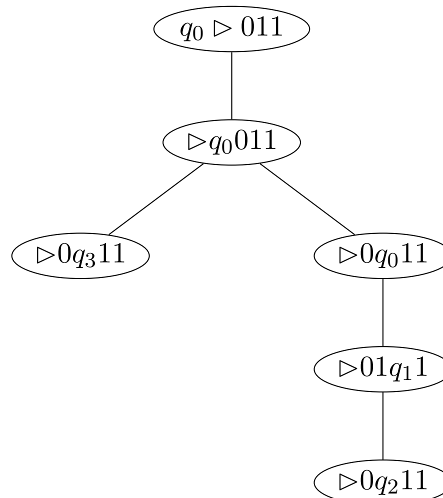


Abbildung 3.3: Berechnungsbaum der NTM N_1 bei Eingabe $w = 011$.

Allgemein sieht ein Berechnungsbaum aus wie in Abbildung 3.4 skizziert. Jeder Knoten des Berechnungsbaums ist mit einer Konfiguration von N gelabelt. Der Wurzelknoten ist mit der Startkonfiguration $q_0 \rhd w$ gelabelt. Ist ein Knoten mit der Konfiguration K gelabelt, so sind die Label seiner Kinderknoten die möglichen direkten Nachfolgekonfigurationen von K. Dies sind diejenigen Konfigurationen, die aus K durch eine einmalige Anwendung der Übergangsfunktion δ erreicht werden können. Ist $K = \alpha q a \beta$ und ist $|\delta(q, a)| = c$, so hat ein Knoten gelabelt

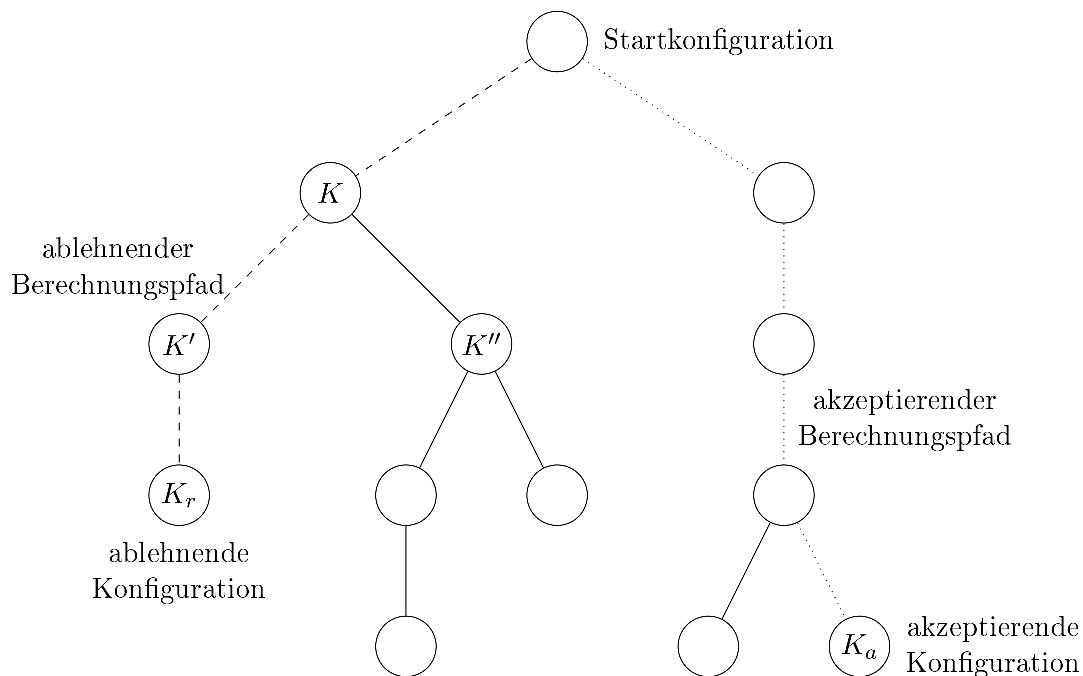


Abbildung 3.4: Berechnungsbaum einer NTM.

mit K genau c viele Kinder. Dabei zählen wir identische Nachfolgekonfigurationen, die durch unterschiedliche Elemente aus $\delta(q, a)$ entstehen, doppelt.

Einen Pfad im Berechnungsbaum nennen wir *Berechnungspfad*. Jeder Berechnungspfad entspricht einer möglichen Berechnung von N bei Eingabe w. Da es Berechnungen gibt, die zu Endlosschleifen führen, kann der Berechnungsbaum unendlich groß sein und es kann unendlich lange Berechnungspfade geben. Berechnungspfade endlicher Länge entsprechen Berechnungen, die irgendwann im Zustand q_{accept} oder im Zustand q_{reject} enden. Endet eine Berechnung im Zustand q_{accept}, so nennen wir die Berechnung und den entsprechenden Berechnungspfad *akzeptierend*. Endet die Berechnung im Zustand q_{reject}, so heißt die Berechnung und der Berechnungspfad *ablehnend*. In Abbildung 3.4 ist der gestrichelte Pfad ablehnend, während der gepunktete Pfad akzeptierend ist.

Wir werden in Zukunft stets annehmen, dass $|\delta(q, a)| \leq 2$, d.h., eine NTM hat zu jedem Zeitpunkt höchstens zwei Wahlmöglichkeiten. Gilt $|\delta(q, a)| = 1$, befindet sich die NTM im Zustand q und liest das Symbol a, so verhält sich die NTM N wie eine deterministische Turingmaschine, sie hat keinerlei Wahlmöglichkeiten. Man kann sich leicht überlegen, dass die Einschränkung auf $|\delta(q, a)| \leq 2$ NTMs in ihrer Rechenkraft nicht einschränkt. Jede NTM, die dieser Einschränkung nicht unterliegt, kann durch eine NTM mit dieser Einschränkung simuliert werden. Die Forderung $|\delta(q, a)| \leq 2$ für alle $q \in Q, a \in \Gamma$ entspricht der Tatsache, dass jeder Knoten in einem Berechnungsbaum von N höchstens zwei Kinderknoten besitzt.

Als nächstes wollen wir definieren, wann eine NTM N ein Wort $w \in \Sigma^*$ akzeptiert. Wie wir im Beispiel der NTM N_1 gesehen haben, kann eine NTM bei einem Wort w sowohl akzeptierende als auch ablehnende Berechnungen haben. Die folgende Definition mag etwas überraschend erscheinen, aber sie hat sich durch die auf sie aufbauende Theorie als sehr nützlich erwiesen.

Definition 3.11 Sei $N = (\mathbb{Q}, \Sigma, \Gamma, \delta)$ eine NTM. Die NTM N *akzeptiert* ein Wort $w \in \Sigma^*$, wenn es *mindestens eine* akzeptierende Berechnung von N bei Eingabe w gibt.

Anders formuliert, die NTM N akzeptiert w, wenn der Berechnungsbaum von N bei Eingabe w einen akzeptierenden Berechnungspfad enthält. Aufbauend auf dieser Definition können wir

nun die von einer NTM akzeptierte bzw. entschiedene Sprache definieren. Wir sagen dabei, dass die NTM N bei jeder Eingabe hält, wenn für alle Eingaben alle möglichen Berechnungen im Zustand q_{accept} oder im Zustand q_{reject} enden. Mit anderen Worten, die NTM N hält bei jeder Eingabe, wenn die Berechnungsbäume für alle Eingaben w nur endliche Pfade besitzen.

Definition 3.12 Die von einer NTM $N = (Q, \Sigma, \Gamma, \delta)$ *akzeptierte Sprache* $L(N)$ ist definiert als

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

Wir sagen, die NTM *akzeptiert die Sprache* L, wenn $L = L(N)$. Die NTM N *entscheidet* die Sprache L, wenn N immer hält und N die Sprache L akzeptiert.

Das Verhältnis von Akzeptieren und Entscheiden bei einer NTM ist vollkommen analog zum Verhältnis von Akzeptieren und Entscheiden bei einer DTM (siehe Definition 2.3).

Bei einer NTM N herrscht eine deutliche Asymmetrie zwischen den Worten $w \in L(N)$ und den Worten $w \notin L(N)$. Ein Wort w wird von einer NTM N akzeptiert oder liegt in $L(N)$, wenn die NTM N *mindestens einen* akzeptierenden Berechnungspfad bei Eingabe w besitzt. Ein Wort w liegt nicht in $L(N)$, wenn *kein* Berechnungspfad von N bei Eingabe w ein akzeptierender Berechnungspfad ist.

Betrachten wir als Beispiel die NTM N_1 (siehe die Übergangsfunktion in Abbildung 3.2). Wie wir gesehen haben, besitzt N_1 bei Eingabe $w = 011$ einen akzeptierende Berechnungspfad. Daher gilt $011 \in L(N_1)$. Man kann sich nun leicht überlegen, dass N_1 bei Eingabe $w \in \{0,1\}^*$ nur dann einen akzeptierenden Berechnungspfad besitzen kann, wenn w die Teilfolge 11 enthält. Wir erhalten somit, dass die von N_1 akzeptierte (und auch entschiedene) Sprache gegeben ist durch

$$L(N_1) := \big\{w \in \{0,1\}^* \mid w \text{ enthält die Teilfolge 11.}\big\}.$$

NTMs sind aufgrund ihrer Wahlmöglichkeiten kein realistisches Rechenmodell. Sie bieten jedoch eine Alternative zur Definition der Klasse NP, die zu einer sehr interessanten Theorie führt und uns Probleme wie das Rucksackproblem RS_{ent} besser verstehen lässt. Wir wollen daher als nächstes eine NTM betrachten, die RS_{ent} entscheidet. Die NTM erhält als Eingabe die Kodierung einer Instanz der Sprache RS_{ent}.

N bei Eingabe $\langle G, W, g, w\rangle$:

1. Erzeuge nichtdeterministisch ein Wort $c \in \{0,1\}^n$. c repräsentiert die Binärkodierung einer Teilmenge $S \subseteq \{1, \ldots, n\}$.
2. Teste, ob $\sum_{i \in S} g_i \leq g$ und $\sum_{i \in S} w_i \geq w$. Falls ja, akzeptiere. Sonst lehne ab.

Im 1. Schritt wird die Kodierung einer beliebigen Teilmenge $S \subseteq \{1, \ldots, n\}$ erzeugt. Dieses sicherzustellen ist kein Problem, da es hier ausreicht, nichtdeterministisch mithilfe eines Zählers n-mal entweder eine 0 oder eine 1 auf das Band zu schreiben. Der zweite Schritt garantiert dann, dass es nur für Worte $\langle G, W, g, w\rangle \in \text{RS}_{\text{ent}}$ akzeptierende Berechnungen gibt. Damit ist $L(N) = \text{RS}_{\text{ent}}$. Man beachte auch, dass der 2. Schritt dieser NTM deterministisch ist, d.h., in diesem Schritt hat N keine Wahlmöglichkeiten.

Als nächstes wollen wir die Laufzeit einer NTM definieren. Wir beschränken uns dabei wie bei DTMs auf NTMs, die bei jeder Eingabe halten. Wie bei einer DTM besteht bei einer NTM ein Rechenschritt aus der einmaligen Anwendung der Übergangsfunktion. Befindet sich die NTM im Zustand q und liest das Zeichen a, so wählt die NTM in einem Rechenschritt ein Element

aus $\delta(q, a)$ aus und führt die Aktionen entsprechend dem ausgewählten Element aus $\delta(q, a)$ aus.

Definition 3.13 Sei $N = (Q, \Sigma, \Gamma, \delta)$ eine nichtdeterministische Turingmaschine, die bei jeder Eingabe hält.

- Für $w \in \Sigma^*$ ist $\mathrm{T}_N(w)$ definiert als die maximale Anzahl von Rechenschritten in einer möglichen Berechnung von N bei Eingabe w.
- Für $n \in \mathbb{N}$ ist
$$\mathrm{T}_N(n) := \max\{\mathrm{T}_N(w) \mid w \in \Sigma^{\leq n}\}.$$
- Die Funktion $\mathrm{T}_N : \mathbb{N} \to \mathbb{N}$ heißt die *Zeitkomplexität* oder *Laufzeit* der NTM N.

Nach dieser Definition ist $\mathrm{T}_N(w)$ genau die Länge eines längsten Pfades im Berechnungsbaum von N bei Eingabe w.
Bis auf den ersten Punkt entspricht diese Definition genau der Definition der Laufzeit einer DTM. Für die Anzahl der Rechenschritte einer NTM bei Eingabe w gibt es unterschiedliche Definitionen. Einige alternative Definitionen können auch in den Büchern, die wir empfohlen haben, gefunden werden. Im Buch von Wegener ist z.B. die Laufzeit einer NTM im wesentlichen definiert als die Länge eines kürzesten akzeptierenden Rechenwegs. Diese verschiedenen Definitionen der Laufzeit einer NTM sind jedoch, soweit es den Inhalt dieser Vorlesung betrifft, äquivalent.
Schließlich definieren wir noch die Klassen NTIME$(t(n))$ analog zu den Zeitklassen DTIME$(t(n))$.

Definition 3.14 Sei $t : \mathbb{N} \to \mathbb{N}$ eine monoton wachsende Funktion. Die Klasse NTIME$(t(n))$ ist definiert als

$$\mathrm{NTIME}(t(n)) := \left\{ L \;\middle|\; \begin{array}{l} L \text{ ist eine Sprache, die von einer NTM mit Laufzeit} \\ \mathcal{O}(t(n)) \text{ entschieden wird.} \end{array} \right\}.$$

Nun kommen wir zur angekündigten alternativen Definition der Klasse NP. Wir sagen, dass eine NTM N *polynomielle Laufzeit* hat, wenn es ein $k \in \mathbb{N}$ gibt, so dass $\mathrm{T}_N(n) = \mathcal{O}(n^k)$.

Satz 3.15 NP *ist die Klasse aller Sprachen, die von einer nichtdeterministischen Turingmaschine mit polynomieller Laufzeit entschieden werden.*

Beweis: Zum Beweis des Satzes müssen wir zeigen

1. Gibt es für eine Sprache L einen polynomiellen Verifizierer, dann gibt es auch eine NTM N mit polynomieller Laufzeit, die L entscheidet.
2. Gibt es für eine Sprache L eine NTM N mit polynomieller Laufzeit, die L entscheidet, dann gibt es auch einen polynomiellen Verifizierer für L.

Zunächst zu 1.: Gegeben sei also ein polynomieller Verifizierer V für die Sprache L. Sei weiter die Laufzeit von V beschränkt durch n^k, $k \in \mathbb{N}$. Wir konstruieren aus V eine NTM, die L entscheidet, wie folgt.
N bei Eingabe w der Länge n:

1. Erzeuge nichtdeterministisch ein Wort c der Länge höchstens n^k.
2. Simuliere V mit Eingabe $\langle w, c\rangle$.
3. Wenn V die Eingabe $\langle w, c\rangle$ akzeptiert, akzeptiere.
   Sonst lehne ab.

Da die Laufzeit von V polynomiell ist, ist die Laufzeit von N ebenfalls polynomiell. Außerdem wird $\langle w \rangle$ genau dann von N akzeptiert, wenn es ein Zertifikat c gibt, so dass V die Eingabe $\langle w, c \rangle$ akzeptiert. Damit entscheidet N die Sprache L.

Nun zu 2.: Gegeben sei also eine NTM N, die die Sprache L entscheidet. Wir konstruieren dann einen Verifizierer V wie folgt.

V bei Eingabe $\langle w, c \rangle$:

1. Interpretiere $\langle c \rangle$ als die Kodierung eines möglichen Berechnungspfades im Berechnungsbaum von N bei Eingabe w.
2. Simuliere die Berechnung, die diesem Berechnungspfad entspricht.
3. Ist die Berechnung von N akzeptierend, akzeptiere. Sonst lehne ab.

Wir haben nicht gesagt, wie $\langle c \rangle$ als Kodierung eines Berechnungspfades anzusehen ist. Wir können hierzu annehmen $c \in \{0,1\}^*$. Wird eine 0 gelesen, so bedeutet dies, dass wir bei der nächsten Wahl eines Elementes aus einer der Mengen $\delta(q, a)$ das erste aufgeführte Element nehmen sollen. Bei einer gelesenen 1 soll das zweite aufgeführte Element genommen werden. Gilt $|\delta(q, a)| = 1$, so wird bei 0 und 1 jeweils das einzige in $\delta(q, a)$ enthaltenen Element genommen.

Hat N polynomielle Laufzeit, so hat auch V polynomielle Laufzeit. Weiter gibt es nur dann ein c das V zur Akzeptanz von $\langle w, c \rangle$ führt, wenn es für w eine akzeptierende Berechnung bei Eingabe w gibt. Ein Zertifikat für w ist also die Kodierung einer akzeptierenden Berechnung der NTM N bei Eingabe w. Da N polynomielle Laufzeit besitzt, ist die Länge der Kodierung eines Berechnungspfads von N polynomiell. Damit ist V ein polynomieller Verifizierer für L, wenn N die Sprache L entscheidet. □

Das folgende Korollar ergibt sich unmittelbar aus dem letzten Satz und der Definition der nichtdeterministischen Zeitklassen NTIME(n^k).

Korollar 3.16 $\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathit{NTIME}(n^k)$.

Man kann leicht sehen, dass die oben angegebene NTM für $\mathrm{RS}_{\mathrm{ent}}$ polynomielle Laufzeit besitzt. Wir erhalten damit einen neuen Beweis für $\mathrm{RS}_{\mathrm{ent}} \in \mathrm{NP}$. Jetzt noch drei weitere wichtige Beispiele für Sprachen, die in NP liegen.

Beispiel 1 - SAT Eine *Boolesche Variable* x ist eine Variable, die die Werte `wahr` und `falsch` annehmen kann. Im weiteren identifizieren wir `wahr` mit 1 und `falsch` mit 0. Reden wir von *Booleschen Operationen* so meinen wir das logische `oder` $\vee$, das logische `und` $\wedge$ oder das logische `nicht` $\neg$. Statt $\neg x$ schreiben wir in aller Regel $\overline{x}$. Eine *Boolesche Formel* ist ein Ausdruck mit Booleschen Variablen und Booleschen Operationen. Die folgende Formel ist eine Boolesche Formel:

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z}).$$

Eine Boolesche Formel muss natürlich auch korrekt geklammert sein, es dürfen nicht zwei Variablen aufeinanderfolgen, usw. Aber wir verzichten auf eine genaue Definition.

