Berechenbarkeit und Komplexität

Skript zur Vorlesung Sommersemester 2012

Eva Richter

13. Juli 2012

Inhaltsverzeichnis

Literatur **3**

1 Vorwort **4**

I Berechenbarkeitstheorie **5**

2 Die Church-Turing-These **6**
2.1 Wiederholung Turingmaschinen 6
2.1.1 Formale Definition 8
2.2 Andere Turingmaschinenvarianten 12
2.2.1 Mehrband-Turingmaschine 12
2.2.2 Nichtdeterministische Turingmaschinen 14
2.2.3 Aufzähler 16
2.3 Äquivalenz zu anderen Modellen 17
2.4 Algorithmen 18
2.4.1 Historisches 19
2.4.2 Church-Turing-These 20
2.5 Sprechen über Turingmaschinen 21
2.5.1 Beschreibung von Turingmaschinen 22
2.5.2 Speichern von Daten in Zuständen 24
2.5.3 Abhaken von Symbolen 25
2.5.4 Verschiebungen 26
2.5.5 Unterprogramme 27

3 Entscheidbarkeit **31**
3.1 Entscheidbare Mengen 31
3.1.1 Probleme regulärer Sprachen 32
3.1.2 Entscheidbare Probleme kontextfreier Sprachen 34
3.2 Das Halteproblem 37
3.2.1 Formulierung des Halteproblems 38
3.2.2 Die Methode der Diagonalisierung 39
3.2.3 Unentscheidbarkeit des Halteproblems 42
3.2.4 Eine nicht-Turing-akzeptierbare Sprache 44

4 Reduzierbarkeit **47**
4.1 Unentscheidbare Probleme der Sprachtheorie 47
4.2 Reduktion über Berechnungshistorie 52
4.3 Das Postsche Korrespondenzproblem 58
4.4 Funktionale Reduzierbarkeit 64

5 Der Rekursionssatz **67**
5.1 Selbstreferenz 67
5.2 Anwendungen 71
5.3 Entscheidbarkeit logischer Theorien 73
5.3.1 Eine entscheidbare Theorie 76
5.3.2 Eine unentscheidbare Theorie 79

II Komplexitätstheorie **83**

6 Grundbegriffe **84**
6.1 Aufgaben und Ziele der Komplexitätstheorie 84
6.2 Komplexitätsmaße und Komplexität 85
6.3 Klein-O und Groß-O Schreibweise 88
6.4 Algorithmenanalyse 90

7 Zusammenhänge zwischen den Komplexitätsklassen **93**
7.1 Komplexitätsbeziehungen zwischen den Berechnungsmodellen 93
7.2 Beschleunigungs- und Kompressionssätze 97
7.3 Hierarchiesätze 100

8 Die Zeitkomplexitätsklassen P und NP **106**
8.1 Die Klasse P 106
8.2 Die Klasse NP 111
8.2.1 Beispiele für NP-Probleme 115
8.2.2 Die P-ungleich- NP-Frage 117

9 NP-Vollständigkeit **117**
9.1 Polynomialzeit – Reduzierbarkeit 118
9.2 Definition von NP-Vollständigkeit 120
9.3 Der Satz von Levin-Cook 122
9.4 Weitere NP-vollständige Probleme 126
9.4.1 Kantenüberdeckung–Vertex-Cover 127
9.4.2 Das Hamilton-Pfad-Problem 128

1 Vorwort

Dieses Vorlesungsskript dient als Begleitung für die Vorlesung zur Berechenbarkeit und Komplexität. Mit einigen wenigen Ausnahmen ist es eine zusammenfassende Darstellung der für die Vorlesung relevanten Inhalte aus dem Buch von Sipser [6] und stimmt in der Notation mit diesem überein. Die Numerierung von Definitionen, Sätzen und Beispielen erfolgt kapitelweise und getrennt voneinander, d.h. Definition 3.2 kann nach Satz 3.5 kommen. Folgerungen werden nicht mitnumeriert, da sie unmittelbar nach den Sätzen auftreten. Auch wenn die Autorin keine Freundin von gemischtsprachigen Texten ist, hat sie sich in einigen Fällen entschieden, englische Begriffe als technische Terme beizubehalten. Das soll es vor allem den Lesern erleichtern, die entsprechenden Passagen im Buch „wiederzuerkennen“.

Teil I
Berechenbarkeitstheorie

2 Die Church-Turing-These

In der Vorlesung über Automaten und Sprachen haben Sie verschiedene Berechnungsmodelle wie DEA und PDA kennengelernt und gesehen, welchen Beschränkungen sie im Bezug auf die akzeptierten Sprachen unterliegen. Sie waren also nicht für allgemeine Berechnungszwecke geeignet. Erst mit der Beschreibung von Turingmaschinen haben Sie ein universelles Berechnungsmodell gesehen. Im ersten Kapitel der Vorlesung über Berechenbarkeit wird es um die sogenannte Church-Turing-These gehen, die im Wesentlichen besagt, dass der Begriff der intuitiven Berechenbarkeit mit dem der Turingberechenbarkeit übereinstimmt. Da die Turingmaschine in der weiteren Vorlesung stets unser Referenzmodell sein wird, werden im Folgenden noch einmal die Definition und einige Varianten von Turingmaschinen wiederholt. Außerdem werden wir ein weiteres universelles Berechenbarkeitsmodell kennenlernen, das besonders im Zusammenhang mit funktionalen Programmiersprachen interessant ist, nämlich den λ-Kalkül.

2.1 Wiederholung Turingmaschinen

Turingmaschinen wurden 1936 von Alan Turing als Gedankenmodell vorgestellt. Eine Turingmaschine kann alles, was ein realer Computer kann. Aber auch für Turingmaschinen gibt es Probleme (ebenso wie für reale Computer), die sie nicht lösen können. Die Grenzen der Berechenbarkeit werden uns besonders im Kapitel über Entscheidbarkeit interessieren. Das Turingmaschinenmodell hat ein rechtsseitig unbeschränktes Band als unendlichen Speicher, es gibt einen Kopf, der sich auf dem Band hin und her bewegen kann und Symbole schreiben und lesen kann. Am Anfang der Berechnung enthält das Band nur den Eingabestring und sonst nur Blank-Symbole bezeichnet durch $\sqcup$. Um ein Symbol zu lesen, muss sich der Kopf der Turingmaschine über die entsprechende Stelle bewegen. Um ein gerade geschriebenes Symbol zu lesen, muss die Maschine den Kopf zurück auf das Symbol bewegen. Die Maschine setzt das Berechnen solange fort, bis sie entscheidet, eine Ausgabe zu machen. Die Ausgaben „*accept*“ und „*reject*“ erhält man, wenn die Maschine in die ausgezeichneten accept- bzw. reject-Zustände wechselt. Falls sie nicht in einen dieser beiden Zustände kommt, dann läuft sie unendlich lange weiter.

Die folgende Liste führt die Unterschiede der Turingmaschinen zu endlichen Automaten auf:

1. Turingmaschine kann vom Band sowohl lesen als auch darauf schreiben,
2. Kopf kann sich in beide Richtungen bewegen,

3. das Band ist unendlich lang,

4. es gibt ausgezeichnete Zustände *accept* und *reject*, die unmittelbare Wirkung haben.

Als Beispiel betrachten wir eine Turingmaschine M_1, die für ein Eingabewort testen soll, ob es in der Sprache $L = \{w\sharp w \mid w \in \{0,1\}^*\}$ enthalten ist. M_1 soll ein Wort akzeptieren, wenn es in L liegt. M_1 wird so gebaut, dass der Kopf sich auf der linken und rechten Seite von $\sharp$ die passenden Symbole markiert und dabei solange über dem Eingabewort hin und her fährt, bis es abgearbeitet ist, oder ein Fehler festgestellt wird.

Beispiel 1. *Turingmaschine M_1 für die Sprache $L = \{w\sharp w \mid w \in \{0,1\}^*\}$*
M_1=„auf einem Eingabewort w:

1. *Überprüfe die Eingabe darauf, ob sie genau ein $\sharp$ enthält, falls nicht,* reject.

2. *Bewege den Kopf zu den einander entsprechenden Positionen links und rechts von $\sharp$ hin und her, um festzustellen ob die Positionen das gleiche Zeichen enthalten. Falls nicht* reject. *Streiche dabei die schon überprüften Symbole um die Abarbeitung zu kontrollieren.*

3. *Wenn alle Symbole links von $\sharp$ ausgestrichen wurden, überprüfe, ob rechts von $\sharp$ noch Symbole verblieben sind, wenn ja* reject*, wenn nein* accept.*“*

Abbildung 1 zeigt einen Blick auf das Band von M_1 während der (wiederholten) Berechnung der Schritte zwei und drei, bei Eingabe des Wortes 011000♯011000.

Abbildung 1: Auszug aus der Arbeit von M_1

0	1	1	0	0	0	♯	0	1	1	0	0	0	␣ . . .
x	1	1	0	0	0	♯	0	1	1	0	0	0	␣ . . .
0	1	1	0	0	0	♯	x	1	1	0	0	0	␣ . . .
x	1	1	0	0	0	♯	x	1	1	0	0	0	␣ . . .
x	x	1	0	0	0	♯	x	1	1	0	0	0	␣ . . .
													. . .
x	x	x	x	x	x	♯	x	x	x	x	x	x	␣ . . .
													accept

Die Beschreibung von M_1 skizziert die Arbeitweise, gibt aber nicht alle Details. Wir können eine Turingmaschine in allen Details beschreiben, in dem wir eine formale Definition analog zu der für DEA und PDA verwendeten einführen. Da eine formal korrekte Beschreibung aber oft sehr lang ist, werden wir eine solche Beschreibung im weiteren Verlauf der Vorlesung nur selten benutzen.