Eine Boolesche Formel ϕ heißt *erfüllbar*, wenn es eine Belegung der Variablen in ϕ mit $0, 1$ gibt, so dass die Formel wahr wird, also mit diesen Belegungen der Wert von ϕ `wahr` oder 1 ist. Wir sagen, die Belegung *erfüllt* die Formel ϕ und nennen die Belegung eine *erfüllende Belegung*. Eine erfüllende Belegung der Formel $\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$ ist $x = 0, y = 1, z = 0$.

Enthält eine Formel ϕ genau n Variablen, so kann ϕ über dem Alphabet $\{0, 1, (,), \wedge, \vee, \neg\}$ codiert werden, indem jeder Variable eine Nummer zwischen 1 und n zugeordnet wird. Eine Variable kann dann durch die Binärdarstellung ihrer Nummer kodiert werden. Nun können wir die Sprache SAT definieren.

$$\text{SAT} := \{\langle\phi\rangle \mid \phi \text{ ist eine erfüllbare Boolesche Formel}\}.$$

Die Formel $\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$ liegt in SAT. Die Formel $\psi = (x \vee y) \wedge \overline{x} \wedge \overline{y}$ hingegen liegt nicht in SAT. Die folgende DTM V ist ein Verifizierer für SAT.

V bei Eingabe $\langle\phi, c\rangle$:

1. Teste, ob $\langle c\rangle$ die Kodierung einer Belegung der Variablen in ϕ ist. Falls nicht, lehne ab.
2. Falls die Belegung ϕ erfüllt, akzeptiere. Sonst lehne ab.

Einen Zeugen c, der V die Eingabe $\langle\phi, c\rangle$ akzeptieren lässt, gibt es nur für erfüllbare Formeln ϕ. Daher ist V ein Verifizierer für SAT.

Um die Laufzeit zu analysieren, bemerken wir zunächst, dass eine Zertifikat für eine erfüllbare Formel in n Variablen durch eine Bitfolge der Länge n codiert werden kann. Damit kann Schritt 1 in Zeit linear in n und damit linear in der Beschreibungsgröße der Formel ausgeführt werden.

In Zeit, die quadratisch in der Beschreibungsgröße einer Formel ist, kann überprüft werden, ob eine gegebene Belegung diese Formel erfüllt. Hierzu wird zunächst in der Formel jede Variable durch ihren Wert in der Belegung ersetzt. Dieses kann selbst auf einer 1-Band DTM in quadratischer Zeit durchgeführt werden. Den Wert der Formel bei der gegebenen Belegung zu berechnen, erfordert dann nur noch lineare Zeit. Damit ist V ein polynomieller Verifizierer für SAT.

Somit ist V ein polynomieller Verifizierer für SAT und SAT liegt in NP.

Hier ist eine NTM, die SAT entscheidet.

N bei Eingabe $\langle\phi\rangle$:

1. Erzeuge nichtdeterministisch eine Belegung der Variablen in ϕ.
2. Falls die Belegung ϕ erfüllt, akzeptiere. Sonst lehne ab.

Der 2. Schritt stellt sicher, dass genau die erfüllbaren Formeln ϕ von N akzeptiert werden. Außerdem ist die Laufzeit von N polynomiell. Die Laufzeit kann nämlich wie die des Verifizierers für SAT analysiert werden. Außerdem beachte man, dass eine Belegung einer Formel in n Variablen nur aus n Bits bestehen muss. Somit ist N eine NTM mit polynomieller Laufzeit, die SAT entscheidet. Auch auf diese Weise sehen wir, dass SAT $\in$ NP.

Beispiel 2 - 3SAT Ein *Literal* ist eine Boolesche Variable x oder die Negation einer Booleschen Variable $\overline{x}$. Eine Klausel ist eine Boolesche Formel bestehend aus mehreren Literalen, die durch $\vee$ miteinander verknüpft sind. So ist $\overline{x_1} \vee x_3 \vee \overline{x_2}$ eine Klausel. Eine Boolesche Formel ϕ ist in *konjunktiver Normalform (KNF)* wenn sie aus mehreren Klauseln verknüpft durch $\wedge$ besteht. Die Formel heißt dann eine KNF-Formel. Damit ist

$$(\overline{x_1} \vee x_3 \vee \overline{x_2}) \wedge (x_5 \vee \overline{x_6}) \wedge (x_7 \vee \overline{x_1} \vee \overline{x_3} \vee x_4)$$

eine KNF-Formel.

Eine *3-KNF Formel* ist eine KNF-Formel in der jede Klausel genau drei Literale enthält. Unser obiges Beispiel für eine KNF-Formel ist somit keine 3-KNF-Formel, wohl aber

$$(\overline{x_1} \vee x_3 \vee \overline{x_2}) \wedge (x_5 \vee \overline{x_6} \vee x_4) \wedge (x_7 \vee \overline{x_1} \vee \overline{x_3}).$$

Nun ist

$$\text{3SAT} := \{\langle \phi \rangle \mid \phi \text{ ist eine erfüllbare 3-KNF Formel}\}.$$

Die Sprache 3SAT ist eine echte Teilmenge der Sprache SAT. Die Sprache 3SAT liegt auch in NP, wie durch die folgende NTM gezeigt wird.

N_{3SAT} bei Eingabe $\langle \phi \rangle$:

1. Falls ϕ keine 3-KNF-Formel ist, lehne ab.
2. Erzeuge nichtdeterministisch eine Belegung der Variablen in $\langle \phi \rangle$.
3. Falls die Belegung ϕ erfüllt, akzeptiere. Sonst lehne ab.

Die NTM $\bar{N}$ entscheidet 3SAT. Weiter ist die Laufzeit von $\bar{N}$ polynomiell. Denn zusätzlich zu den Schritten, die wir bereits bei der NTM für SAT benutzt haben, muss die NTM $\bar{N}$ nur noch überprüfen, ob die Eingabeformel ein 3-KNF Formel ist. Dieses kann aber durch ein einmaliges Durchlaufen der Beschreibung der Formel überprüft werden. Damit gilt 3SAT $\in$ NP.

Beispiel 3 - Clique Sei $G = (V, E)$ ein ungerichteter Graph. Eine Teilmenge C der Knoten von G heißt *Clique*, wenn für alle $i, j \in C$ gilt $\{i, j\} \in E$. Die Knoten einer Clique sind also alle paarweise durch Kanten im Graphen G verbunden. Eine Clique C heißt k-Clique, wenn C genau k Knoten enthält. Nun setzen wir

$$\text{Clique} := \{\langle G, k \rangle \mid G \text{ ist ein ungerichteter Graph mit einer } k\text{-Clique}\}.$$

Auch Clique liegt in NP, wie die folgende NTM zeigt.

N bei Eingabe $\langle G, k \rangle$:

1. Erzeuge nichtdeterministisch eine Teilmenge C der Größe k der Knoten von G.
2. Bilden die Knoten in C eine Clique, akzeptiere. Sonst lehne ab.

Von dieser NTM ist leicht zu zeigen, dass sie Clique entscheidet und polynomielle Laufzeit besitzt.
Zum Abschluss dieses Abschnitts wollen wir uns überlegen, wie gut eine NTM durch eine DTM simuliert werden kann. Aus Korollar 3.16 kann gefolgert werden, dass P = NP gelten würde, wenn jede NTM N mit Laufzeit $t(n)$ durch eine DTM M mit Laufzeit $t(n)^k$ simuliert werden kann, wobei k eine beliebige aber feste natürliche Zahl ist. Leider ist die beste bekannte Simulation aber im wesentlichen die Simulation, die im Beweis des nächsten Satzes angegeben wird.

Satz 3.17 *Sei $t : \mathbb{N} \to N$ eine monoton wachsende Funktion mit $t(n) \geq n$ für alle n. Für jede NTM mit Laufzeit $t(n)$ gibt es eine DTM mit Laufzeit $2^{\mathcal{O}(t(n))}$, die dieselbe Sprache entscheidet.*

Beweis: Eine Konfiguration der NTM N heißt akzeptierend, wenn in dieser Konfiguration der Zustand der Maschine q_{accept} ist. Eine Konfiguration mit Zustand q_{reject} heißt ablehnend. Sei N eine NTM mit Laufzeit $t(n)$. Die NTM N entscheide die Sprache $L \subseteq \Sigma^*$. Um eine DTM

D zu erhalten, die ebenfalls L entscheidet, gehen wir folgendermaßen vor. Bei Eingabe $w \in \Sigma^*$ führt D eine Suche (im wesentlichen eine Breitensuche) auf dem Berechnungsbaum $B(w)$ von N bei Eingabe w aus. Die Suche stoppt und die DTM M akzeptiert, sobald ein Knoten gefunden wird, der mit einer akzeptierenden Konfiguration gelabelt ist. Wird bei der Suche kein Knoten gefunden, der mit einer akzeptierenden Konfiguration gelabelt ist, so wird D das Wort w ablehnen. Da N Laufzeit $t(n)$ hat, kann ein Berechnungspfad im Berechnungsbaum von N höchstens Länge $t(n)$ haben. Daher kann bei der Suche leicht sichergestellt werden, dass jeder Knoten im Berechnungsbaum $B(w)$ mindestens einmal besucht wird. Dann ist nach der Definition von $L(N)$ die von D entschiedene Sprache $L(M)$ gerade $L(N)$.

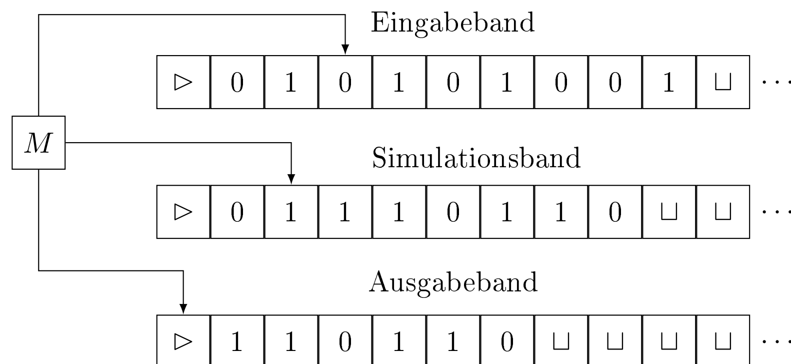


Abbildung 3.5: DTM D, die eine NTM N simuliert.

Um diese Idee umzusetzen, benutzen wir eine 3-Band DTM D. Auf dem ersten Band der DTM steht die Eingabe. Das zweite Band wird zur Simulation einzelner Schritte der NTM N benutzt. Das dritte Band, genannt Adressband, wird benutzt, um die Suche der Maschine D zu steuern (siehe Abbildung 3.5). Dieses geschieht folgendermaßen. Auf dem dritten Band steht immer ein Wort $c \in \{0,1\}^{t(n)}$, wobei n die Länge des Eingabewortes w ist. Das Wort c legt fest, welche mögliche Berechnung von N simuliert werden soll. Ist das aktuell gelesene Zeichen von c eine 0, befindet sich die NTM nach den bislang simulierten Schritten im Zustand q und liest das Symbol a, so wird im nächsten Schritt der erste in $\delta(q,a)$ aufgeführte Übergang durch die DTM D auf ihrem zweiten Band simuliert. Ist das aktuell gelesene Zeichen in c eine 1, so wird der zweite Übergang aus $\delta(q,a)$ simuliert. Ist $|\delta(q,a)| = 1$, so wird unabhängig vom aktuell in c gelesenen Symbol, der einzige in $\delta(q,a)$ angegebene Übergang ausgeführt. Nach der Simulation eines Schrittes wird der Kopf des 3. Bandes immer um eine Zelle nach rechts bewegt und das nächste Symbol in c gelesen. Insgesamt arbeitet D wie folgt.

1. Schreibe $c = 0^{t(n)}$ auf das 3. Band (Adressband).
2. Kopiere die Eingabe w vom 1. Band (Eingabeband) auf das 2. Band (Simulationsband)
3. Simuliere die durch c beschriebene Berechnung von N bei Eingabe w. Wird in der Simulation eine akzeptierende Konfiguration erreicht, akzeptiere. Sonst gehe zu 4.
4. Ist $c = 1^{t(n)}$, lehne ab. Sonst ersetze c durch das lexikographisch nächste Wort in $\{0,1\}^{t(n)}$ und gehe zu 3.

Da wir im 3. Schritt alle Worte aus $\{0,1\}^{t(n)}$ erzeugen, wird sichergestellt, dass D alle Knoten des Berechnungsbaums $B(w)$ besucht. Damit gilt $L(N) = L(D)$. Die Laufzeit von D ist beschränkt durch $\mathcal{O}(t(n)2^{t(n)})$. Es gibt $2^{t(n)}$ Worte in $\{0,1\}^{t(n)}$. Jedes dieser Worte führt zur

Simulation einer Berechnung von N für maximal $t(n)$ Schritte. Solch eine Simulation benötigt maximal $\mathcal{O}(t(n))$ Schritte. Damit erhalten wir als Laufzeit $\mathcal{O}(t(n)2^{t(n)}) = 2^{\mathcal{O}(t(n))}$. Die DTM D ist eine Mehrband DTM. Diese können wir durch eine 1-Band DTM simulieren. Für diese erhalten wir dann nach Satz 3.3 eine Laufzeit von $\left(2^{\mathcal{O}(t(n))}\right)^2 = 2^{\mathcal{O}(t(n))}$. □

Zwar zeigt der Satz nicht, dass jede NTM durch eine DTM effizient simuliert werden kann. Aber er zeigt, dass jede NTM überhaupt durch eine DTM simuliert werden kann. Damit zeigt der Satz, dass die Menge der durch eine NTM entschiedenen oder akzeptierten Sprachen mit der Menge der durch eine DTM entschiedenen oder akzeptierten Sprache übereinstimmt. Für den Inhalt der Vorlesung "Berechenbarkeit und Formale Sprache" hätte es also keinen Unterschied gemacht, wenn wir NTMs statt DTMs zur Definition von Entscheidbarkeit und rekursiver Aufzählbarkeit genommen hätten.

3.6 NP-Vollständigkeit

In diesem Abschnitt werden wir den Begriff der NP-*Vollständigkeit* einführen. Nicht alle Probleme, die in NP liegen, sind schwer zu lösen. Insbesondere gilt ja P $\subseteq$ NP. Die NP-vollständigen Probleme sind die schwierigsten Probleme in NP. Es wird nämlich aus der Definition von NP-Vollständigkeit folgen, dass alle Sprachen in NP durch DTMs polynomieller Laufzeit entschieden werden können, wenn nur für irgendeine beliebige NP-vollständige Sprache eine polynomielle DTM existiert, die die Sprache entscheidet.

3.6.1 Polynomielle Reduktionen

Reduktionen hatten wir schon in Kapitel 2 als ein Mittel kennengelernt, um die Komplexität von Sprachen oder Problemen miteinander vergleichen zu können. In der Theorie der Berechenbarkeit wollten wir aber nur eine grobe Unterteilung in entscheidbare und nicht entscheidbare Sprachen treffen. Jetzt wollen wir eine feinere Unterteilung der Komplexität vornehmen. Daher müssen wir Reduktionen zu polynomiellen Reduktionen verfeinern. [2] Hierzu zunächst die folgende Definition.

Definition 3.18 Sei Σ ein Alphabet. Eine Funktion $f : \Sigma^* \to \Sigma^*$ heißt *polynomiell berechenbar*, wenn es eine DTM M mit polynomieller Laufzeit gibt, die f berechnet. Dabei berechnet M die Funktion f, wenn für alle $w \in \Sigma^*$ die DTM M bei Eingabe w im Zustand q_{accept} hält, der Kopf auf der ersten Zelle steht und der Inhalt des Bandes $f(w)$ ist (hierbei werden das Startsymbol $\triangleright$ und die $\sqcup$s am Ende ignoriert).

Eine wichtige Eigenschaft polynomieller berechenbarer Funktionen ist die Abgeschlossenheit bei Komposition. Genauer

Lemma 3.19 *Seien $f : \Sigma^* \to \Gamma^*, g : \Gamma^* \to \Xi^*$ polynomiell berechenbare Funktionen. Dann ist auch die Hintereinanderschaltung $g \circ f : \Sigma^* \to \Xi^*$ polynomiell berechenbar.*

Beweis: Seien M_f, M_g DTMs mit polynomieller Laufzeit, die die Funktionen f und g berechnen. Die Laufzeit von M_f sei n^k und die Laufzeit von M_g sei n^l. Um $g(f(w)), w \in \Sigma^*$, zu berechnen, wird mit M_f zunächst $f(w)$ berechnet. Danach wird M_g auf $f(w)$ angewandt, um $g(f(w))$ zu berechnen. Die Berechnung von $f(w)$ durch M_f benötigt Zeit $|w|^k$. Gegeben $f(w)$ benötigt die Berechnung von $g(f(w))$ durch M_g Zeit $|f(w)|^l$. Da M_f Laufzeit n^k besitzt, muss $|f(w)| \leq |w|^k$ gelten. Damit gilt $|f(w)|^l \leq |w|^{kl}$. Insgesamt erhalten wir, dass $g(f(w))$ in Zeit $n^k + n^{kl} = \mathcal{O}(n^{kl})$ berechnet werden kann. Damit ist $g \circ f$ polynomiell berechenbar. □

Die folgende Definition ist grundlegender für den Rest dieses Kapitels.

Definition 3.20 Seien $A, B \subseteq \Sigma^*$ zwei Sprachen. A heißt auf B *polynomiell reduzierbar*, wenn es eine polynomiell berechenbare Funktion f gibt mit

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

Die Funktion f wird in diesem Fall eine polynomielle Reduktion genannt. Ist A auf B polynomiell reduzierbar, so schreiben wir $A \leq_p B$.

[2] Zur Wiederholung von Reduktionen siehe Definition 2.34 und zur Veranschaulichung Abbildung 2.16 auf Seite 39.

Bei einer polynomiellen Reduktion verlangen wir also zusätzlich zu den Eigenschaften einer Reduktion, dass die Funktion f effizient berechenbar ist. Dieses stellt sicher, dass die Entscheidung, ob $w \in A$, sogar effizient auf die Entscheidung, ob $f(w) \in B$, zurückgeführt werden kann. Insbesondere erhalten wir

Satz 3.21 *A, B seien zwei Sprachen. Gilt $A \leq_p B$ und $B \in \mathrm{P}$, dann gilt auch $A \in \mathrm{P}$.*

Beweis: Da $B \in \mathrm{P}$, gibt es eine DTM M_B mit polynomieller Laufzeit, die B entscheidet. Weiter sei f die polynomielle Reduktion von A auf B und M_f eine DTM polynomieller Laufzeit, die f berechnet. Wir erhalten dann die folgende DTM M_A für A.

M_A bei Eingabe w :

1. Simuliere M_f, um $f(w)$ zu berechnen.
2. Simuliere M_B mit Eingabe $f(w)$. Akzeptiert M_B die Eingabe $f(w)$, akzeptiere w. Sonst lehne ab.