2.1.1 Formale Definition

Das Kernstück einer Turingmaschine ist die Überführungsfunktion δ, die beschreibt, unter welchen Umständen die Maschine in den nächsten Zustand übergeht. Für eine Turingmaschine hat sie den Typ $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$. Wenn die Maschine im Zustand q das Zeichen a liest, dann gibt $\delta(q, a) = (r, b, L)$ an, dass der Kopf nach dem Lesen nach links bewegt werden soll und an die Stelle von a ein b aufs Band geschrieben wird und die Maschine sich anschließend im Zustand r befindet. L und R stehen dabei für die Bewegungsrichtung des Kopfes, also links oder rechts.

Definition 1. *Eine Turingmaschine ist ein 7-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, wobei Q, Σ und Γ endliche Mengen sind und folgende Bedingungen erfüllt werden:*

1. *Q ist die Menge der Zustände,*
2. *Σ ist das Eingabealphabet, das nicht das Leersymbol $\sqcup$ enthält,*
3. *Γ ist das Bandalphabet, wobei $\sqcup \in \Gamma$ und $\Sigma \subset \Gamma$,*
4. *$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ ist die Überführungsfunktion,*
5. *q_0 ist der Startzustand,*
6. *q_{accept} ist der akzeptierende Zustand,*
7. *q_{reject} ist der ablehnende Zustand.*

Verhalten

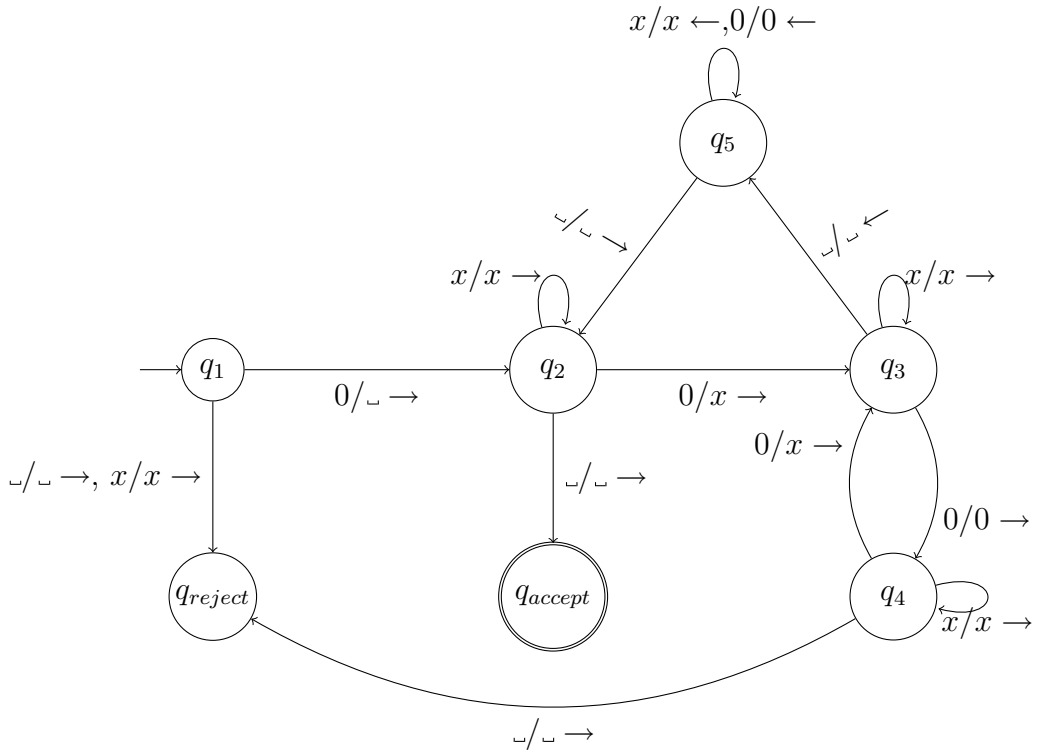
Eine Turingmaschine $M = (Q, \Sigma, \Gamma\delta, q_0, q_{accept}, q_{reject})$ arbeitet wie folgt: Am Anfang bekommt die Maschine als Eingabe ein Wort $w = w_1...w_n \in \Sigma^*$, das auf den ersten(linkesten) n Zellen des Bandes steht, der Rest des Bandes ist mit Blanks gefüllt. Man beachte, dass Σ nicht das Blanksymbol enthält, sodass mit dem ersten Blank das Ende der Eingabe erkennbar ist. Der Kopf steht auf w_1 und die Maschine ist im Zustand q_0. Wenn die Maschine

anfängt zu arbeiten, so tut sie das entsprechend den Regeln, die durch δ vorgegeben sind. Falls der Kopf ganz links steht und die nächste Anweisung lautet, nach links zu gehen, so bleibt der Kopf trotzdem in der ersten Position. Die Berechnungen erfolgen solange, bis einer der beiden Zustände q_{accept} oder q_{reject} erreicht wird, wo die Maschine anhält. Falls das nicht eintritt, läuft sie unendlich lange weiter.

Übergangsdiagramme

Eine Möglichkeit, das Verhalten von Turingmaschinen darzustellen besteht darin, Übergangsdiagramme anzugeben. Dabei werden die Zustände durch Knoten dargestellt, wobei der Startknoten durch einen Startpfeil markiert wird und der akzeptierende Endzustand durch einen doppelten Kreis. An die Übergänge werden das gelesene Zeichen und durch einen Schrägstrich davon getrennt das zu schreibende Zeichen sowie die Bewegungsrichtung des Kopfes notiert. Die Abbildung 2 zeigt ein Beispiel:

Abbildung 2: Übergangsdiagramm für M mit $L(M) = \{0^{2^n} \mid n \in \mathbb{N}\}$



Konfigurationen

In jedem Schritt kann sich der aktuelle Zustand, der aktuelle Bandinhalt und die aktuelle Position des Kopfes ändern.

Die Belegung dieser drei Größen nennt man Konfiguration der Turingmaschine. Konfigurationen werden auf eine bestimmte Weise dargestellt:

Definition 2. *Die Konfiguration einer Turingmaschine ist ein Tripel (u, q, v), geschrieben als (uqv), wobei uv der aktuelle Inhalt des Bandes, q der aktuelle Zustand und das erste Zeichen von v die aktuelle Position des Kopfes ist.*

Beispiel 2. $(1011q_701111)$ *stellt die Konfiguration dar, wenn der aktuelle Bandinhalt* 101101111 *ist, die Maschine sich im Zustand q_7 befindet und der Kopf über der zweiten Null steht.*

Um das intuitive Verständnis der Arbeitsweise von Turingmaschinen zu formalisieren, verwenden wir *Konfigurationsübergänge.*

Definition 3. *Eine Konfiguration K_2 ist Nachfolgekonfiguration von K_1, falls die Turingmaschine in einem Schritt von K_1 nach K_2 gelangt.*

Beispiel 3. *Seien a, b und c Zeichen des Bandalphabetes Γ einer Turingmaschine und $u, v \in \Gamma^*$, weiterhin seien q_i und q_j Zustände. Dann sind uaq_ibv und uq_jacv zwei Konfigurationen. Die Konfiguration uq_jacv ist die Nachfolgekonfiguration von uaq_ibv, falls die Überführungsfunktion an der Stelle (q_i, b) den Wert (q_j, c, L) hat, also $\delta(q_i, b) = (q_j, c, L)$. Weiterhin ist $uacq_iv$ Nachfolgekonfiguration falls $\delta(q_ib) = (q_jcR)$. Sonderfälle treten ein, wenn, wie schon erwähnt, der Kopf bei Anweisung für eine Bewegung nach links bereits am linken Bandende steht, dann ist q_jcv die Nachfolgekonfiguration von q_ibv, wenn $\delta(q_i, b) = (q_j, c, L)$ Für das rechte Ende ist die Konfiguration (uaq_i) äquivalent zu $(uaq_i\sqcup)$, da wir annehmen, dass das Band mit Blanks gefüllt ist. Das heißt, wir können diesen Fall wie gewöhnlich behandeln, d.h der Kopf geht über das rechte Ende des Wortes hinaus.*

Die Startkonfiguration von M bei Eingabe von w ist q_0w, was kennzeichnet, dass der Kopf über dem ersten Zeichen von w steht und die Maschine sich im Zustand q_0 befindet. In einer akzeptierenden Konfiguration ist der Zustand q_{accept}– in einer ablehnenden Konfiguration ist der Zustand q_{reject}. Akzeptierende und ablehnende Konfigurationen sind Haltekonfigurationen und haben keine Nachfolgekonfigurationen.

Definition 4. *Eine Turingmaschine M akzeptiert eine Eingabe w, falls eine Folge $K_1 \dots K_k$ von Konfigurationen existiert, sodass*

1. *K_1 ist Startkonfiguration von M bei Eingabe von w,*

2. *jedes K_{i+1} ist Nachfolgekonfiguration von K_i und*

3. *K_k ist eine akzeptierende Konfiguration.*

Die Menge der Eingaben, die von M akzeptiert werden, heißt Sprache von M, bezeichnet mit $L(M)$.