Da f eine polynomiell berechenbare Funktion ist und auch die DTM M polynomielle Laufzeit hat, folgt analog zum Beweis vo Lemma 3.19, dass D eine DTM mit polynomieller Laufzeit ist. Da $w \in A \Leftrightarrow f(w) \in B$ entscheidet D die Sprache A. □ Polynomielle Reduktionen,

wie wir sie definiert haben, werden gelegentlich auch Karp-Reduktionen (nach Richard Karp) oder many-one-Reduktionen genannt. Der letzte Name erklärt sich aus der Tatsache, dass bei einer Reduktion häufig vielen Urbildern dasselbe Bild zugewiesen wird.
Das nächste Lemma zeigt, dass polynomielle Reduzierbarkeit transitiv ist.

Lemma 3.22 *Ist $A \leq_p B$ und $B \leq_p C$, so gilt $A \leq_p C$.*

Beweis: Sei f die polynomielle Reduktion von A auf B und g die polynomielle Reduktion von B auf C. Aus

1. $w \in A \Leftrightarrow f(w) \in B$ für alle w
2. $v \in B \Leftrightarrow g(v) \in C$ für alle v

folgt

$$w \in A \Leftrightarrow (g \circ f)(w) \in C.$$

Damit ist $g \circ f$ eine Reduktion von A auf C. Nach Lemma 3.19 ist $(g \circ f)(w)$ polynomiell berechenbar. Damit ist $g \circ f$ eine polynomielle Reduktion von A auf C. □

Jetzt werden wir ein erstes Beispiel einer polynomiellen Reduktion kennenlernen. Viele weitere Beispiele werden wir im Zusammenhang mit NP-Vollständigkeitsbeweisen durchgehen.

Satz 3.23 *3SAT ist auf Clique polynomiell reduzierbar.*

Beweis: Zu jeder 3-KNF Formel ϕ konstruieren wir einen Graphen G und eine Zahl k, so dass ϕ genau dann erfüllbar ist, wenn G eine Clique der Größe k enthält. Die Reduktion f wird dann die Formel ϕ auf das Paar (G, k) abbilden.
Sei also ϕ eine 3-KNF Formel. Wir schreiben

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k),$$

wobei die a_i, b_i, c_i nicht notwendigerweise verschiedene Literale sind. Diese Formel wird abgebildet auf (G, k), wobei k die Anzahl der Klauseln von ϕ ist und der Graph G folgendermaßen

konstruiert wird. G besitzt $3k$ Knoten. Jeden Knoten von G assoziieren wir mit einem Literal in ϕ. Der Knoten ist dann mit dem Literal, zu dem er assoziiert ist, gelabelt. Die Knoten sind organisiert in k Tripel. Diese assoziieren wir mit den Klauseln in ϕ. Ein Tripel assoziiert zu einer Klausel C enthält genau die drei Knoten, die zu den Literalen der Klausel C assoziiert sind.

Kanten existieren zwischen je zwei Knoten i, j des Graphen G, es sei denn für die beiden Knoten i, j trifft eine der beiden folgenden Bedingungen zu.

1. i, j sind zu Literalen aus derselben Klausel assoziiert.

2. i ist mit dem Literal x und j mit $\overline{x}$ gelabelt oder umgekehrt.

Damit ist die Konstruktion von G abgeschlossen. Ein Beispiel dieser Graphenkonstruktion ist in Abbildung 3.6 dargestellt. Die drei Knoten links sind den Literalen der ersten Klausel assoziiert. Die oberen drei Knoten sind den Literalen der zweiten Klausel assoziiert. Schließlich bilden die Knoten rechts das Tripel für die dritte Klausel. Wir wollen uns nun überlegen, dass

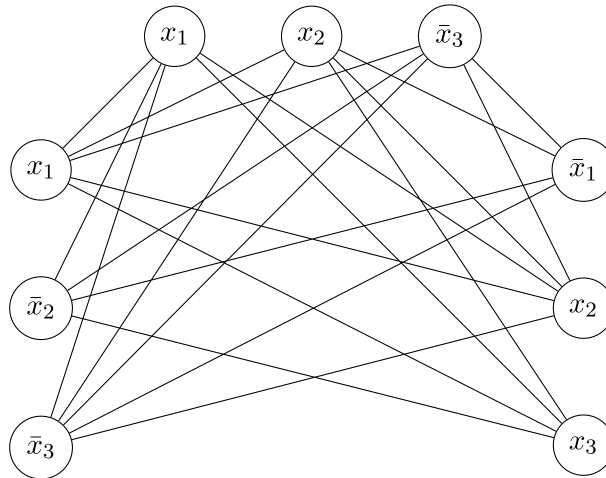


Abbildung 3.6: Graph zu $(x_1 \vee \bar{x_2} \vee \bar{x_3}) \wedge (x_1 \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_1} \vee x_2 \vee x_3)$

eine 3-KNF-Formel genau dann erfüllbar ist, wenn der Graph G, den wir aus ϕ konstruieren, eine k-Clique besitzt. Hierzu müssen zwei Dinge gezeigt werden.

1. Ist ϕ erfüllbar, so besitzt G eine k-Clique.

2. Besitzt G eine k-Clique, so ist ϕ erfüllbar.

Nehmen wir zunächst an, dass ϕ erfüllbar ist. Wir wählen eine beliebige erfüllende Belegung für ϕ aus. Aus dieser werden wir eine Clique in G der Größe k konstruieren.

Eine erfüllende Belegung für ϕ muss in jeder der k Klauseln mindestens ein Literal wahr machen. In jeder Klausel wählen wir nun eines der durch die Belegung wahr gemachten Literale aus. Jetzt betrachten wir die Knoten in G, die zu den ausgewählten Literalen assoziiert sind. Dieses sind genau k Knoten. Wir behaupten, dass diese k Knoten eine Clique bilden. Seien i, j zwei beliebige der k Knoten. Diese Knoten sind nach unserer Auswahl zu Literalen aus unterschiedlichen Klauseln assoziiert. Außerdem können sie nicht mit x und $\overline{x}$ für eine Variable x gelabelt sein. Denn die Knoten sind zu Literalen assoziiert, die gleichzeitig von einer Belegung wahr gemacht werden. Literale $x, \overline{x}$ können natürlich nicht gleichzeitig von einer Belegung erfüllt werden. Damit sind je zwei der k ausgewählten Knoten durch eine Kante in G verbunden und die Knoten zusammen bilden eine k-Clique.

Nehmen wir jetzt an, dass der Graph G eine k-Clique C enthält. Aus dieser konstruieren wir eine erfüllende Belegung für die Formel ϕ. Da Knoten innerhalb eines Tripels nicht durch eine Kante verbunden sind, muss die Clique C aus jedem Tripel genau einen Knoten enthalten. Außerdem gibt es unter den Knoten aus C keine zwei, die mit x und $\overline{x}$ gelabelt sind, denn solche Knoten sind nicht durch eine Kante miteinander verbunden. Damit können wir eine Belegung der Variablen in ϕ finden, die alle Literale erfüllt, mit denen die Knoten in C gelabelt sind. Diese Belegung macht pro Klausel mindestens ein Literal wahr. Die Belegung ist daher eine erfüllende Belegung für ϕ.

Um den Beweis des Satzes zu vervollständigen, müssen wir noch zeigen, dass wir (G, k) aus ϕ in polynomieller Zeit konstruieren können. Dieses ist aber leicht zu sehen. Die Knoten von G und ihre Labels können durch einmaliges Durchlaufen von ϕ konstruiert werden. Um die Kanten zu konstruieren, müssen wir uns für jedes Paar von Knoten nur ihre Labels anschauen und entscheiden, ob sie in unterschiedlichen Tripeln liegen. Auch dieses kann in polynomieller Zeit durchgeführt werden. Damit ist f eine polynomielle Reduktion und der Satz ist bewiesen. □

Streng genommen haben wir im letzten Satz die Reduktion f nicht vollständig beschrieben. Wir haben nicht gesagt, worauf Worte w abgebildet werden, die keiner Kodierung einer 3-KNF Formel entsprechen. Dieses ist auch ziemlich uninteressant und lenkt nur von der eigentlichen Konstruktion ab. Wir werden in Zukunft auf Fragen dieser Art nicht eingehen. Aber hier wollen wir kurz erläutern, was zu tun ist. Wir können in polynomieller Zeit feststellen, ob ein Wort einer Kodierung einer 3-KNF Formel entspricht oder nicht. Entspricht ein Wort einer Kodierung einer 3-KNF Formel, so wenden wir die Konstruktion aus dem Beweis von Satz 3.23 an. Ist das Wort hingegen keine Kodierung einer 3-KNF Formel, so bilden wir das Wort auf ein Paar (G, k) ab, so dass G sicherlich keine k-Clique enthält. Eine mögliche Wahl ist es G bestehend aus einem Knoten zu wählen und $k = 2$ zu setzen.

Die Beweise, die wir zur Äquivalenz $w \in A \Leftrightarrow f(w) \in B$ im Beweis des letzten Satzes geführt haben, sind typisch für alle Beweise dieser Art. Diese Beweise gehen immer nach dem gleichen Schema vor. Betrachten wir die Richtung $w \in A \Rightarrow f(w) \in B$. Nehmen wir an, dass sowohl A als auch B in NP liegen. Für $w \in A$ gibt es dann ein Zertifikat c. Um die Implikation $w \in A \Rightarrow f(w) \in B$ zu zeigen, wird dann aus c mit Hilfe der Funktion f ein Zertifikat für $f(w) \in B$ konstruiert. So haben wir im Beweis von Satz 3.23 aus einem Zertifikat für $\phi \in$ 3SAT (eine erfüllende Belegung) mit Hilfe der Reduktion f ein Zertifikat für $G = f(\phi) \in$ Clique (eine Clique der Größe k) konstruiert. Dieses zeigte, $\langle\phi\rangle \in \text{3SAT} \Rightarrow f(\phi) = \langle G, k\rangle \in \text{Clique}$.

3.6.2 Definition von NP-Vollständigkeit

Die folgende Definition wurde 1971 von Steve Cook aufgestellt und hat seitdem die Entwicklung der Komplexitätstheorie und theoretischen Informatik geprägt.

Definition 3.24 Eine Sprache L heißt NP-*vollständig*, wenn sie die folgenden zwei Bedingungen erfüllt.

1. L ist in NP enthalten.
2. Für jede Sprache $A \in$ NP gibt es eine polynomielle Reduktion von A auf L.

Wir erhalten jetzt die folgenden wichtigen Sätze über NP-vollständige Probleme.

Satz 3.25 *Ist* L NP-*vollständig und ist* $L \in$ P, *so gilt* P = NP.

Beweis: Nach Voraussetzung gibt es für jede Sprache A eine polynomielle Reduktion auf L. Weiter liegt L nach Voraussetzung in P. Dann aber sagt Satz 3.21, dass auch jede Sprache $A \in \mathrm{NP}$ in P liegen muss. □

Satz 3.26 *Ist $A \in \mathrm{NP}$ und gilt $L \leq_p A$ für eine Sprache L, die NP-vollständig ist, so ist auch A NP-vollständig.*

Beweis: Da A nach Voraussetzung in NP liegt, müssen wir nur noch zeigen, dass sich jede Sprache $B \in \mathrm{NP}$ in polynomieller Zeit auf A reduzieren lässt. Sei $B \in \mathrm{NP}$ beliebig. Da L NP-vollständig ist, gibt es eine polynomielle Reduktion g von B auf L. Nach Voraussetzung gibt es auch eine polynomielle Reduktion f von L auf A. Dann aber ist die Hintereinanderausführung $f \circ g$ eine polynomielle Reduktion von B auf A. □

Wenn wir ein NP-vollständiges Problem kennen, können wir Satz 3.26 benutzen, um von anderen Sprachen zu zeigen, dass sie NP-vollständig sind. Unser erstes NP-vollständiges Problem wird SAT sein.

3.6.3 Der Satz von Cook-Levin - SAT ist NP-vollständig

Ziel dieses Abschnitts ist es, die NP-Vollständigkeit von SAT zu zeigen. Dieses Ergebnis geht auf Steve Cook und Leonid Levin zurück, die Anfang der siebziger Jahre unabhängig voneinander die Theorie der NP-Vollständigkeit begründeten.

Satz 3.27 *SAT ist* NP-*vollständig.*

Beweis: Wir haben bereits gesehen, dass SAT in NP liegt. Es muss also noch gezeigt werden, dass jede Sprache $L \in$ NP auf SAT reduziert werden kann. Da L in NP liegt, gibt es eine NTM N mit polynomieller Laufzeit, die L entscheidet. Die Reduktion von L auf SAT wird nun bei Eingabe w eine Boolesche Formel ϕ konstruieren, die das Verhalten von N bei Eingabe w simuliert. Ist $w \in L$, so wird eine erfüllende Belegung von ϕ eine akzeptierende Berechnung von N bei Eingabe w kodieren. Ist $w \notin L$, so wird ϕ nicht erfüllbar sein.

Um diese grobe Beweisskizze umzusetzen, werden wir sogenannte *Berechnungstabellen* von N bei Eingabe w benutzen. Nehmen wir an, dass die DTM N die Sprache L in Zeit $n^k - 2$ entscheidet. Der Term -2 hat dabei nur technische Gründe, die in Kürze klar werden. Hat w nun Länge n, so ist eine Berechnungstabelle von N bei Eingabe w eine $n^k \times n^k$ Tabelle, also eine Tabelle bestehend aus n^k Zeilen und n^k Spalten. Die Zeilen und Spalten der Tabelle sind jeweils mit 1 bis n^k durchnummeriert. Den Eintrag in der i-ten Zeile und j-ten Spalte bezeichnen wir mit Eintrag(i, j). Eine solche $n^k \times n^k$ Tabelle heißt *Berechnungstabelle von N bei Eingabe w* wenn die erste Zeile der Tabelle die Startkonfiguration von N bei Eingabe w enthält und jede Zeile $i, i \geq 2$, eine der möglichen Nachfolgekonfigurationen der Konfiguration in Zeile $i-1$ enthält. In Abbildung 3.7 haben wir eine Berechnungstabelle schematisch dargestellt. Das darin hervorgehobene *Fenster* werden wir in Kürze erklären.

q_0	$\triangleright$	w_1	w_2	$\cdots$	w_n	$\sqcup$	$\cdots$	$\sqcup$	$\#$
$\triangleright$	q_0	$\star$	$\star$	$\cdots$				$\star$	$\#$
$\vdots$									$\vdots$
$\vdots$									$\vdots$
$\vdots$									$\vdots$
$\triangleright$			$\star$	$\cdots$				$\star$	$\#$

n^k (Zeilen) · n^k (Spalten)

Abbildung 3.7: Eine $n^k \times n^k$ Berechnungstabelle

Um Tabellen und Berechnungstabellen genauer zu fassen, werden wir wiederum Konfigurationen benutzen (siehe Seite 7). Da wir voraussetzen, dass die NTM N, die die Sprache L entscheidet, Laufzeit $n^k - 2$ besitzt, kann in einer Konfiguration $\alpha q \beta$ von N das Wort $\alpha\beta$ höchstens Länge $n^k - 2$ haben. Zusätzlich wird für den Zustand q einer Konfiguration ein Symbol benötigt. Damit wären wir bei $n^k - 1$ Symbolen, die benötigt werden, um eine Konfiguration darzustellen. Aus technischen Gründen werden wir eine Konfiguration mit dem Symbol # enden lassen. Von nun an werden wir die Konfiguration $\alpha q \beta$ also durch $\alpha q \beta\#$ darstellen. Das Symbol # wird *ausschließlich* am Ende einer Konfiguration benutzt. Damit kann jede Konfiguration von N durch n^k Symbole dargestellt werden. Reservieren wir für jedes der

Symbole in Γ, Q und $\{\#\}$ einen Eintrag in der Berechnungstabelle, so sehen wir, dass jede Konfiguration von N in einer Zeile einer $n^k \times n^k$ Tabelle dargestellt werden kann.

Eine Berechnungstabelle von N bei Eingabe w heißt *akzeptierend*, wenn eine der Zeilen der Tabelle eine akzeptierende Konfiguration ist. Anders formuliert, eine Berechnungstabelle ist akzeptierend, wenn es $1 \leq i, j \leq n^k$ gibt, so dass Eintrag$(i, j) = q_{\text{accept}}$. Damit entspricht eine akzeptierende Berechnungstabelle von N bei Eingabe w einer akzeptierenden Berechnung von N bei Eingabe w. Um zu entscheiden, ob $w \in L$, muss daher entschieden werden, ob es eine akzeptierende Berechnungstabelle von N bei Eingabe w gibt. Die Reduktion von L auf SAT wird nun bei Eingabe w eine Formel ϕ liefern, die genau dann erfüllbar ist, wenn es eine akzeptierende Berechnungstabelle von N bei Eingabe w gibt. Außerdem werden wir aus einer erfüllenden Belegung von ϕ unmittelbar eine akzeptierende Berechnungstabelle konstruieren können.

Um die Variablen von ϕ zu definieren, setzen wir zunächst

$$C := Q \cup \Gamma \cup \{\#\},$$

wobei Q die Zustandsmenge der NTM N und Γ das Bandalphabet der NTM N ist. ϕ ist nun definiert über den Variablen

$$x_{i,j,s}, \quad \text{mit} \quad 1 \leq i, j \leq n^k, s \in C.$$

D.h. für jede mögliche Position (i, j) in einer Berechnungstabelle von N bei Eingabe w und für jedes mögliche Symbol s, das an einer Position der Berechnungstabelle stehen kann, gibt es eine Variable. Wird nun in einer Belegung von ϕ die Variable $x_{i,j,s}$ mit 1 belegt, so wird in der Berechnungstabelle Eintrag$(i, j) = s$ gelten.

Die Formel ϕ setzt sich aus vier Teilformeln $\phi_{\text{Start}}, \phi_{\text{accept}}, \phi_{\text{Eintrag}}, \phi_{\text{move}}$, verbunden durch logische $\wedge$'s, zusammen:

$$\phi = \phi_{\text{Start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{Eintrag}} \wedge \phi_{\text{move}}.$$

Um aus einer erfüllenden Belegung für ϕ eine Berechnungstabelle für N bei Eingabe w ablesen zu können, muss sichergestellt sein, dass jede mögliche Position der Berechnungstabelle mit genau einem Symbol aus C belegt ist. Mit der Interpretation der Variablenbelegungen oben bedeutet dies, für alle Paare (i, j), mit $1 \leq i, j \leq n^k$, muss es in einer erfüllenden Belegung für ϕ genau ein $s \in C$ geben, so dass $x_{i,j,s} = 1$. Setzen wir

$$\phi_{\text{Eintrag}} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right) \right) \right],$$

so ist dieses gewährleistet. Der Teil $\bigvee_{s \in C} x_{i,j,s}$ stellt sicher, dass in einer erfüllenden Belegung für ϕ mindestens eine der Variablen $x_{i,j,s}$ erfüllt ist. Der Teil $\bigwedge_{\substack{s,t \in C \\ s \neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}})$ stellt sicher, dass in einer erfüllenden Belegung für ϕ höchstens eine der Variablen $x_{i,j,s}$ für jedes Paar (i, j) erfüllt ist.