Definition 5. *Eine Sprache heißt Turing-akzeptierbar, rekursiv aufzählbar, semi-entscheidbar, wenn es eine Turingmaschine gibt, die diese Sprache akzeptiert.*

Wenn wir eine Turingmaschine auf einer Eingabe starten, gibt es drei Möglichkeiten, wie sie sich verhalten kann: sie kommt in einen der Haltezustände (accept oder reject) oder sie hält nicht an. Wenn das Wort nicht akzeptiert wird, hat man entweder ein *reject* oder eine Schleife.

Bemerkung: Mit Schleife ist, wenn nicht ausdrücklich anders definiert, im Folgenden nur gemeint, dass die Maschine nicht anhält. Sie muss nicht unbedingt für immer dieselben Schritte wiederholen, wie der Ausdruck Schleife suggeriert. Schleife kann jedes einfache oder komplexe Verhalten bedeuten, das niemals zu einem Haltezustand führt.

Praktisch lässt sich eine Schleife aber nicht von einer langen Laufzeit unterscheiden. Am liebsten hat man daher solche Turingmaschinen, die auf jeder Eingabe anhalten, sogenannte Entscheider.

Definition 6. *Ein Entscheider ist eine Turingmaschine, die auf jeder Eingabe anhält (sie entscheidet die Sprache). Eine Sprache heißt (Turing) - entscheidbar oder rekursiv, wenn es eine Turingmaschine gibt, die sie entscheidet.*

Jede entscheidbare Sprache ist auch Turing-akzeptierbar, aber nicht notwendig umgekehrt. Im folgenden geben wir einige Beispiele für entscheidbare Sprachen an. In Kapitel 2, wenn wir eine Technik kennengelernt haben werden, mit deren Hilfe man beweisen kann, dass eine Sprache nicht entscheidbar ist, folgen Beispiele für Sprachen, die Turing-akzeptierbar, aber nicht entscheidbar sind.

Beispiel 4.

1. $L_1 = \{0^{2^n} \mid n \in \mathbb{N}\}$.

2. $L_2 = \{a^i b^j c^k \mid i \times j = k \text{ und } i, j, k \geq 1\}$.

3. $L_3 = \{\sharp x_1 \sharp x_2 \sharp \ldots \sharp x_n \mid x_i \in \{0,1\}^* \text{ und } x_i \neq x_j \text{ wenn } i \neq j\}$.

2.2 Andere Turingmaschinenvarianten

Es gibt sehr viele alternative Definitionen für Turingmaschinen inklusive solche mit meherern Bändern oder mit Nichtdeterminismus. Das ursprüngliche Modell und all seine Variationen sind gleich mächtig, sie erkennen dieselbe Klasse von Sprachen. In diesem Abschnitt werden wir drei solche Klassen beschreiben und die jeweiligen Äquivalenzbeweise skizzieren. Dabei beruht die Idee für Äquivalenzbeweise von Modellen immer darauf, dass wir zeigen, wie wir das eine durch das andere Modell simulieren können.

2.2.1 Mehrband-Turingmaschine

Bei einer Mehrband-Turingmaschine gibt es statt einem, mehrere rechtsseitig unendliche Bänder, jedes Band hat einen eigenen Kopf zum Lesen und Schreiben. Die Überführungsfunktion darf jetzt alle Köpfe gleichzeitig bewegen und lesen bzw. schreiben. Formal bedeutet das, dass die Überführungsfunktion δ den folgenden Typ hat:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k,$$

wobei k die Anzahl der Bänder ist. Der Ausdruck

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

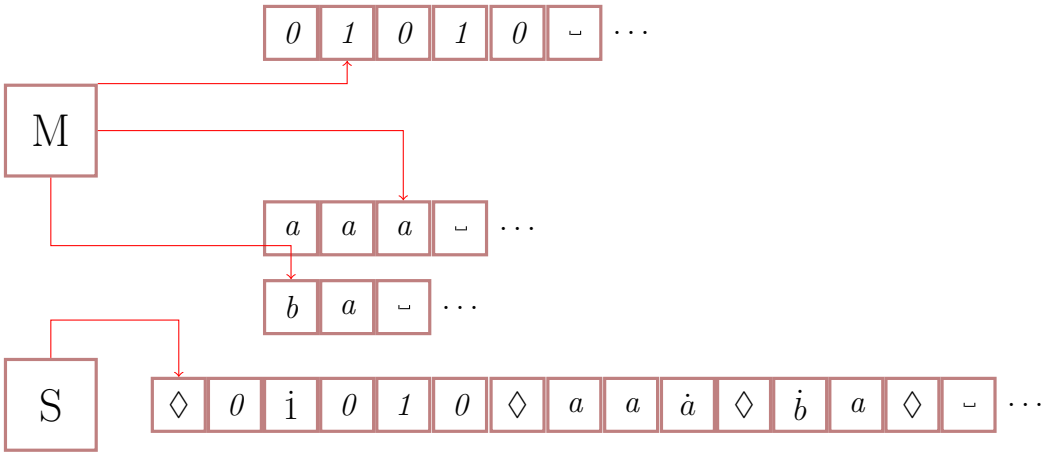
bedeutet, dass wenn die Maschine im Zustand q_i mit den Köpfen 1 bis k die Zeichen $a_1 \ldots a_k$ von den k Bändern liest, geht sie in den Zustand q_j über schreibt auf die i-ten Bänder die Zeichen b_i und bewegt die i-ten Köpfe wie in der i-ten Position der Folge von L und R angegeben. Die Mehrband-Turingmaschine startet mit der Eingabe auf dem ersten Band, die Köpfe stehen alle auf der jeweils ersten Position ihres Bandes. Auf den ersten Blick scheinen Mehrbandturingmaschinen mächtiger zu sein als gewöhnliche Turingmaschine. Wir erinnern uns, dass zwei Maschinen äquivalent sind, wenn sie die gleiche Sprache akzeptieren.

Satz 1. *Zu jeder Mehrband-Turingmaschine gibt es eine äquivalente Einband-Turingmaschine.*

Beweis: Wir zeigen, wie man die Mehrbandturingmaschine M in eine äquivalente Einbandturingmaschine S umwandelt. Sagen wir, M hat k Bänder. Die Einband-Turingmaschine S simuliert den Effekt der k Bänder, indem sie deren Information auf ihrem einzigen Band speichert. Wir benutzen das neue Symbol $\Diamond$ als Trennzeichen, um die Inhalte der verschiedenen Bänder voneinander abzugrenzen. Zusätzlich zum Bandinhalt muss sich S

auch die Position der verschiedenen Köpfe „merken“ . Das wird gemacht, indem das Zeichen über dem der Kopf stehen würde mit einem Punkt über dem Zeichen versehen wird. Wie zuvor beim ◊ sind die punktierten Zeichen neue Zeichen, die zum Bandalphabet hinzugefügt wurden. Abbildung 3 zeigt, wie ein Band verwendet werden kann, um drei Bänder zu simulieren:

Abbildung 3: Simulation einer 3-Band-TM durch eine TM



$S =$

„Auf Eingabe $w = w_1, \ldots w_n$:

1. S stellt alle k Bänder von M auf einem Band dar. Das formatierte Band enthält:
$$\Diamond \dot{w}_1 w_2 \ldots w_n \Diamond \dot{\sqcup} \Diamond \dot{\sqcup} \Diamond \ldots \Diamond$$

2. Um einen Schritt zu simulieren, sucht S das Band vom ersten ◊, der die linke Grenze markiert, bis zum $k+1$-ten ◊, der die rechte Grenze markiert ab, um die Symbole unter den vituellen Köpfen zu bestimmen. Anschließend macht S den Übergang, entsprechend der Überführungsfunktion von M.

3. Wenn an irgendeinem Punkt S einen der virtuellen Köpfe nach rechts auf eins der ◊-Symbole bewegt, so ist das ein Zeichen dafür, dass M den entsprechenden Kopf auf ein vorher mit ␣ gefüllten Teil des Bandes bewegt hätte. Also schreibt S ein Blank-Symbol an diese Stelle und schiebt den restlichen Bandinhalt bis zum letzten ◊ um eine Zelle nach rechts. Anschließend fährt S mit der Simulation fort wie zuvor.

“

□

Folgerung: Eine Sprache ist genau dann Turing-akzeptierbar, wenn es eine Mehrband-Turingmaschine gibt, die sie akzepiert.

2.2.2 Nichtdeterministische Turingmaschinen

Ähnlich wie bei NEA lassen sich auch nichtdeterministische Turingmaschinen definieren. An jedem Punkt der Berechnung hat die Maschine mehrere (festgelegte) Möglichkeiten, wie sie sich verhalten kann. Die Überführungsfunktion einer Nichtdeterministische Turingmaschine hat die Form:

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

Die Berechnung einer Nichtdeterministische Turingmaschine ist ein Baum, dessen Zweige den verschiedenen Möglichkeiten der Maschine entsprechen. Eine Nichtdeterministische Turingmaschine akzeptiert eine Eingabe, wenn mindestens ein Zweig zu einem akzeptierenden Zustand führt. Auch diese scheinbare Erweiterung der Kapazität führt nicht zu einem mächtigeren Berechnungsmodell. Eine Nichtdeterministische Turingmaschine ist ein Entscheider, wenn jeder Pfad zu einem Haltezustand führt.