In einer Berechnungstabelle muss die erste Zeile die Startkonfiguration enthalten. Die Formel ϕ_{Start} soll sicherstellen, dass eine erfüllende Belegung von ϕ einer Berechnungstabelle entspricht, in der die erste Zeile die Startkonfiguration der NTM N bei Eingabe w ist. Man überprüft leicht, dass

$$\begin{aligned} \phi_{\text{Start}} \quad = \quad & x_{1,1,q_0} \wedge x_{1,2,\triangleright} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \cdots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#} \end{aligned}$$

das Gewünschte leistet. Dabei haben wir $w = w_1 \cdots w_n$ gesetzt.

In einer akzeptierenden Berechnungstabelle muss in einer der Positionen (i, j) das Symbol q_{accept} stehen. Also muss, wenn einer erfüllenden Belegung von ϕ eine akzeptierende Berechnungstabelle entsprechen soll, sichergestellt sein, dass eine Belegung nur dann ϕ erfüllen kann, wenn mindestens eine der Variablen $x_{i,j,q_{\text{accept}}}$ auf 1 gesetzt wird. Die Teilformel ϕ_{accept} garantiert dieses, indem

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{\text{accept}}}$$

gesetzt wird.

Schließlich muss noch sichergestellt sein, dass eine erfüllende Belegung von ϕ einer Tabelle entspricht, in der jede Zeile eine der möglichen Nachfolgekonfigurationen der Konfiguration in der vorangehenden Zeile kodiert. Anders formuliert, es muss sichergestellt sein, dass eine erfüllende Belegung zu einer Berechnungstabelle von N bei Eingabe w führt. Dieses zu garantieren, ist Aufgabe der Teilformel ϕ_{move}. Hierzu zunächst der Begriff eines Fensters.

Ein *Fenster* ist eine 2×3 Teiltabelle einer Tabelle. Mit dem (i, j)-Fenster bezeichnen wir das Fenster, dessen obere Zeile die Positionen $(i, j-1), (i, j), (i, j+1)$ und dessen untere Zeile die Positionen $(i+1, j-1), (i+1, j), (i+1, j+1)$ der Tabelle umfasst. Eine $n^k \times n^k$ Tabelle besitzt $(n^k - 1) \cdot (n^k - 2)$ viele verschiedene Fenster, denn nur für $1 \le i \le n^k - 1$ und $2 \le j \le n^k - 1$ ist die Definition des (i, j)-Fensters sinnvoll. Ein Fenster, dessen Positionen mit Symbolen aus C belegt sind, heißt *legal*, wenn seine Einträge nicht der Übergangsfunktion der NTM N widersprechen. Anders formuliert, ein Fenster ist legal, wenn es Teil einer Berechnungstabelle von N sein kann. Statt dieses noch genauer zu definieren, hier ein Beispiele zur Veranschaulichung.

Nehmen wir an, die Übergangsfunktion von N hat u.a. folgende Werte

1. $\delta(q_1, a) = \{(q_1, b, R)\}$
2. $\delta(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$

Dann sind die Fenster in der Abbildung 3.8 alle legal.

(a)

a	q_1	b
q_0	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

$\sqcup$	b	a
$\sqcup$	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Abbildung 3.8: Beispiele legaler Fenster

In Fenster (a) kann die zweite Zeile aus der ersten durch Anwendung des ersten Übergangs in 2. erhalten werden. In Fenster (b) kann die zweite Zeile aus der ersten durch Anwendung des zweiten Übergangs in 2. erhalten werden. (c) ist eine legales Fenster, denn die erste Zeile des Fensters lässt es zu, dass der Kopf der NTM N das Symbol a liest. Dann kann die zweite Zeile des Fensters durch Anwendung von 1. erreicht werden. (d) ist legal, da die beiden Zeilen des Fensters identisch sind und kein Zustandssymbol im Fenster auftaucht. (e) ist legal, denn in der Zeile, aus der die erste Zeile dieses Fensters ein Teil ist, könnte unmittelbar rechts vom rechten a das Zustandssymbol q_1 und dann b stehen. Dann erhält man die zweite Zeile von (e) aus der ersten Zeile durch Anwendung des ersten Übergangs in 2. Schließlich ist (f) legal, da die Zeile, aus der die erste Zeile des Fensters ein Teil ist, in der Position, die unmittelbar links

an die erste Zeile des Fensters anschließt, das Zustandssymbol q_1 enthalten könnte. Dann kann man die zweite Zeile des Fensters aus der ersten durch Anwendung des ersten Übergangs in 2. erhalten.

In Abbildung 3.9 sind einige Beispiele für illegale Fenster dargestellt. (a) ist illegal, da ein

a	b	a
a	a	a

(a)

a	q_1	b
q_1	a	a

(b)

a	q_1	b
q_2	b	q_2

(c)

Abbildung 3.9: Beispiele illegaler Fenster

Symbol der oberen Zeile, das nicht zu einem Zustandssymbol benachbart ist, nicht geändert werden darf. (b) ist illegal, denn wenn die NTM N im Zustand q_1 das Symbol b liest, kann sie zwar nach links gehen, muss dann aber b durch c ersetzen und nicht wie in diesem Fenster durch a. (c) enthält zwei Zustandssymbole in einer Zeile und kann daher nicht Teil einer Berechnungstabelle sein, die in jeder Zeile genau ein Zustandssymbol enthält. Es gilt nun

Lemma 3.28 *Ist die erste Zeile einer Tabelle die Startkonfiguration von N bei Eingabe w und ist jedes Fenster der Tabelle legal, so ist jede Zeile der Tabelle eine Nachfolgekonfiguration der Konfiguration der vorangegangenen Zeile und die Tabelle ist eine Berechnungstabelle von N bei Eingabe w.*

Zum Beweis dieses Lemmas betrachten wir zwei aufeinanderfolgende Zeilen der Tabelle und die Konfigurationen in diesen beiden Zeilen. Wir nennen diese die obere und die untere Konfiguration. In der oberen Konfiguration gibt es zu jeder Position (i, j) mit $\text{Eintrag}(i, j) \neq \rhd, \#$, das (i, j)-Fenster. Wir unterscheiden nun zwei Fälle:

1. Sind die beiden Einträge, die zu $\text{Eintrag}(i, j)$ benachbart sind, nicht von Zustandssymbolen belegt, so kann das (i, j)-Fenster nur dann legal sein, wenn $\text{Eintrag}(i, j) = \text{Eintrag}(i + 1, j)$ gilt. Dieses stellt sicher, dass beim Übergang von der oberen zur unteren Konfiguration nur Symbole unmittelbar neben einem Zustandssymbol geändert werden können.

2. Ist $\text{Eintrag}(i, j)$ zu einem Zustandssymbol benachbart, so müssen die Änderungen von der oberen zur unteren Zeile des Fensters den Übergängen der Übergangsfunktion δ entsprechen.

Damit muss aber die untere Konfiguration eine Nachfolgekonfiguration der oberen Konfiguration sein und Lemma 3.28 ist bewiesen.

Nun zurück zur Konstruktion von ϕ_{move}. Nach dem oben Gesagten können wir sicherstellen, dass eine erfüllende Belegung von ϕ einer Berechnungstabelle von N bei Eingabe w entspricht, indem wir informell ϕ_{move} definieren durch

$$\phi_{\text{move}} = \bigwedge_{\substack{1 \leq i \leq n^k - 1 \\ 2 \leq j \leq n^k - 1}} (\text{Das } (i, j)\text{-Fenster ist legal}).$$

Man beachte, dass die Anzahl legaler Fenster nur von der Anzahl der möglichen Übergänge in der Übergangsfunktion von N, nicht jedoch von der Eingabegröße n von w abhängt. D.h. es gibt eine nur von N abhängende Menge von 6-Tupeln $(a_1, a_2, \ldots, a_6) \in C^6$, so dass ein Fenster genau dann legal ist, wenn seine 6 Einträge von links oben nach rechts unten gelesen eines

dieser 6-Tupel sind. Wir identifizieren legale Fenster mit diesen 6-Tupeln. Damit erhalten wir für ϕ_{move}

$$\phi_{\text{move}} = \bigwedge_{\substack{1\le i\le n^k-1\\ 2\le j\le n^k-1}} \bigvee_{\substack{(a_1,\ldots,a_6)\\ \text{ist legales Fenster}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge \ldots \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}).$$

Damit ist nun sichergestellt, dass einer erfüllenden Belegung von $\phi = \phi_{\text{Start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{Eintrag}} \wedge \phi_{\text{move}}$ eine akzeptierende Berechnungstabelle von N bei Eingabe w entspricht. Umgekehrt sieht man auch, dass man aus einer akzeptierenden Berechnungstabelle von N bei Eingabe w auch eine erfüllende Belegung von ϕ erhält. Diese erfüllende Belegung ergibt sich aus

$$x_{i,j,s} = 1 \Leftrightarrow \text{Eintrag}(i,j) \text{ in der Berechnungstabelle ist } s.$$

Damit ist unsere Konstruktion einer Reduktion von L auf SAT vollständig beschrieben. Wir müssen noch zeigen, dass es eine polynomielle Reduktion ist. Dazu müssen wir uns zunächst die Anzahl der Literale in ϕ anschauen. ϕ_{Start} enthält pro Indexpaar ein Literal, insgesamt damit n^k Literale. ϕ_{accept} enthält $n^k \cdot n^k = n^{2k}$ Literale. ϕ_{Eintrag} enthält pro Indexpaar (i,j) genau $|C| + \binom{|C|}{2}$ Literale. Da die Größe $|C|$ der Menge C nur von N nicht jedoch von w abhängt, enthält also ϕ_{Eintrag} $\mathcal{O}(n^{2k})$ Literale. Wir hatten oben schon gesagt, dass die Anzahl legaler Fenster ebenfalls nur von N nicht aber von w abhängt. Damit enthält dann auch ϕ_{move} $\mathcal{O}(n^{2k})$ Literale. Insgesamt enthält ϕ damit $\mathcal{O}(n^{2k})$ Literale. Da k unabhängig von w ist und nur von N abhängt, ist dieses polynomiell in der Größe von w. Damit ist aber auch die Größe von ϕ insgesamt polynomiell in der Größe von w.

Die Konstruktion von ϕ_{Start} ergibt sich unmittelbar aus w. Weiter können ϕ_{accept} und ϕ_{Eintrag} ebenfalls leicht aus der Beschreibung von N konstruiert werden. Es werden hierzu nur das Bandalphabet Γ und die Zustandsmenge Q benötigt. Aus der Übergangsfunktion δ von N können alle legalen Fenster $(a_1, a_2, \ldots, a_6)$ einer Berechnungstabelle von N abgeleitet werden. Dann erhält man die Formel ϕ_{move}. Insgesamt kann ϕ in Zeit polynomiell in der Größe n von w aus der Eingabe w berechnet werden. Damit haben wir eine polynomielle Reduktion von L auf SAT und der Beweis von Satz 3.27 ist erbracht. □

Aus diesem Beweis erhalten wir auch fast unmittelbar

Satz 3.29 *3SAT ist* NP*-vollständig.*

Beweis: Wir wissen bereits, dass 3SAT in NP liegt. Wir müssen noch zeigen, dass sich jede Sprache in NP polynomiell auf 3SAT reduzieren lässt. Hierzu werden wir die Reduktion aus dem Beweis von Satz 3.27 so erweitern, dass sie eine Formel ϕ in 3-KNF liefert.

Zunächst beobachten wir, dass die Teilformeln ϕ_{Start} und ϕ_{accept} bereits KNF-Formeln sind, beide bestehen aus einer einzigen Klausel. Schreiben wir

$$\begin{aligned}\phi_{\text{Eintrag}} &= \bigwedge_{1\le i,j\le n^k} \left[\left(\bigvee_{s\in C} x_{i,j,s}\right) \wedge \left(\bigwedge_{\substack{s,t\in C\\ s\neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}})\right)\right] \\ &= \bigwedge_{1\le i,j\le n^k} \left(\bigvee_{s\in C} x_{i,j,s}\right) \wedge \bigwedge_{1\le i,j\le n^k} \left(\bigwedge_{\substack{s,t\in C\\ s\neq t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}})\right)\end{aligned}$$

so ist auch ϕ_{Eintrag} eine KNF-Formel. Schließlich können wir die Formel

$$\phi_{\text{move}} = \bigwedge_{\substack{1\le i\le n^k-1\\ 2\le j\le n^k-1}} \bigvee_{\substack{(a_1,\ldots,a_6) \text{ ist}\\ \text{legales Fenster}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \ldots \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}).$$

als KNF-Formel schreiben, indem wir die Disjunktion von Konjunktionen als Konjunktion von Disjunktionen schreiben. Dadurch wird die Formel zwar viel größer, aber da die Größe der ursprünglichen Formel nur von der NTM N nicht jedoch von der Eingabe w abhängt, ist dieses unerheblich. Damit können wir die Formel ϕ insgesamt als KNF-Formel schreiben. Diese Umformung erfolgt in polynomieller Zeit.

Wir zeigen nun noch, dass wir jede KNF-Formel ϕ durch eine äquivalente Formel in 3-KNF ersetzen können. Enthalten Klauseln in ϕ weniger als drei Literale, so können wir durch Wiederholen der Literale die Anzahl der Literale in diesen Klauseln auf 3 erhöhen. Jede Klausel mit mehr als drei Variablen werden wir wie folgt durch eine 3-KNF-Formel ersetzen.

Sei $a_1 \vee a_2 \vee \cdots \vee a_l$ eine Klausel in der Formel ϕ mit $l > 3$ Literalen. Wir ersetzen diese Klausel durch die 3-KNF Formel

$$\begin{aligned}(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \\ \cdots \wedge (\overline{z_{l-4}} \vee a_{l-2} \vee z_{l-3}) \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l)\end{aligned} \tag{3.2}$$

wobei die z_i neue Variablen sind, die auch für jede Klausel in ϕ unterschiedlich sind. Man sieht leicht, dass eine Belegung der Variablen in (3.2) diese Formel nur erfüllen kann, wenn mindestens eines der Literale $a_1, \ldots, a_l$ auf 1 gesetzt wird. Daraus folgt, dass ϕ genau dann erfüllbar ist, wenn die neue Formel erfüllbar ist. □

3.7 Beispiele NP-vollständiger Probleme

Zunächst zeigen wir

Satz 3.30 *Clique ist* NP-*vollständig.*

Beweis: Wir wissen bereits, dass Clique in NP liegt. Weiter haben wir auch eine polynomielle Reduktion von 3SAT auf Clique kennengelernt. Damit ist Clique nach Satz 3.29 und Lemma 3.26 NP-vollständig. □

Als nächstes betrachten wir die Sprache Knotenüberdeckung. Eine *Knotenüberdeckung* eines ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $U \subseteq V$, so dass

$$e \cap U \neq \emptyset \quad \text{für alle } e \in E.$$

Ein Graph G *besitzt eine* k*-Knotenüberdeckung*, wenn G eine Knotenüberdeckung U der Größe höchstens k besitzt. Nun ist

$$\text{Knotenüberdeckung} := \{\langle G, k\rangle \mid G \text{ besitzt eine } k\text{-Knotenüberdeckung.}\}$$

Satz 3.31 *Knotenüberdeckung ist* NP-*vollständig.*

Beweis: Knotenüberdeckung ist in NP. Ein Zertifikat für einen Verifizierer besteht bei Eingabe $\langle G, k\rangle$ aus einer Teilmenge U der Knoten von G, die Grösse höchstens k hat.

Nun werden wir zeigen, dass sich 3SAT polynomiell auf Knotenüberdeckung reduzieren lässt. Damit ist dann der Satz nach Satz 3.29 und Lemma 3.26 bewiesen.

Sei also ϕ eine Formel in 3-KNF. Wir konstruieren aus ϕ einen Graphen G und eine Zahl k, so dass

$$\phi \in \text{3SAT} \Leftrightarrow \langle G, k\rangle \in \text{Knotenüberdeckung}.$$

ϕ bestehe aus den Klauseln $C_1, \ldots, C_l$. Die Variablen in ϕ seien $x_1, \ldots, x_n$. Die Literale in Klausel C_j bezeichnen wir mit $l_{j1}, l_{j2}, l_{j3}, j = 1, \ldots, l, l_{ji} \in \{x_1, \ldots, x_n\} \cup \{\overline{x_1}, \ldots, \overline{x_n}\}$.

Die Konstruktion von k aus ϕ ist einfach. Wir wählen $k = n + 2l$.

Nun zur Konstruktion von G. Pro Variable x_i enthält G zunächst einmal zwei Knoten. Einen davon labeln wir mit x_i, den anderen mit $\overline{x_i}$. Die beiden Knoten mit Label x_i und $\overline{x_i}$ sind durch eine Kante verbunden. Die bislang konstruierten Knoten nennen wir *Variablenknoten*. G besitzt also $2n$ Variablenknoten.

G besitzt aber auch noch $3l$ sogenannte *Klauselknoten*. Und zwar enthält G für jede Klausel C_j drei Klauselknoten. Die zu C_j gehörigen Klauselknoten labeln wir mit den Literalen l_{j1}, l_{j2}, l_{j3} von C_j. Insgesamt besitzt G also $2n + 3l$ Knoten.

Die Klauselknoten, die zu einer Klausel gehören, sind durch 3 Kanten paarweise miteinander verbunden. Jeder Klauselknoten ist außerdem mit dem Variablenknoten desselben Labels verbunden. Damit ist die Konstruktion von k und G aus ϕ abgeschlossen. Für die Formel $(x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$ ist die Konstruktion von G in Abbildung 3.10 ausgeführt.