Satz 2. *Für jede nichtdeterministische Turingmaschine gibt es eine äquivalente deterministische Turingmaschine.*

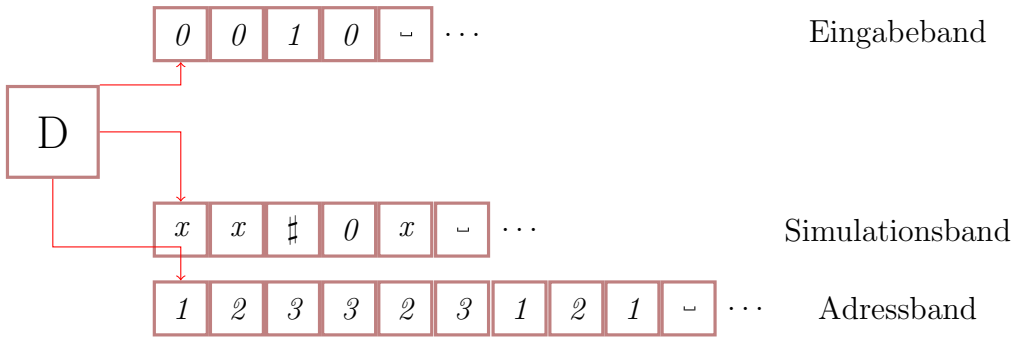
Beweisidee: Wir zeigen, dass man jede Nichtdeterministische Turingmaschine N durch eine deterministische Maschine D simulieren kann. Die Idee dabei ist, dass D alle möglichen Zweige der nichtdeterministischen Berechnung von N ausprobiert. Falls D dabei in einem der Zweige einen akzeptierenden Zustand findet, dann akzeptiert D. Andernfalls läuft D unendlich lange weiter. Wir betrachten die Berechnung von N auf der Eingabe w als Baum. Jeder Zweig des Baumes stellt einen Berechnungszweig dar, jeder Knoten ist eine Konfiguration. Wir legen fest, dass D eine Breitensuche nach der akzeptierenden Konfiguration durchführt. (Bei Tiefensuche besteht die Gefahr, dass D sich in einem unendliche langen Pfad „verrennt“.)

Beweis: Unsere simulierende TM D hat drei Bänder, die sie in folgender Weise verwendet: Band 1 enthält den Eingabestring und wird nie verändert, Band 2 verwaltet die Kopie vom Band von N, während eines Zweiges der nichtdeterministischen Berechnung und Band 3 „merkt sich“ , an welcher Stelle vom Berechnungsbaum von N sich D befindet.

Die Bänder von D sind in Abbildung 4 dargestellt.

Betrachten wir zunächst die Darstellung auf dem dritten Band. Jeder Knoten im Berechnungsbaum kann höchstens n Nachfolger haben, wenn n die

Abbildung 4: Simulation einer Nichtdeterministische Turingmaschine durch eine 3-Band-Turingmaschine



maximale Anzahl an möglichen Verzweigungen für die Überführungsfunktion von N ist. Wir geben jedem Knoten im Baum eine Adresse, d.h. einen String über dem Alphabet $\Sigma_n = \{1, 2 \dots n\}$. Für jede Ebene des Baumes werden die gewählten Abzweigungen notiert. Da nicht immer die maximale Anzahl der Verzweigungen möglich ist, sind nicht alle solche Adressen gültig, d.h. entsprechen möglichen Konfigurationen. Die aktuelle Adresse bei Simulation von N wird auf Band 3 notiert, der leere String steht dabei für die Wurzel. D arbeitet wie folgt:

1. Band 1 enthält am Anfang die Eingabe w, Band 2 und 3 sind leer.

2. Kopiere Band 1 auf Band 2.

3. Verwende Band 2 um N mit Eingabe w auf einem Zweig der Berechnung zu simulieren. Vor jedem Schritt von N betrachte das Symbol auf Band 3, um zu bestimmen, welche Entscheidung der Überführungsfunktion von N gefolgt wird. Falls auf Band 3 kein Symbol mehr übrig ist, oder keine gültige Wahl ergibt, brich diesen Zweig ab und gehe über in Schritt 4. Falls eine reject-Konfiguration erreicht wird, gehe zu Schritt 4. Falls eine akzeptierende Konfiguration erreicht wird, akzeptiere die Eingabe.

4. Ersetze den String auf Band 3 durch den lexikographisch nächsten String. Simuliere den nächsten Berechnungszweig von N durch Übergang in Schritt 2.

□

Folgerung:

1. Eine Sprache ist Turing-akzeptierbar, genau dann, wenn es eine Nichtdeterministische Turingmaschine gibt, die sie akzeptiert.

2. Eine Sprache ist genau dann entscheidbar, wenn eine Nichtdeterministische Turingmaschine existiert, die sie entscheidet.