Jetzt zeigen wir, dass ϕ genau dann erfüllbar ist, wenn G eine k-Knotenüberdeckung besitzt. Sei zunächst ϕ erfüllbar. Wir betrachten eine beliebige erfüllende Belegung von ϕ. Ist in dieser Belegung $x_i = 1$, so nehmen wir den Variablenknoten mit Label x_i in die Menge U auf. Setzt die Belegung hingegen $x_i = 0$, so nehmen wir den Variablenknoten mit Label $\overline{x_i}$ in die Menge U auf. Schließlich wählen wir noch aus jeder Klausel ein Literal aus, das durch die Belegung erfüllt wird. Wir nehmen jedoch nicht diese Knoten mit in die Überdeckung auf. Stattdessen

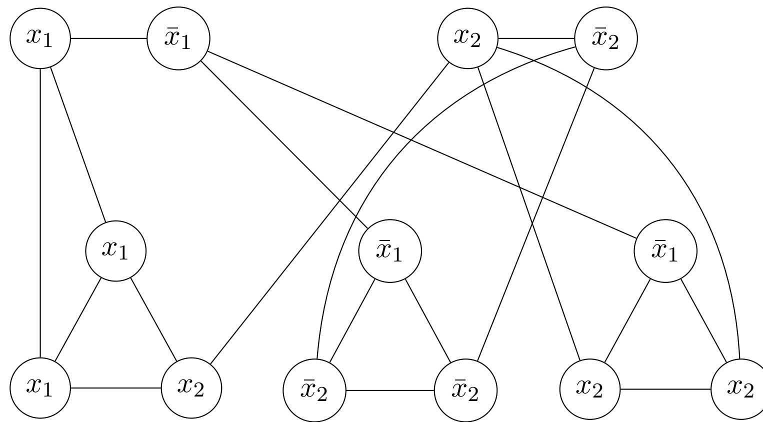


Abbildung 3.10: Beispiel der Reduktion von 3SAT auf Knotenüberdeckung

nehmen wir die beiden anderen Klauselknoten einer Klausel in U auf. Damit enthält U genau $n + 2l$ Knoten. Wir müssen noch zeigen, dass U eine Knotenüberdeckung von G ist.

Die Kanten zwischen den Variablenknoten werden durch die Variablenknoten in U überdeckt. Die drei Kanten zwischen den 3 Klauselknoten einer festen Klausel sind abgedeckt, da U 2 der 3 Klauselknoten enthält. Es fehlen noch die Kanten zwischen einem Klauselknoten und einem Variablenknoten. Ist der Klauselknoten einer solchen Kante in U, ist die Kante sicherlich überdeckt. Ist der Klauselknoten hingegen nicht in U, so wird das Literal dieses Klauselknotens nach Konstruktion von der Belegung erfüllt und der Variablenknoten dieser Kante ist in U. Damit sind alle Kanten überdeckt und U ist eine k-Knotenüberdeckung mit $k = n + 2l$.

Sei nun $\langle G, k\rangle \in$ Knotenüberdeckung, $k = n + 2l$. Wir betrachten eine beliebige k-Knotenüberdeckung U von G. Für jede Variable x_i muss der Variablenknoten gelabelt mit x_i oder der Variablenknoten gelabelt mit $\overline{x_i}$ in U enthalten sein, denn sonst könnte die Kante zwischen diesen beiden Knoten nicht von U überdeckt sein. Wir konstruieren nun eine Belegung für ϕ, indem wir $x_i = 1$ setzen, falls der Variablenknoten mit Label x_i in U ist, sonst setzen wir $x_i = 0$.

Da U eine Knotenüberdeckung ist, muss U von den 3 Klauselknoten einer Klausel C_j mindestens 2 Knoten enthalten, denn sonst können die Kanten zwischen den Klauselknoten zu C_j nicht von U überdeckt werden. Da in U aber nur $n + 2l$ Knoten enthalten sind, folgt, dass für jede Klausel C_j die Menge U *genau* 2 der 3 zu Klausel C_j gehörigen Knoten enthält.

Betrachten wir nun eine feste Klausel C_j. Wir müssen zeigen, dass die gewählte Belegung ein Literal in C_j erfüllt. Nun wissen wir aber, dass es in jeder Klausel C_j ein Literal $l_{jh}, 1 \leq h \leq 3$, gibt, so dass der mit l_{jh} gelabelte Klauselknoten *nicht* in U ist. Betrachten wir weiter die Kante zwischen dem mit l_{jh} gelabelten Klauselknoten und dem Variablenknoten desselben Labels. Auch diese Kante wird von U überdeckt. Da der Klauselknoten nicht in U ist, muss der Variablenknoten mit Label l_{jh} in U sein. Nach Konstruktion erfüllt dann die Belegung das Literal l_{jh}. Somit enthält jede Klausel ein erfülltes Literal und die aus U konstruierte Belegung erfüllt ϕ. □

3.7.1 $\mathbf{RS}_{\text{ent}}$ ist NP-vollständig

Wir wollen zeigen, dass

$$\text{RS}_{\text{ent}} := \left\{ \langle G, W, g, w\rangle \;\middle|\; \begin{array}{l} G = \{g_1, \ldots, g_n\}, W = \{w_1, \ldots, w_n\} \text{ und es existiert ein} \\ S \subseteq \{1, \ldots, n\} \text{ mit } \sum_{i \in S} g_i \leq g \text{ und } \sum_{i \in S} w_i \geq w. \end{array} \right\}$$

NP-vollständig ist. Wir betrachten zunächst die Sprache

$$\text{SubsetSum} := \left\{ \langle S,t\rangle \;\middle|\; \begin{array}{l} S = \{s_1,\ldots,s_n\} \subset \mathbb{N}, t \in \mathbb{N}, \text{ so dass ein } T \subseteq S \text{ existiert} \\ \text{mit } \sum_{s_i \in T} s_i = t. \end{array} \right\}$$

SubsetSum liegt in NP. Für eine Menge S und einen Zielwert t ist ein Zertifikat für $\langle S,t\rangle \in$ SubsetSum gegeben durch eine Teilmenge $T \subseteq S$ mit $\sum_{s_i \in T} s_i = t$.

Lemma 3.32 SubsetSum *ist polynomiell reduzierbar auf* RS_{ent}.

Beweis: Gegeben eine Menge $S = \{s_1,\ldots,s_n\} \subset \mathbb{N}$ und einen Zielwert t, so setzen wir $g_i = w_i = s_i, i = 1,\ldots,n$, $G = \{g_1,\ldots,g_n\}$ und $W = \{w_1,\ldots,w_n\}$. Weiter setzen wir $g = w = t$. Dann gilt

$$\langle S,t\rangle \in \text{SubsetSum} \Leftrightarrow \langle G,W,g,w\rangle \in \text{RS}_{\text{ent}}.$$

Da wir auch G,W,g und w in polynomieller Zeit aus S,t berechnen können, ist das Lemma bewiesen. □

Können wir nun zeigen, dass SubsetSum NP-vollständig ist, so ist auch RS_{ent} NP-vollständig (Lemma 3.26).

Satz 3.33 SubsetSum *ist* NP-*vollständig.*

Beweis: Wir zeigen, dass 3SAT polynomiell auf SubsetSum reduzierbar ist. Da SubsetSum $\in$ NP folgt dann aus Satz 3.29 und aus Lemma 3.26 die Behauptung des Satzes.
Zu einer Klausel ϕ in 3-KNF müssen wir eine Menge $S \subset \mathbb{N}$ und einen Zielwert t konstruieren, so dass

$$\langle \phi\rangle \in \text{3SAT} \Leftrightarrow \langle S,t\rangle \in \text{SubsetSum}.$$

Sei also ϕ eine 3-KNF Formel. ϕ sei definiert über den Variablen $x_1,\ldots,x_n$ und habe die Klauseln $C_1,\ldots,C_k$. Die Menge S wird $2n+2k$ Elemente haben. S enthält für jede Variable $x_i, i = 1,\ldots,n$, zwei Elemente y_i, z_i und für jede Klausel $C_j, j = 1,\ldots,k$, zwei Elemente g_j, h_j. Um die Werte dieser Zahlen und des Zielwerts t festzulegen, betrachten wir eine Tabelle mit $2n+2k+1$ Zeilen. Die Zeilen dieser Tabelle sind mit den y_i, z_i und g_j, h_j sowie mit t gelabelt. Die Tabelle hat $n+k$ Spalten. Diese sind mit den Variablen $x_1,\ldots,x_n$ und den Klauseln $C_1,\ldots,C_k$ gelabelt.
Die Zeilen dieser Tabelle werden die Dezimaldarstellung der Elemente von S sowie des Zielwerts t enthalten. Jeder Spalte, und damit jeder Variablen oder Klausel von ϕ, entspricht also eine Stelle in der Dezimaldarstellung der Elemente in S, bzw. in der Dezimaldarstellung von t. Die Einträge der Tabelle und damit die Dezimaldarstellungen der Zahlen in S und des Zielwerts t werden folgendermaßen bestimmt.

- Die Zeile gelabelt mit $y_i, i = 1,\ldots,n$, enthält in der Spalte gelabelt mit x_i eine 1. Außerdem enthält diese Zeile eine 1 in allen Spalten gelabelt mit Klauseln C_j, in denen das Literal x_i auftaucht. Alle anderen Einträge in der Zeile gelabelt mit y_i sind 0.
- Die Zeile gelabelt mit $z_i, i = 1,\ldots,n$, enthält ebenfalls in der Spalte gelabelt mit x_i eine 1. Weiter enthält diese Zeile eine 1 in allen Spalten gelabelt mit Klauseln C_j, in denen das Literal $\overline{x_i}$ auftaucht. Alle anderen Einträge in der Zeile gelabelt mit z_i sind 0.
- Die Zeilen gelabelt mit g_i und $h_i, i = 1,\ldots,k$, enthalten in der Spalte gelabelt mit C_i eine 1. Alle anderen Einträge in diesen Zeilen sind 0.

- Die Zeile gelabelt mit t enthält in den Spalten gelabelt mit $x_i, i = 1, \ldots, n$, eine 1. In den Spalten gelabelt mit $C_j, j = 1, \ldots, k$, enthält diese Zeile jeweils eine 3.

Betrachten wir die Formel

$$(x_1 \lor \overline{x_2} \lor x_3) \land (x_2 \lor x_3 \lor x_4) \land (\overline{x_3} \lor \overline{x_5} \lor \overline{x_1}),$$

so erhalten wir die folgende Tabelle.

	x_1	x_2	x_3	x_4	x_5	C_1	C_2	C_3
y_1	1	0	0	0	0	1	0	0
z_1	1	0	0	0	0	0	0	1
y_2	0	1	0	0	0	0	1	0
z_2	0	1	0	0	0	1	0	0
y_3	0	0	1	0	0	1	1	0
z_3	0	0	1	0	0	0	0	1
y_4	0	0	0	1	0	0	1	0
z_4	0	0	0	1	0	0	0	0
y_5	0	0	0	0	1	0	0	0
z_5	0	0	0	0	1	0	0	1
g_1	0	0	0	0	0	1	0	0
h_1	0	0	0	0	0	1	0	0
g_2	0	0	0	0	0	0	1	0
h_2	0	0	0	0	0	0	1	0
g_3	0	0	0	0	0	0	0	1
h_3	0	0	0	0	0	0	0	1
t	1	1	1	1	1	3	3	3

(3.3)

Damit ist die Beschreibung der Reduktion beendet. Die Reduktion ist in polynomieller Zeit berechenbar. Wir müssen aber noch zeigen, dass eine Formel ϕ genau dann erfüllbar ist, wenn das aus der Formel konstruierte Paar $\langle S, t \rangle$ in SubsetSum liegt.

Hierzu zunächst einige Vorbemerkungen. Lassen wir die mit t gelabelte Zeile außer Acht, so sehen wir, dass in jeder Spalte einer Tabelle, die aus einer 3-KNF Formel konstruiert wird, höchstens 5 Einsen auftauchen. Daraus können wir schließen, dass wir eine beliebige Teilmenge der Zahlen in S aufsummieren können, ohne dass es bei dieser Summation zu Überträgen kommt. Damit summieren sich die Elemente einer Teilmenge T von S genau dann zu t auf, wenn folgendes gilt:

- In den mit Variablen x_i gelabelten Spalten summieren sich die Einträge, die in den mit den Elementen aus T gelabelten Zeilen stehen, zu 1 auf.
- In den mit Klauseln C_j gelabelten Spalten summieren sich die Einträge, die in den mit den Elementen aus T gelabelten Zeilen stehen, zu 3 auf.

Sei zunächst ϕ erfüllbar. Wir betrachten eine beliebige erfüllende Belegung für ϕ. Aus dieser Belegung werden wir eine Teilmenge $T \subset S$ herleiten, so dass die Elemente in T sich zu t aufsummieren. Setzt die erfüllende Belegung von ϕ $x_i = 1$, so wählen wir $y_i \in T$ und $z_i \notin T$. Setzt hingegen die erfüllende Belegung $x_i = 0$, so wählen wir $z_i \in T$ und $y_i \notin T$. Summieren wir die bis jetzt in T aufgenommenen Elemente auf, so sehen wir, dass sich in den mit den Variablen gelabelten Spalten jeweils eine 1 ergibt. In den mit den Klauseln gelabelten Spalten ergibt sich ein Wert zwischen 1 und 3, je nachdem wie viele Literale von der erfüllende Belegung in einer Klausel erfüllt werden. Um nun in der mit der Klausel C_j gelabelten Spalte den Wert

3 zu erreichen, nehmen wir entweder sowohl g_j als auch h_j in T auf oder wir nehmen nur eine dieser beiden Zeilen in T auf oder aber wir nehmen weder g_j noch h_j in T auf. Wir erhalten so eine Teilmenge T, deren Elemente sich zu t aufsummieren und $\langle S, t\rangle$ ist in SubsetSum.

Sei nun das Paar $\langle S, t\rangle$ in SubsetSum enthalten. Wir müssen zeigen, dass dann die Formel ϕ, aus der $\langle S, t\rangle$ konstruiert wurden, erfüllbar ist. Sei hierzu T eine Teilmenge von S, deren Elemente sich zu t aufsummieren. T muss für jedes $i = 1, \ldots, n$ genau eines der Elemente y_i, z_i enthalten. Hieraus konstruieren wir eine Belegung für ϕ wie folgt. Ist $y_i \in T$, so setzen wir $x_i = 1$. Ist $z_i \in T$, so setzen wir $x_i = 0$. Wir zeigen, dass dieses eine erfüllende Belegung von ϕ ist. Hierzu müssen wir zeigen, dass jede Klausel C_j durch diese Belegung erfüllt wird.

Betrachten wir eine beliebige Klausel C_j von ϕ und die mit C_j gelabelte Spalte unserer Tabelle. In dieser Spalte müssen wir den Wert 3 erreichen. Durch die Zeilen g_j, h_j erreichen wir höchstens den Wert 2. Also muss es noch ein $y_i \in T$ oder ein $z_i \in T$ geben, so dass der Tabelleneintrag in der mit C_j gelabelten Spalte und mit y_i oder z_i gelabelten Zeile eine 1 ist. Nehmen wir nun zunächst an, dass es ein $y_i \in T$ mit dieser Eigenschaft gibt. Da im Eintrag in der mit y_i gelabelten Zeile und der mit C_j gelabelten Spalte eine 1 steht, taucht das Literal x_i in C_j auf. Da $y_i \in T$, wird die Variable x_i in der Belegung, die wir aus T erhalten, auf 1 gesetzt. Damit macht das Literal x_i die Klausel C_j wahr. Ist hingegen ein Eintrag in einer mit $z_i \in T$ gelabelten Zeile und der mit C_j gelabelten Spalte 1, so enthält die Klausel C_j das Literal $\overline{x_i}$. Da $z_i \in T$, gilt $x_i = 0$. Damit macht in diesem Fall das Literal $\overline{x_i}$ die Klausel C_j wahr.

Damit werden alle Klauseln von ϕ durch die aus T gewonnene Belegung erfüllt. ϕ ist also erfüllbar. □

Auch die Sprache TSP_{ent} (siehe Abschnitt 3.5) ist NP-vollständig. Der Beweis hierfür ist allerdings recht aufwendig. Er kann im Buch von Hopcroft, Motwani und Ullman nachgelesen werden.

Satz 3.34 *TSP_{ent} ist* NP-*vollständig.*

Kapitel 4

NP-vollständig - Was nun?

Wurde von einem Problem gezeigt, dass es NP-vollständig ist, ist das Problem damit nicht gelöst oder aus der Welt geschafft. In der Praxis muss es trotzdem gelöst werden. Allerdings muss man bescheidener werden und darf nicht mehr auf Algorithmen hoffen, die in allen Fällen effizient eine korrekte oder optimale Lösung finden. Es gibt im wesentlichen drei Möglichkeiten, mit NP-vollständigen Problemen oder anderen schwierigen Problemen umzugehen oder sie zu umgehen. Diese sind

- Spezialfälle,
- Heuristiken,
- Approximationsalgorithmen.

In dieser Vorlesung behandeln wir nur Spezialfälle und Approximationsalgorithmen.

4.1 Spezialfälle

Trifft man in der Praxis auf ein NP-vollständiges Problem, so sollte man sich als erstes fragen, ob die Problemstellung aus der Praxis wirklich die Lösung des *allgemeinen* Problems verlangt, oder ob man es nicht vielleicht doch nur mit *Spezialfällen* des Problems zu tun hat, die eventuell leicht zu lösen sind.

Beispiel 1: Betrachten wir das Problem Clique und zwar sogar das Problem $Clique_{\mathrm{opt}}$, wo zu gegebenem Graphen G eine möglichst große Clique in G gefunden werden muss. Da schon Clique NP-vollständig ist, können wir keinen effizienten Algorithmus erwarten, der $Clique_{\mathrm{opt}}$ effizient löst.
Nehmen wir nun zusätzlich an, dass wir $Clique_{\mathrm{opt}}$ nur für Bäume lösen sollen, also für zusammenhängende Graphen, die keine Kreise besitzen. In diesem Fall ist $Clique_{\mathrm{opt}}$ einfach, denn die größte Clique in einem Baum kann nur aus einem oder aus zwei Knoten bestehen. Im ersten Fall besitzt der gesamte Baum nur einen Knoten. Hat nämlich ein Graph eine Clique der Größe $k \geq 3$, so besitzt er auch einen Kreis auf dem mindestens 3 Knoten liegen.

Beispiel 2: Betrachten wir die Optimierungsvariante $\text{Unabhängig}_{\mathrm{opt}}$ von Unabhängige Menge. Wiederum nehmen wir an, dass das Problem nur für Bäume gelöst werden muss. Wir nehmen weiter an, dass die Eingaben schon als Bäume mit Wurzel gegeben sind (siehe Abbilddung 4.1). Sei nun T solch ein Baum mit Wurzel r. Wird r in eine unabhängige Menge

für T aufgenommen, so können die Kinder von r nicht mehr in die unabhängige Menge aufgenommen werden. Die größte unabhängige Menge von T, die r enthält, ist $\{r\}$ vereinigt mit den größten unabhängigen Mengen der Teilbäume von T, deren Wurzeln die Enkel von r sind. Nennen wir diese unabhängige Menge U_1. Die größte unabhängige Menge U_2 von T, die r nicht enthält, erhalten wir, indem wir die Vereinigung der größten unabhängigen Mengen der Teilbäume, deren Wurzel die Kinder von r sind, bilden. Vergleichen wir die Größe von U_1 und U_2, so können wir die größte unabhängige Menge in T bestimmen.

Wir können also die Berechnung der größten unabhängigen Menge in einem Baum T auf die Bestimmung von größten unabhängigen Mengen in *Teilbäumen* von T zurückführen. Dieses führt auf einen polynomiellen Algorithmus für $\text{Unabhängig}_{\text{opt}}$ auf Bäume, der die Technik des *Dynamischen Programmierens* benutzt. Wir wollen die Einzelheiten nicht ausführen, erinnern aber daran, dass sie die Dynamische Programmierung in "Datenstrukturen und Algorithmen" bereits kennengelernt haben. Wir werden im Laufe dieser Vorlesung auch noch zweimal auf die Dynamische Programmierung eingehen. In Abbildung 4.1 ist ein Baum mit Wurzel dargestellt. In jeden Knoten v haben wir die Größe der größten unabhängigen Menge des Teilbaums mit Wurzel v geschrieben. Man sieht dann gut, wie sich die Größe für v aus den Größen für die Kinder bzw. Enkel ergibt (Details sind den Lesern überlassen).

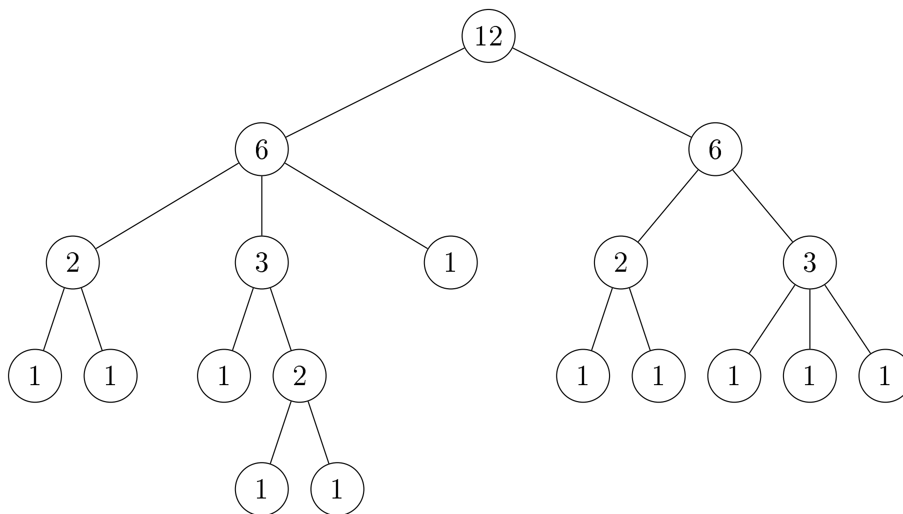


Abbildung 4.1: Unabhängige Menge auf Bäumen

Beispiel 3: Eine Boolesche Formel ist in 2KNF, wenn sie in konjunktiver Normalform ist und jede ihrer Klauseln genau 2 Literale enthält. Die Sprache 2SAT besteht dann aus allen erfüllbaren Formeln in 2-KNF. Wir wollen zeigen, dass 2SAT in P liegt.

Betrachten wir eine 2KNF Formel ϕ. Es sei

$$\phi = \bigwedge_{i=1}^{k} C_i,$$

wobei jede Klausel C_i die Disjunktion zweier Literale ist. Die Menge der Variablen, über denen ϕ definiert ist, sei $x_1, \ldots, x_n$. Sei B eine Belegung der Variablen $x_1, \ldots, x_n$. Wir schreiben $B(x_i) = 1$, falls die Belegung B die Variable x_i auf wahr setzt, sonst schreiben wir $B(x_i) = 0$. Allgemeiner schreiben wir $B(\alpha) = 1$, wenn B das Literal α erfüllt. Sonst schreiben wir $B(\alpha) = 0$. Eine Belegung B der Variablen $x_1, \ldots, x_n$ erfüllt ϕ genau dann, wenn B jede einzelne Klausel C_i erfüllt. Betrachten wir nun eine Klausel C_i. Wir schreiben $C_i = \alpha_i \vee \beta_i$,

wobei α_i, β_i Literale sind. Ist nun $B(\alpha_i) = 0$, so kann B die Klausel C_i dann und nur dann erfüllen, wenn $B(\beta_i) = 1$. Diese Abhängigkeiten zwischen den Wahrheitswerten von Literalen, kann elegant durch einen gerichteten Graphen G_ϕ ausgedrückt werden.
Wir setzen $G_\phi = (V_\phi, E_\phi)$. Dabei gilt

$$V_\phi := \{x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n\}.$$

Für jedes mögliche Literal in ϕ enthält G_ϕ also einen Knoten. Weiter gilt

$$E_\phi := \{(\alpha, \beta) \mid \phi \text{ enthält die Klausel } \bar{\alpha} \vee \beta \text{ oder die Klausel } \beta \vee \bar{\alpha}.\}$$

Der Graph G_ϕ drückt genau die oben beschriebenen Abhängigkeiten zwischen den Wahrheitswerten der Literale einer erfüllenden Belegung von ϕ aus. Ein Beispiel eines Graphen $G(\phi)$ ist in Abbildung 4.2 dargestellt.

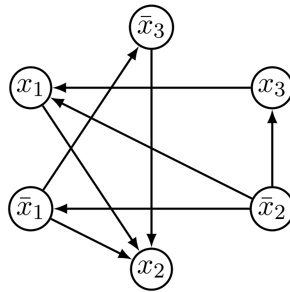


Abbildung 4.2: Der Graph G_ϕ für $\phi = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2) \wedge (x_2 \vee x_3)$

Der folgende Satz führt nun die Frage, ob eine 2KNF Formel ϕ erfüllbar ist, zurück auf einige Erreichbarkeitsprobleme in dem gerichteten Graphen G_ϕ. Da wir bereits wissen, dass Erreichbarkeit in gerichteten Graphen in P liegt (siehe Abschnitt über die Klasse P), wird dieser Satz zeigen, dass die Sprache 2SAT ebenfalls in P liegt.

Satz 4.1 *Sei ϕ eine 2KNF Formel. ϕ ist genau dann nicht erfüllbar, wenn es in ϕ eine Variable x_i gibt, so dass in G_ϕ gerichtete Pfade von x_i nach $\bar{x}_i$ und von $\bar{x}_i$ nach x_i existieren.*

Beweis: Nehmen wir zunächst an, das es eine Variable x_i mit den beiden im Satz genannten Pfaden gibt. Wir zeigen, dass es dann weder eine erfüllende Belegung B von ϕ mit $B(x_i) = 1$ noch eine eine erfüllende Belegung B von ϕ mit $B(\bar{x}_i) = 0$ geben kann. Da aber jede Belegung B der Variablen x_i einen Wahrheitswert zuweisen muss, folgt dann, dass ϕ keine erfüllende Belegung besitzt.
Wir zeigen nur, dass es keine erfüllende Belegung von ϕ mit $B(x_i) = 1$ geben kann. Der andere Fall $B(x_i) = 0$ geht analog. Nach Voraussetzung gibt es in G_ϕ einen gerichteten Pfad von x_i nach $\bar{x}_i$. Der Pfad durchlaufe in dieser Reihenfolge die Knoten $x_i = \alpha_1, \alpha_2, \ldots, \alpha_l = \bar{x}_i$. In G_ϕ existieren damit die Kanten $(\alpha_j, \alpha_{j+1}), j = 1, \ldots, n-1$. Da $B(x_i) = 1$ und $B(\bar{x}_i) = 0$, muss es ein $j, 1 \leq j \leq l-1$ geben mit $B(\alpha_j) = 1, B(\alpha_{j+1}) = 0$. Dann erfüllt B die Klauseln $\bar{\alpha}_j \vee \alpha_{j+1}$ und $\alpha_{j+1} \vee \bar{\alpha}_j$ nicht. Aber nach Konstruktion von G_ϕ ist eine dieser Klauseln eine Klausel in ϕ. Damit erfüllt B die Formel ϕ nicht.
Existiere nun keine Variable x_i, so dass in G_ϕ Pfade von x_i nach $\bar{x}_i$ und von $\bar{x}_i$ nach x_i existieren. Wir zeigen, dass ϕ dann erfüllbar ist. Hierzu geben wir einen Algorithmus an, der eine erfüllende Belegung findet. Eingabe ist immer eine 2KNF Formel ϕ.

2SAT-Belegung

1. Solange es noch eine Variable x_i gibt, der noch kein Wahrheitswert zugewiesen wurde, wiederhole folgende Schritte.

2. Wähle eine beliebige Variable x_i aus, der noch kein Wahrheitswert zugewiesen wurde.

3. Falls in G_ϕ kein Pfad von x_i nach $\bar{x}_i$ existiert, setze den Wahrheitswert von x_i und aller Literale α, so dass der Knoten α in G_ϕ von x_i aus erreichbar ist, auf 1. Die Negationen dieser Literale setze auf 0.

4. Sonst setze den Wahrheitswert aller von x_i aus erreichbaren Literale auf 0. Die Negation dieser Literale setze auf 1.

Aufgrund der Voraussetzung, dass es keine Variable in ϕ gibt, für die in G_ϕ sowohl ein Pfad von x_i nach $\bar{x}_i$ als auch von $\bar{x}_i$ nach x_i existiert, sowie aufgrund der Bedingung in 1. wird jeder Variable in ϕ irgendwann ein Wahrheitswert zugeordnet. Jetzt zeigen wir zunächst, dass einem einmal mit einem Wahrheitswert versehenen Literal im Laufe des Algorithmus 2SAT-Belegung kein neuer Wert zugewiesen wird. Dieses könnte geschehen, wenn wir in 3. oder 4. auf ein Literal treffen, dem bereits ein Wahrheitswert zugewiesen wurde. Wir betrachten nur den Fall, dass dieses in 3. geschieht. Die Argumentation für 4. ist analog.

Nehmen wir an, wir treffen in 3. auf ein Literal γ, dem bereits ein Wahrheitswert zugewiesen wurde. Nehmen wir weiter an, dass die Variable mit der 3. gestartet wurde, die Variable x_s ist. Dann existiert ein Pfad von x_s nach γ. Nun folgt aber aus der Definition der Kantenmenge E_ϕ von G_ϕ, dass in G_ϕ eine Kante (α, β) genau dann existiert, wenn die Kante $(\bar{\beta}, \bar{\alpha})$ existiert. Weiter folgt dann, dass zwischen zwei Knoten α, β in G_ϕ genau dann ein gerichteter Pfad existiert, wenn zwischen den Knoten $\bar{\beta}, \bar{\alpha}$ ein gerichteter Pfad existiert.

Damit ist γ von x_s aus genau dann erreichbar, wenn $\bar{x}_s$ von $\bar{\gamma}$ aus erreichbar ist. Hat nun γ bereits eine Wahrheitswert, dann auch $\bar{\gamma}$. Dann wurde aber auch bereits jedem Knoten, der von $\bar{\gamma}$ aus erreichbar ist, ein Wahrheitswert zugewiesen. Da $\bar{x}_i$ von $\bar{\gamma}$ aus erreichbar ist, wurde damit $\bar{x}_s$ und somit x_s ein Wahrheitswert bereits zugewiesen. x_s hätte damit nicht ausgewählt werden können, um 3. zu starten. Insgesamt schließen wir, dass wir in 3. oder 4. nie auf ein Literal treffen können, dem bereits ein Wahrheitswert zugewiesen wurde.

Bis jetzt haben wir gezeigt, dass 2SAT-Belegung eine Belegung B der Variablen in ϕ berechnet. Wir müssen noch zeigen, dass die Belegung ϕ erfüllt. Nach Konstruktion von 2SAT-Belegung gilt nun, dass B Literalen α, β, die in G_ϕ durch eine Kante verbunden sind, identische Wahrheitswerte zuweist. Ist nun $C_i = \alpha_i \vee \beta_i$ eine Klausel in ϕ, so existiert in G_ϕ die Kante $(\bar{\alpha}_i, \beta_i)$. Damit weist die durch 2SAT-Belegung berechnete Belegung B den Literalen $\bar{\alpha}_i, \beta_i$ identische Wahrheitswerte zu. Damit erfüllt die Belegung B die Klausel C_i. Da dieses für alle Klauseln in ϕ gilt, erfüllt B die Formel ϕ. □

Wie bereits oben angedeutet, können wir aus diesem Satz folgern, dass 2SAT in P liegt.

Korollar 4.2 *Die Sprache 2SAT liegt in P.*

Beweis: Wir wissen, dass Erreichbarkeit in gerichteten Graphen in P liegt. Nun können wir aus einer 2KNF Formel ϕ den Graphen G_ϕ in polynomieller Zeit konstruieren. Ist ϕ über n Variablen definiert, so können wir nach Satz 4.1 durch die Entscheidung von $2n$ Erreichbarkeitsproblemen in gerichteten Graphen entscheiden, ob ϕ erfüllbar ist. Wir müssen also den polynomiellen Algorithmus zur Entscheidung von Erreichbarkeit nur polynomiell häufig aufrufen, um zu entscheiden, ob eine 2KNF Formel entscheidbar ist. Damit liegt 2SAT in P. □

4.2 Approximationsalgorithmen - Notation

Mit einem Approximationsalgorithmus wird ein Optimierungsproblem gelöst. Wir verlangen von einem Approximationsalgorithmus immer, dass er polynomielle Laufzeit besitzt. Ein Approximationsalgorithmus muss nicht eine optimale Lösung einer Instanz eines Optimierungsproblems finden. Bei einem guten Approximationsalgorithmus wird die Güte der gefundenen Lösung allerdings nicht sehr viel schlechter sein als die Güte einer optimalen Lösung. Ein Optimierungsproblem ist definiert durch eine Menge $\mathcal{I}$ von *Instanzen*. TSP_{opt} und RS_{opt} sind Optimierungsprobleme. Bei TSP_{opt} z.B. ist eine Instanz spezifiziert durch die Distanzmatrix Δ. Zu jeder Instanz I haben wir eine Menge $F(I)$ von *zulässigen Lösungen*. Bei TSP_{opt} ist eine zulässige Lösung eine Rundreise, die jede Stadt genau einmal besucht und dann zum Ausgangspunkt zurückkehrt. Weiter haben wir zu jeder zulässigen Lösung $s \in F(I)$ einen Wert $w(s) > 0$. Bei TSP_{opt} ist dieses die Länge der Rundreise s. Das Ziel ist es dann, bei gegebener Instanz eine zulässige Lösung zu finden, so dass $w(s)$ möglichst gross ist (*Maximierungsprobleme*) oder möglichst klein ist (*Minimierungsprobleme*). TSP_{opt} ist ein Minimierungsproblem und RS_{opt} ist ein Maximierungsproblem.

Falls $w(s) \in \mathbb{N}$ für alle zulässigen Lösungen s ist, nennen wir Π ein *diskretes Optimierungsproblem.*

Ein *Approximationsalgorithmus* für ein Optimierungsproblem Π ist ein Algorithmus, der bei Eingabe einer Instanz I von Π eine in $|I|$ polynomielle Laufzeit hat, und eine zulässige Lösung $s \in F(I)$ ausgibt. Hierbei ist $|I|$ die Eingabegröße der Instanz I. Wir schreiben $A(I)$ für die Ausgabe s von A bei Eingabe I.

Bislang scheint diese Definition noch nichts mit Approximationen zu tun zu haben. Darum noch folgende Definitionen. Sei A ein Approximationsalgorithmus für ein Optimierungsproblem Π. Der *Approximationsfaktor* oder die *Approximationsgüte* von A bei Eingabe I ist definiert als

$$\delta_A(I) = \frac{w(A(I))}{\text{opt}(I)},$$

hier ist $\text{opt}(I)$ der Wert einer optimalen zulässigen Lösung für die Instanz I.

Sei $k \in \mathbb{R}^+$. Wir sagen, dass A *Approximationsfaktor* oder *Approximationsgüte* k hat, wenn für jede Instanz I gilt

$$\delta_A(I) \geq k \ \text{ bei einem Maximierungsproblem,}$$

$$\delta_A(I) \leq k \ \text{ bei einem Minimierungsproblem.}$$

Man beachte, dass wir bei einem Maximierungsproblem stets $\delta_A(I) \leq 1$ haben, bei einem Minimierungsproblem dagegen $\delta_A(I) \geq 1$. Analog wird bei Maximierungsproblemen die Zahl k nur Werte kleiner als 1 annehmen, bei Minimierungsproblemen dagegen nur Werte ≥ 1. Man beachte außerdem, dass bei einem Maximierungsproblem ein Approximationsalgorithmus, der Approximationsfaktor k hat, auch Approximationsfaktor k' hat für jede Zahl k' mit $k' \leq k$. Bei Minimierungsproblemen gilt dieses für k', die immer größer sind als k. Wir haben diese Ausdrucksweise gewählt, da bei vielen Approximationsalgorithmen der bestmögliche Approximationsfaktor nicht bekannt ist, sondern nur eine Abschätzung.

4.3 Das Max-Cut Problem

Wir beginnen mit einem einfachen Approximationsalgorithmus für das Problem MaxCut. Bei dem MaxCut-Problem ist eine Instanz ein Graph $G = (V, E)$ mit Knotenmenge V und Kantenmenge E. Gegeben G, so besteht die Menge der zulässigen Lösungen $F(G)$ für die

Instanz G aus allen Teilmengen S der Knotenmenge. Die Funktion w ordnet einer Teilmenge S die Anzahl der Kanten $e \in E$ zu, die einen Endpunkt in S und den anderen Endpunkt in $V \setminus S$ haben. Die Funktion w soll maximiert werden. Eine Teilmenge S der Knotenmenge V eines Graphen werden wir als *Schnitt* bezeichnen. Die Kanten zwischen S und $V \setminus S$ sind die *Schnittkanten*, ihre Anzahl die *Größe* des Schnitts. Beim Problem MaxCut wird also zu gegebenem Graphen G ein maximaler Schnitt gesucht.

Betrachte den Graphen in Abbildung 4.3. Der Schnitt $S = \{1, 4, 5\}$ hat Grösse 5, während $S = \{3, 4\}$ Grösse 6 hat. Dies ist auch gleichzeitig der größtmögliche Schnitt, wie man sich leicht überlegen kann. Die Schnittkanten sind jeweils rot gezeichnet. Der folgende Algorithmus

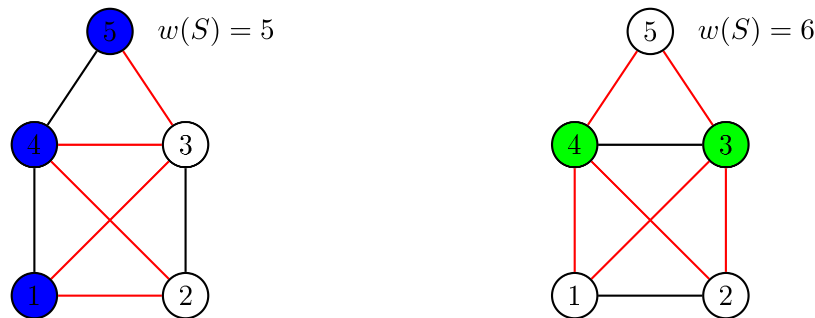


Abbildung 4.3: Schnitte in Graphen

LI (*local improvement*) ist ein Beispiel für die Technik der lokalen Verbesserung, einer Standardtechnik für Approximationsalgorithmen. Man beginnt mit einer beliebigen Menge S (z.B. $S = \emptyset$); solange es noch einen Knoten $v \in V$ gibt, dessen Hinzunahme zu S, bzw. Wegnahme aus S, den aktuellen Schnitt vergrößern würde, wird S entsprechend angepasst. Sobald keine solchen lokalen Verbesserungen mehr möglich sind, wird die aktuelle Menge S als Lösung ausgegeben.

Um den Algorithmus formal schön aufschreiben zu können, benötigen wir noch die Notation der *symmetrischen Differenz* $S \triangle \{v\}$, gegeben durch

$$S \triangle \{v\} := \begin{cases} S \cup \{v\}, & \text{falls } v \notin S \\ S \setminus \{v\}, & \text{andernfalls} \end{cases} .$$

Algorithmus $\mathbf{LI}_{\text{MaxCut}}$

Eingabe: Ein Graph $G = (V, E)$

1. $S := \emptyset, T := V$
2. Solange es ein $v \in V$ gibt, so dass $w(S \triangle \{v\}) > w(S)$
3. Setze $S := S \triangle \{v\}$.
4. Ausgabe S.

Der Algorithmus ist polynomiell in der Größe der Kodierung von G, denn es gibt höchstens $|E|$ Schleifendurchläufe (jedes mal wird der Schnitt um mindestens eine Kante vergrößert). Der Test, ob $w(S \triangle \{v\}) > w(S)$ gilt, geht natürlich auch in polynomieller Zeit für jeden Knoten v.

Satz 4.3 *Algorithmus* $\boldsymbol{LI}_{\text{MaxCut}}$ *hat Approximationsfaktor* $1/2$.

Beweis: Sei (S,T) der von $\mathbf{LI}_{\text{MaxCut}}$ berechnete Schnitt. Dann gilt für jeden Knoten $v \in V$, dass mindestens die Hälfte der zu v inzidenten Kanten Schnittkanten bzgl. S sind. Andernfalls wäre $S \triangle \{v\}$ ein größerer Schnitt als S, was im Widerspruch dazu steht, dass LI die Menge S ausgegeben hat. Dann gilt aber, dass mindestens die Hälfte *aller* Kanten Schnittkanten bzgl. S sind. Das heißt, es gilt

$$\mathbf{LI}_{\text{MaxCut}}(G) = w(S) \geq |E|/2 \geq \text{opt}(G)/2,$$

denn die Gesamtanzahl aller Kanten ist sicherlich eine obere Schranke für die maximale Schnittgröße. Daraus folgt dann

$$\frac{\mathbf{LI}_{\text{MaxCut}}(G)}{\text{opt}(G)} \geq \frac{1}{2},$$

was den Satz beweist, da diese Ungleichung für jeden Graphen (also jede Instanz des Max-Cut Problems) gilt. □

Ein immer wiederkehrendes Muster in den Analysen der Güte von Approximationsalgorithmen kann an diesem Beweis schon gut aufgezeigt werden. Wir wollen die Lösung eines Approximationsalgorithmus mit einem optimalen Wert vergleichen, den wir aber gar nicht kennen! Wir brauchen daher gute Schranken für diesen optimalen Wert. Das Finden solcher Schranken ist häufig der entscheidende Punkt in den Analysen von Approximationsalgorithmen. Um bei einem Maximierungsproblem einen Approximationsfaktor herleiten zu können, benötigen wir eine *obere Schranke* für den Wert einer Optimallösung. Bei Minimierungsproblemen hingegen hilft uns nur eine *untere Schranke* weiter. Die obere Schranke, die wir im Fall von Max-Cut verwendet haben, ist völlig offensichtlich und ergibt sich direkt aus dem Problem, denn es gilt immer $\text{opt}(G) \leq |E|$.

Bevor wir zu weiteren Approximationsalgorithmen kommen, ein kleiner Exkurs über Max-Cut. Können wir nicht vielleicht sogar einen maximalen Schnitt in polynomieller Zeit berechnen? Dieses ist aller Voraussicht nach nicht der Fall. Die Sprache

$$\text{Cut} := \{\langle G, k\rangle \mid G \text{ besitzt einen Schnitt der Größe mindestens } k\}$$

ist NP-vollständig. Wir werden dieses allerdings nicht beweisen. Könnte nun Max-Cut optimal in polynomieller Zeit gelöst werden, so wäre $\text{Cut} \in \text{P}$ und es würde $\text{P} = \text{NP}$ folgen.

4.4 Das Problem des Handlungsreisenden

Wir wollen einen Approximationsalgorithmus für einen wichtigen und interessanten Spezialfall des Problems des Handlungsreisenden beschreiben. Wir nennen dieses Problem ETSP. Gegeben ist wie beim allgemeinen TSP eine Landkarte mit n Städten, die diesmal die Namen $s_1, \ldots, s_n$ tragen. Die Städte s_i entsprechen Punkten in der euklidischen Ebene $\mathbb{R}^2$. Die Entfernung $d_{ij} = d_{ji}$ zwischen den Städten s_i, s_j ist gegeben durch die euklidische Distanz zwischen den Städten s_i, s_j. Die Entfernungen zwischen den Städten sind in einer $n \times n$-Matrix $\Delta = (d_{ij})$ zusammengefasst, wobei $d_{ij} \in \mathbb{N}$, $d_{ii} = 0$ und $d_{ij} = d_{ji}$ gilt. Gesucht ist dann eine kürzeste Rundreise durch alle s_i. Hierbei können wir eine Rundreise auffassen als eine Permutation π der Zahlen $1, \ldots, n$. Die zu π gehörige Rundreise beginnt und endet in $s_{\pi(1)}$ und $s_{\pi(i)}$ ist die i-te Stadt auf der Rundreise.

In der Terminologie, die zu Beginn dieses Abschnitts eingeführt wurde, ist eine Instanz I von ETSP gegeben durch n Punkte $s_i \in \mathbb{R}^2, i = 1, \ldots, n$, und durch die Distanzmatrix Δ, die die euklidischen Abstände zwischen den Punkten s_i enthält. Die Menge der zulässigen Lösungen für die Instanz I ist die Menge der Permutationen der Zahlen $1, \ldots, n$. Es soll dann

die Funktion w minimiert werden, wobei für eine Permutation π die Funktion w definiert ist durch

$$w(\pi) = \left(\sum_{i=1}^{n-1} d_{\pi(i),\pi(i+1)} \right) + d_{\pi(n),\pi(1)}.$$

Gesucht ist ein π, das w minimiert.

Um unseren Approximationsalgorithmus für ETSP zu beschreiben, benötigen wir noch zwei Begriffe aus der Graphentheorie. Aus der Vorlesung "Datenstrukturen und Algorithmen" erinnere man sich, dass ein *Spannbaum* in einem Graphen $G = (V, E)$ ein Baum T auf den Knoten in V ist, so dass jede Kante in T in der Kantenmenge E von G enthalten ist. In einem gewichteten Graphen ist das Gesamtgewicht eines Spannbaums die Summe der Kantengewichte der Kanten des Baumes. Ein *minimaler Spannbaum* ist dann ein Spannbaum mit minimalem Gesamtgewicht. Ein minimaler Spannbaum einer Punktemenge im $\mathbb{R}^2$ ist ein minimaler Spannbaum für den vollständigen Graphen zwischen diesen Punkten, wobei das Gewicht einer Kante durch die euklidische Distanz zwischen den Endpunkten der Kante gegeben ist. Ein minimaler Spannbaum in einem Graphen mit m Kanten kann in Zeit $\mathcal{O}(m \log m)$ Zeit gefunden werden. Der Algorithmus von Kruskal z.B. erreicht diese Laufzeit.

Ein *Euler-Kreis* in einem Graph $G = (V, E)$ ist ein Kreis, der jede Kante genau einmal enthält. Anders als bei Hamiltonschen Kreisen ist es leicht zu entscheiden, ob ein Graph einen Euler-Kreis besitzt. Es gilt nämlich

Lemma 4.4 *Ein Graph $G = (V, E)$ besitzt genau dann einen Euler-Kreis, wenn jeder Knoten in V geraden Grad besitzt. Dabei ist der Grad eines Knotens i die Anzahl der Kanten in G, in denen i enthalten ist.*

Nun können wir den Approximationsalgorithmus für ETSP beschreiben. Der Algorithmus erwartet als Eingabe eine Instanz I von ETSP spezifiziert durch n Punkte $s_1, \ldots, s_n \in \mathbb{R}^2$ und die Distanzmatrix Δ.

Algorithmus MSB

1. Berechne einen minimalen Spannbaum T auf den Punkten $s_1, \ldots, s_n$.
2. Konstruiere aus T den Graphen H, in dem jede Kante in T verdoppelt wird.
3. Finde in H einen Euler-Kreis K.
4. Berechne die Reihenfolge $s_{\pi(1)}, \ldots, s_{\pi(n)}$ der ersten Auftritte der Knoten $s_1, \ldots, s_n$ in K.
5. Gib $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$ aus.

Zu diesem Algorithmus einige Bemerkungen. Eigentlich haben wir bei Graphen keine doppelten Kanten zugelassen. Um den Algorithmus MSB kurz beschreiben zu können, haben wir in der Beschreibung von MSB eine Ausnahme gemacht und lassen beim Graphen H doppelte Kante zu. Da wir H aus T durch Verdoppeln jeder Kante erhalten, hat dann in H jeder Knoten geraden Grad. Damit besitzt H nach Lemma 4.4 einen Eulerkreis. Dieser kann in polynomieller Zeit gefunden werden. Da auch ein minimaler Spannbaum in polynomieller Zeit gefunden werden kann, ist MSBalso ein Approximationsalgorithmus. In Abbildung 4.4 ist das Verhalten von MSB an einem Beipiel mit 8 Städten illustriert. Die Reihenfolge der Städte auf dem Eulerkreis ist

$$s_1, s_5, s_2, s_3, s_2, s_4, s_2, s_6, s_7, s_6, s_8, s_6, s_2, s_5, s_1.$$

Die Reihenfolge der ersten Auftritte ist dann

$$s_1, s_5, s_2, s_3, s_4, s_6, s_7, s_8.$$

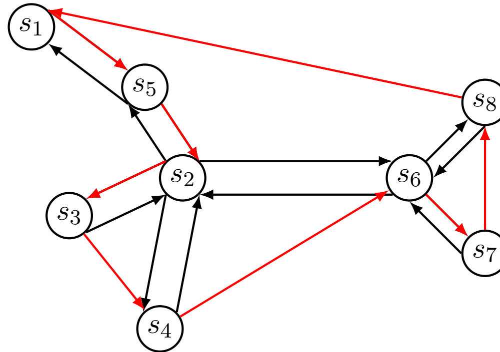


Abbildung 4.4: Beispiel für Berechnung einer Rundreise mit Algorithmus MSB.

Satz 4.5 MSB *hat Approximationsgüte* 2.

Beweis: Sei opt(I) die Länge einer optimalen Rundreise R. Sei $|T|$ das Gewicht des minimalen Spannbaumes T, der in MSB berechnet wird. Durch Entfernen einer beliebigen Kante wird R zu einem Spannbaum. Daher gilt $|T| \leq \text{opt}(I)$. Die Gesamtlänge der Kanten in H ist daher höchstens $2\text{opt}(I)$. Daher hat der Eulerkreis K höchstens Gesamtgewicht $2\text{opt}(I)$. Die Rundreise MSB(I) entsteht aus K, indem "Abkürzungen" genommen werden. Da für die euklidische Distanz in der Ebene die Dreiecksungleichung gilt, erhalten wir $|\text{MSB}(I)| \leq 2\text{opt}(I)$. □

Schränken wir das Entscheidungsproblem TSP_{ent} ein auf Instanzen (Δ, L), wobei Δ wie bei ETSP die Distanzmatrix mit den euklidischen Abständen zwischen Städten in der euklidischen Ebene $\mathbb{R}^2$ ist, erhält man das Entscheidungsproblem ETSP_{ent}. Es kann gezeigt werden, dass auch dieses ein NP-vollständiges Problem ist. Wir können daher keinen polynomiellen Algorithmus erwarten, der ETSP optimal löst.

4.5 Das Rucksackproblem

Eine Instanz des Rucksackproblems RS_{opt} ist definiert durch positive Werte $w_1, \ldots, w_n$ und positive Gewichte $g_1, \ldots, g_n$, sowie eine Schranke g. Zulässige Lösungen sind Teilmengen $I \subseteq \{1, \ldots, n\}$ mit $\sum_{j \in I} g_j \leq g$. Gesucht ist eine zulässige Lösung $I \subseteq \{1, \ldots, n\}$, die $\sum_{j \in I} w_j$ maximiert.

Wir nehmen ohne Beschränkung der Allgemeinheit im Weiteren an, dass $g_j \leq g$ gilt für alle j, denn Gegenstände, die zu schwer für den Rucksack sind, können von vornherein von der Betrachtung ausgenommen werden.

Da RS_{ent} NP-vollständig ist, ist das Auffinden einer optimalen Lösung für das Rucksack-Problem schwer. Trotzdem wollen wir mit einem exakten Algorithmus für das Problem beginnen. Dieser wird später in geeigneter Weise als Baustein für den Approximationsalgorithmus verwendet. Die Idee ist *Dynamisches Programmieren.*

Hierzu definieren wir für $j = 1, \ldots, n$ und i ganzzahlig den Wert

$$F_j(i)$$

als das minimale Gewicht einer Teilmenge der ersten j Gegenstände mit Gesamtwert mindestens i (falls es keine solche Teilmenge gibt, setzen wir $F_j(i)$ auf ∞). $F_n(i)$ gibt also an, wie schwer der Rucksack des Diebes mindestens sein muss, damit die Diebesbeute den Wert i erreicht. Daraus bekommen wir das folgende Lemma.

Lemma 4.6 *Sei opt die optimale Lösung des* Rucksackproblems *(d.h. der Wert einer optimalen Teilmenge von Gegenständen). Dann gilt*

$$opt = \max\{i \mid F_n(i) \leq g\}.$$

Zum Beweis überlegt man sich folgendes: sei i_0 das Maximum. Dann gilt $F_n(i_0) \leq g$, d.h. nach Definition, dass es eine Teilmenge von Gegenständen gibt, die in den Rucksack passt und mindestens Wert i_0 hat. Es gilt also opt $\geq i_0$. Andererseits ist $F_n(i_0 + 1) > g$, d.h. um Werte größer als i_0 zu erreichen, müsste der Dieb seinen Rucksack überladen, was nicht erlaubt ist. Also gilt opt $= i_0$.

Nach der Beobachtung reicht es also aus, das entsprechende Maximum zu berechnen. Dabei hilft folgendes

Lemma 4.7

1. $F_j(i) = 0$ *für* $i \leq 0$.
2. $F_0(i) = \infty$ *für* $i > 0$.
3. $F_j(i) = \min(F_{j-1}(i), g_j + F_{j-1}(i - w_j))$ *für* $i, j > 0$.

Beweis:

1. Wenn der Wert des Diebesgutes nicht positiv sein muss, ist die gewichtsminimale Teilmenge, die dies erreicht, offenbar die leere Menge, und die hat Gewicht 0.

2. Wenn der Wert des Diebesgutes positiv ist, die Auswahl der Gegenstände aber aus der leeren Menge erfolgen soll, so gibt es keine Lösung und $F_0(i) = \infty$ gilt nach Definition.

3. Um eine gewichtsminimale Teilmenge der ersten j Gegenstände mit Wert mindestens i zu finden, kann man wie folgt vorgehen. Man bestimmt einerseits die gewichtsminimale Menge S_1, die den j-ten Gegenstand nicht enthält, andererseits die gewichtsminimale Menge S_2, die ihn enthält. Dann wählt man diejenige mit dem kleineren Gewicht aus. S_1 hat Gewicht $F_{j-1}(i)$, während S_2 Gewicht $g_j + F_{j-1}(i - w_j)$ hat. Denn in diesem Fall muss aus den ersten $j - 1$ Gegenständen nur noch der Wert $i - w_j$ erreicht werden, weil der j-te Gegenstand schon Wert w_j hat. Daraus folgt sofort die Formel.

□

Zum Berechnen von opt werten wir nun solange Funktionswerte von F aus, bis wir das kleinste i mit $F_n(i) > g$ gefunden haben. $i - 1$ ist dann das gesuchte Optimum. Mit Hilfe des Lemmas kann diese Auswertung so geschehen, dass man zur Berechnung des jeweils aktuellen Funktionswerts nur auf bereits berechnete (und in einem zweidimensionalen, mit j und i parametrisierten Feld gespeicherte) Funktionswerte zurückgreifen muss.

Im folgenden Algorithmus nehmen wir an, dass die Werte $F_j(i)$ für $j = 0$ und $i \leq 0$ (siehe Teil (i) und (ii) des Lemmas) bereits bekannt sind.

ExactKnapsack

1. Setze $i := 0$
2. Wiederhole die folgenden Schritte bis $F_n(i) > g$
3. Setze $i := i + 1$
4. Für $j := 1 \ldots, n$
5. Setze $F_j(i) = \min(F_{j-1}(i), g_j + F_{j-1}(i - w_j))$
6. Ausgabe $i - 1$.

Um die Laufzeit des Algorithmus abzuschätzen, beobachten wir, dass es opt + 1 Durchläufe der äußeren Schleife gibt, wobei jeder Durchlauf eine Addition und eine Minimumsberechnung benötigt. Num gilt nach Voraussetzung in jedem Schleifendurchlauf $g_j \leq g$ und $F_{j-1}(i-w_j) \leq g$. Damit benötigt jeder Durchlauf der äußeren Schleife Zeit $O(n \cdot \log(g))$. Insgesamt ergibt sich also

Satz 4.8 *Algorithmus* ExactKnapsack *hat Laufzeit polynomiell in opt und der Eingabegröße.*

Auf den ersten Blick mag dies wie ein polynomieller Algorithmus aussehen. Das ist aber nicht der Fall, denn opt ist im allgemeinen exponentiell in der Eingabegröße. Diese ist von der Größenordnung

$$O\left(\sum_{j=1}^{n}(\log w_j + \log g_j) + \log g\right),$$

denn die Eingabezahlen liegen im Standardmodell binär codiert vor. opt hingegen kann Werte bis zu

$$\sum_{j=1}^{n} w_j$$

annehmen, was exponentiell in der Eingabegröße ist, falls die Gewichte g_j und die Rucksack-kapazität g nicht sehr viel größer als die Werte w_j sind.

Die Idee des folgenden Approximationsalgorithmus ScaledKnapsack ist dann auch, aus dem ursprünglichen Problem ein 'skaliertes' Problem zu konstruieren, bei dem die optimale Lösung so klein ist, dass sie mit Hilfe von ExactKnapsack in polynomieller Zeit bestimmt werden kann. Man muss dann nur noch zeigen, dass daraus dann auch eine vernünftige Lösung für das ursprüngliche Problem konstruiert werden kann.

Der Algorithmus hat außer einer Instanz des Rucksackproblems noch einen Eingabeparameter $\varepsilon > 0$, mit dem die Approximationsgüte kontrolliert wird. Ziel wird es sein, Approximationsgüte $1 - \varepsilon$ zu erreichen.

ScaledKnapsack

1. Setze $w_{max} := \max\{w_j, j = 1, \ldots, n\}$.
2. Wähle $k := \max(1, \lfloor \varepsilon w_{max}/n \rfloor)$
3. Für $j = 1, \ldots n$, setze $w_j(k) := \lfloor w_j/k \rfloor$
4. Berechne $\mathrm{opt}(k)$ und $S(k)$, die optimale Lösung und optimale Teilmenge des Rucksackproblems mit Werten $w_j(k)$ (und unveränderten g_j sowie g) mit Hilfe von ExactKnapsack.
5. Gib $\mathrm{opt}^* := \sum_{j \in S(k)} w_j$ als Lösung aus

Es ist klar, dass die Menge $S(k)$ eine zulässige Lösung auch für das ursprüngliche Problem ist, denn die Gewichte und die Rucksackkapazität haben sich ja nicht verändert.
Nun können wir die Güte von ScaledKnapsack abschätzen.

Satz 4.9 ScaledKnapsack(ε) *hat Approximationsgüte* $1 - \varepsilon$.

Beweis: Sei S die optimale Teilmenge für das ursprüngliche Problem. Dann gilt

$$\begin{aligned} \mathrm{opt}^* &= \sum_{j \in S(k)} w_j \\ &= k \sum_{j \in S(k)} \frac{w_j}{k} \\ &\geq k \sum_{j \in S(k)} \lfloor \frac{w_j}{k} \rfloor \\ &\geq k \sum_{j \in S} \lfloor \frac{w_j}{k} \rfloor, \end{aligned}$$

denn $S(k)$ ist ja optimal (insbesondere also besser als S) bezüglich der skalierten Werte $w_j(k) = \lfloor w_j/k \rfloor$. Es gilt dann weiter

$$\begin{aligned} \mathrm{opt}^* &\geq k \sum_{j \in S} \lfloor \frac{w_j}{k} \rfloor \\ &\geq k \sum_{j \in S} (\frac{w_j}{k} - 1) \\ &= \sum_{j \in S} (w_j - k) \\ &= \mathrm{opt} - k|S| \\ &= \mathrm{opt} \left(1 - \frac{k|S|}{\mathrm{opt}}\right) \geq \mathrm{opt} \left(1 - \frac{kn}{w_{max}}\right), \end{aligned}$$

denn $|S| \leq n$ und $\mathrm{opt} \geq w_{max}$, weil nach der Annahme $g_j \leq g$ für alle j der wertvollste Gegenstand eine untere Schranke für den optimalen Rucksackwert darstellt.
Es bleibt noch zu zeigen, dass aus dieser Herleitung $\mathrm{opt}^* \geq \mathrm{opt}(1 - \varepsilon)$ folgt. Dazu machen wir eine Fallunterscheidung. Falls $k = 1$, so wurde das ursprüngliche Problem gar nicht skaliert, und es gilt sogar $\mathrm{opt}^* = \mathrm{opt}$. Falls $k = \lfloor \varepsilon w_{max}/n \rfloor$, so folgt daraus sofort

$$\frac{kn}{w_{max}} \leq \varepsilon,$$

was die Behauptung impliziert. □

Die gewünschte Approximationsgüte haben wir also erreicht; es bleibt noch zu zeigen, dass der Algorithmus für festes ε auch wirklich polynomielle Laufzeit hat (sonst wäre er gar kein Approximationsalgorithmus in unserem Sinne).

Satz 4.10 ScaledKnapsack(ε) *hat Laufzeit polynomiell in der Eingabegröße und in* $1/\varepsilon$.

Beweis: Der entscheidende Term in der Laufzeit wird der Aufruf von ExactKnapsack für das skalierte Problem sein. Neben diesem Aufruf werden noch die Werte k und $w_j(k) = \lfloor w_j/k \rfloor$ benötigt. Diese Anweisungen zusammen verursachen Kosten die polynomiell in der Eingabegröße und polynomiell in $\log(1/\varepsilon)$ sind.

Die Laufzeit von ExactKnapsack ist nun nach Satz 4.8 polynomiell in der Eingabegröße der skalierten Rucksackinstanz und in $\mathrm{opt}(k)$. Die Eingabegröße der skalierten Rucksackinstanz kann nun nicht größer sein als die der ursprünglichen Instanz. Es bleibt daher noch die Größe von $\mathrm{opt}(k)$ zu bestimmen. Nun $\mathrm{opt}(k)$ kann nicht größer sein als die Summe aller skalierten Werte, insbesondere nicht größer als n-mal der maximale skalierte Wert $\lfloor w_{\max}/k \rfloor$.

Nun zeigen wir, dass $w_{\max}/k < 2n/\varepsilon$ gilt. Hieraus folgt dann die Aussage des Satzes. Um die Abschätzung zu zeigen, beobachten wir zunächst, dass aus der Definition $k := \lfloor \varepsilon w_{\max}/n \rfloor$ die Ungleichung $\varepsilon w_{\max}/n < k + 1$ folgt. Damit gilt weiter

$$\frac{\varepsilon w_{\max}}{kn} < \left(1 + \frac{1}{k}\right)$$

und schließlich

$$\frac{w_{\max}}{k} < \frac{n}{\varepsilon}\left(1 + \frac{1}{k}\right).$$

Da $k \geq 1$ folgt somit, wie gewünscht,

$$\frac{w_{\max}}{k} < \frac{2n}{\varepsilon}.$$

□

Es ist natürlich nicht überraschend, dass die Laufzeit von ε abhängt und mit $\varepsilon \mapsto 0$ beliebig groß wird. Wichtig ist, dass sich für *festes* ε ein polynomieller Algorithmus ergibt – in diesem Fall ein $O(n^3)$-Algorithmus. Die Abhängigkeit von $1/\varepsilon$ ist polynomiell, sie kann aber in einem anderen Fall genausogut exponentiell sein, etwa von der Größenordnung $O(n^{1/\varepsilon})$. Auch dann gilt noch, dass man für festes ε einen polynomiellen Algorithmus hat.

4.6 Ein Unmöglichkeitsergebnis

Nun zeigen wir, dass man für bestimmte Optimierungsprobleme auch nachweisen kann, dass sie hart zu approximieren sind, es sei denn, P = NP. Konkret zeigen wir, dass die Dreiecksungleichung tatsächlich wesentlich ist, um das Problem des Handlungsreisenden gut approximieren zu können.

Satz 4.11 *Wenn es einen polynomiellen Approximationsalgorithmus A mit konstanter Approximationsgüte r für das allgemeine TSP_{opt} Problem gibt, dann ist* P = NP.

Beweis: Wir nehmen an, dass es einen polynomiellen Approximationsalgorithmus A mit Approximationsgüte $r \in \mathbb{N}$ gibt, und formen daraus ein Programm, das das NP-vollständige Hamiltonkreis Problm in Polynomialzeit löst. Was wir also machen werden, ist, das Entscheidungsproblem Hamilton in Polynomzeit auf folgende Variante des $\mathrm{TSP}_{\mathrm{opt}}$, die wir $\mathrm{TSP}[r]$

nennen, zu reduzieren: "Löse TSP_{opt} mit Approximationsgüte r". Also: Hamilton $\leq_p$ TSP$[r]$ für alle $r \in \mathbb{N}$.
Sei ein ungerichteter Graph $G = (V, E)$ gegeben, für den die Frage “Hat G einen Hamiltonkreis?” zu beantworten ist. Wir konstruieren zu G eine Probleminstanz I_G für TSP_{opt}.
Sei $n = |V|$. Wir setzen $I_G = \langle K_n, c\rangle$ mit: K_n ist der vollständige Graph über der Knotenmenge V und

$$c(u,v) = \begin{cases} 1 & \text{falls } \{u,v\} \in E \\ \lceil (r-1)\cdot n\rceil + 2 & \text{falls } \{u,v\} \notin E \end{cases}$$

Sie Kanten mit Gewicht $\lceil (r-1)\cdot n\rceil + 2$ nennen wir *lange* Kanten und die anderen *kurze* Kanten. I_G kann offensichtlich in Polynomzeit aus $\langle G\rangle$ berechnet werden, und es gilt:

(a) Hat G einen Hamiltonkreis, so hat die kürzeste Rundreise in I_G die Länge n, da jeder Kante des Hamiltonkreises in G eine kurze Kante in I_G entspricht.

(b) Hat G keinen Hamiltonkreis, dann hat die kürzeste Rundreise in I_G eine Länge von mindestens

$$\lceil (r-1)\cdot n\rceil + 2 + n - 1 \geq (r-1)\cdot n + n + 1 > r\cdot n\ ,$$

da in jeder Rundtour mindestens eine lange Kante vorkommen muss.

Aus den beiden Beobachtungen folgt, dass I_G grundsätzlich keine zulässige Lösung σ haben kann mit $c(\sigma) \in \{n+1, \dots, \lfloor r\cdot n\rfloor\}$, d.h. eine zulässige Lösung, deren Länge zwischen $n+1$ und $r\cdot n$ liegt. Zwischen diesen beiden Werten haben wir also ein Loch.
Wir benutzen den folgenden Algorithmus, um Hamilton zu lösen. Er verwendet Algorithmus A also Unterprogramm:

Algorithmus ENTSCHEIDE_HAMILTON:

transformiere G in I_G
approximiere mithilfe von A die kürzeste Rundreise in I_G
if $A(I_G) > r\cdot n$
  then gib aus “G hat keinen Hamiltonkreis”
  else gib aus “G hat einen Hamiltonkreis”

Die Korrektheit der if-then-else-Anweisung folgt daraus, dass A eine Approximationsgüte von r garantiert. Wenn also G einen Hamiltonkreis hat, muss $A(I_G) \leq r\cdot n$ sein. Da es aber, wie oben erwähnt, in I_G überhaupt keine Rundreise mit dieser Länge geben kann außer einer der Länge n, muss A diese optimale Rundreise, die einem Hamiltonkreis in G entspricht, gefunden haben. Damit hätten wir aber einen polynomiellen Algorithmus für Hamilton, woraus aufgrund der NP-Vollständigkeit von Hamilton folgen würde, dass P = NP. □

4.7 Approximationsschemata

Wenn sich ein Problem – so wie das Rucksackproblem hier – beliebig gut approximieren lässt, so sagen wir, das Problem besitzt ein Approximationsschema. Formal definieren wir dieses wie folgt.

Definition 4.12 Ein *Approximationsschema A* für ein Optimierungsproblem ist ein Algorithmus, der bei Eingabe ϵ mit $0 < \epsilon < 1$ das gegebene Problem mit Approximationsgüte

$$1 - \epsilon \quad \text{bei Maximierungsproblemen}$$

bzw.

$$1 + \epsilon \quad \text{bei Minimierungsproblemen}$$

lösen kann.

- Ein Approximationsschema A ist ein *polynomielles Approximationsschema* (engl. *polynomial time approximation scheme*, oder *PTAS*), wenn A für jedes *konstante* ϵ eine Laufzeit hat, die polynmiell in der Größe der Instanz ist.
- Ein Approximationsschema A ist ein *streng polynomielles Approximationsschema* (engl. *fully polynomial time approximation scheme*, oder *FPTAS*), wenn A für jedes ϵ eine Laufzeit hat, die polynmiell in der Größe der Instanz *und* $1/\epsilon$ ist.

Beim PTAS sind Laufzeiten der Form $O(|I|^{1/\epsilon})$ für Instanzen I erlaubt, da das immer noch eine polynomielle Laufzeit bei konstantem ϵ ergibt. Für ein FPTAS wäre solch eine Laufzeit aber zu hoch. Eine akzeptable Laufzeit für ein FPTAS wäre z.B. $O((1/\epsilon)^3|I|^2)$. Aufgrund von Satz 4.10 wissen wir, dass das Rucksackproblem ein FPTAS und damit auch ein PTAS besitzt.
Die Angabe von ϵ verlängert die Eingabe für ein FPTAS lediglich um $\Theta(\log(1/\epsilon))$ Bits. Trotzdem erlauben wir, dass die Laufzeit des FPTAS polynomiell in $1/\epsilon$ und nicht polynomiell in $\log(1/\epsilon)$ ist. Würden wir fordern, dass die Laufzeit eines FPTAS $O(poly(|I|, \log(1/\epsilon)))$ ist, würde die Existenz eines solchen FPTAS für ein diskretes Optimierungsproblem Π, dessen Entscheidungsvariante NP-vollständig ist, bedeuten, dass P = NP ist.
Betrachte dafür eine Instanz $I' = \langle I, k\rangle$ der Entscheidungsvariante und setze $\epsilon = 1/2^{|I'|}$. Dann ist offensichtlich $\epsilon < 1/k$. Ist nun $opt(I) \geq k$, dann ist $(1-\epsilon)opt(I) > k-1$ und damit die Ausgabe des FPTAS mindestens k, da wir hier ein *diskretes* Optimierungsproblem betrachten. Ist aber $opt(I) < k$, dann muss offensichtlich auch die Ausgabe des FPTAS kleiner als k sein. Da das FPTAS eine Laufzeit von $O(poly(|I|, \log(1/\epsilon))) = O(poly(|I'|))$ hat, wäre damit $I' = \langle I, k\rangle$ in polynomieller Zeit für den Fall "Ist $opt(I) \geq k$?" entscheidbar. Analoges gilt für den Fall "Ist $opt(I) \geq k$?". D.h. das entsprechende Entscheidungsproblem könnte in polynomieller Zeit gelöst werden, was bedeuten würde, dass P = NP ist. Ist die Entscheidungsvariante bereits für konstante Vergleichswerte (d.h. für Vergleichswerte, die nicht Teil der Eingabe sind) NP-vollständig, folgt daraus der folgende Satz.

Satz 4.13 *Sei* Π *ein diskretes Optimierungsproblem und sei für ein festes* $k \in \mathbb{N}$ *die Entscheidungsvariante "Ist zur Eingabe I von* Π *der Wert* $opt(I) \leq k$*?" falls* Π *ein Minimierungsproblem ist, bzw. "Ist zur Eingabe I von* Π *der Wert* $opt(I) \geq k$*?" falls* Π *ein Maximierungsproblem ist, NP-vollständig. Gibt es ein PTAS für* Π*, dann ist* P = NP.

Beweis: Ein PTAS reicht in diesem Fall, da eine konstante Approximation von $\text{opt}(I) \in \mathbb{N}$ ausreicht um entscheiden zu können, ob $\text{opt}(I) \leq k$ bzw. $\text{opt}(I) \geq k$ ist. □

Offensichtlich gehört das Rucksackproblem nicht zu diesen Optimierungsproblemen. In der Tat reicht es beim Rucksackproblem, wenn die größte Zahl der gegebenen Instanz I, welche wir auch mit $\text{maxnr}(I)$ bezeichnen, polynomiell groß in $|I|$ ist, damit RS_{opt} und damit auch RS_{ent} in polynomieller Zeit gelöst werden kann. Dagegen benötigt das Hamilton Problem keine Gewichte und ist trotzdem NP-vollständig. Man unterscheidet daher zwischen zwei Klassen von NP-vollständigen Problemen.

Definition 4.14 Ein NP-vollständiges Entscheidungsproblem L heißt *stark NP-vollständig*, wenn es ein Polynom q gibt, so dass $L_q = \{I \mid I \in L \text{ und } \text{maxnr}(I) \leq q(|I|)\}$ NP-vollständig ist. Gibt es kein solches Polynom, dann heißt L *schwach NP-vollständig.*

Beispiele:

- Die Entscheidungsprobleme Hamilton und Clique sind stark NP-vollständig, da $q(n) = n$ ausreicht.
- Das Entscheidungsproblem TSP_{ent} ist stark NP-vollständig, da wir im Beweis von Satz 4.11 gesehen haben, dass Hamilton $\leq_p$ TSP$[r]$ für alle $r \in \mathbb{N}$ und damit TSP$[r]$ bereits NP-vollständig ist.
- RS_{ent} ist schwach NP-vollständig, wie wir oben festgestellt haben.

Allgemein besteht eine enge Beziehung zwischen starker NP-Vollständigkeit und der Unmöglichkeit, ein FPTAS anzugeben.

Satz 4.15 *Sei* Π *ein diskretes Optimierungsproblem. Wenn es ein Polynom* $q(x_1, x_2)$ *gibt, so dass für alle Instanzen I gilt, dass* $opt(I) \leq q(|I|, maxnr(I))$ *ist, dann folgt aus der Existenz eines FPTAS für* Π*, dass es einen pseudopolynomiellen exakten Algorithmus (d.h. einen Algorithmus mit Laufzeit* $O(poly(|I|, maxnr(I))))$ *für* Π *gibt.*

Wenn maxnr(I) durch ein Polynom nach oben beschränkt ist, dann ergibt dieser Satz natürlich eine polynomielle Laufzeit für den exakten Algorithmus, was uns unmittelbar zu folgender Aussage führt.

Satz 4.16 *Wenn es für eine diskrete Optimierungsvariante eines stark NP-vollständigen Problems ein FPTAS gibt, dann ist* P = NP.

D.h., dass es z.B. für Hamilton und Clique kein FPTAS geben kann, es sei denn P = NP.