2.2.3 Aufzähler

Manchmal werden Turing-akzeptierbare Sprachen auch als *rekursive* oder *aufzählbare* Sprachen bezeichnet. Diese Begriffe stammen von einer Turingmaschinenvariante, die *Aufzähler* genannt wird. Grob gesagt ist ein Aufzähler eine Turingmaschine mit einem angehängten Drucker. Anstatt Wörter zu akzeptieren oder abzulehnen arbeitet eine solche Maschine ohne Eingabe vor sich hin und gibt von Zeit zu Zeit bestimmte Zeichenketten auf den Drucker aus. Jedes Mal, wenn eine Zeichenkette zur Liste hinzugefügt werden soll, wird sie an den Drucker gesendet.

Der Aufzähler startet mit einem leeren Band. Falls er nicht anhält, druckt er eine unendliche Liste von Zeichenketten. Die Sprache, die von E (dem Aufzähler) aufgezählt wird, ist die Menge aller Strings, die E irgendwann ausgibt. Dabei darf E diese Menge in einer beliebigen Ordnung ausgeben, auch mit Wiederholungen.

Jetzt kommen wir zum Zusammenhang zwischen Aufzählern und Turing-akzeptierbaren Sprachen.

Satz 3. *Eine Sprache ist genau dann Turing-akzeptierbar, wenn es einen Aufzähler gibt, der sie aufzählt.*

Beweis: Zuerst zeigen wir, dass es für jeden Aufzähler E der Sprache A eine Turingmaschine M gibt, die A akzeptiert. M arbeitet folgendermaßen:
$M =$

„Auf Eingabe w:

1. Lasse E laufen, vergleiche alle Ausgaben von E mit w.

2. Falls w in der Liste auftaucht, akzeptiere.

“

Offensichtlich akzeptiert M genau die Wörter, die E aufzählt.

Für die andere Beweisrichtung nehmen wir an, M sei eine Turingmaschine, die eine Sprache A akzeptiert. Wir konstruieren einen Aufzähler E für A:

Sei $s_1, s_2 \ldots$ eine Liste aller möglichen Strings in Σ^*, dann wird E wie folgt konstruiert:

$E =$

„Auf Eingabe w:

1. (Ignoriere die Eingabe.)
2. Wiederhole für $i = 1, 2, \ldots$ die folgenden Schritte:
3. Lasse M jeweils die ersten i Schritte auf den Eingaben $s_1, \ldots, s_i$ arbeiten.
4. Falls eine der Eingaben akzeptiert wird, dann gib das entsprechende s_j aus.

“

Offenbar wird jedes s, das von M akzeptiert wird, irgendwann auf der Liste von E ausgegeben. (Tatsächlich sogar unendlich oft.) Der Effekt dieses Vorgehens ist, dass man M parallel auf allen möglichen Eingaben laufen lässt. □

2.3 Äquivalenz zu anderen Modellen

Bisher haben wir verschiedene Varianten von Turingmaschine gesehen und haben gezeigt, dass diese in Bezug auf die von ihnen berechneten Sprachen äquivalent sind. Dabei sind wir intuitiv von einer bestimmten Vorstellung von Berechnung ausgegangen. Man kann Turingmaschinen als Rechenmaschinen betrachten, wenn man den Inhalt des Bandes[1] bei Erreichen eines Haltezustandes als Funktionswert für die Eingabe betrachtet. Da eine beliebige Turingmaschine nicht auf jeder Eingabe anhält, müssen wir dabei beachten, dass eine solche Funktion nicht überall definiert ist. Wenn wir in der Berechenbarkeitstheorie von Funktionen sprechen, meinen wir daher- anders als in der Mathematik- im Allgemeinen partielle Funktionen. Je nachdem, ob es für eine Funktion eine Maschine gibt, die sie berechnet, können wir Funktionen klassifizieren:

Definition 7. *1. Eine Funktion $f : \Sigma^* \to \Sigma^*$ heißt total berechenbar, wenn es eine Turingmaschine gibt, die auf jeder Eingabe $w \in \Sigma^*$ anhält mit $f(w)$ als aktuellem Bandinhalt.*

[1] Der Inhalt des Bandes ist der String beginnend mit dem Zeichen unter dem Kopf bis zum ersten Zeichen, das nicht in Σ liegt.

2. *Eine Funktion f heißt partiell berechenbar, wenn es eine Turingmaschine gibt, die auf jeder Eingabe aus dem Definitionsbereich von f anhält und den Wert $f(w)$ als aktuellen Bandinhalt hat.*

Neben Turingmaschinen gibt es viele weitere Modelle für allgemeine Berechnungen – solche die Turingmaschine sehr ähnlich sind und solche, die weniger ähnlich sind. Aber es gibt einige Gemeinsamkeiten, die sie alle teilen, wie z.B. den unbeschränkten Zugang zu unbegrenzt viel Speicher im Gegensatz zu schwächeren Modellen wie PDA oder endlichen Automaten. Tatsächlich sind alle denkbaren Modelle mit diesem Merkmal äquivalent, falls sie gewisse vernünftige Forderungen erfüllen, wie zum Beispiel die Forderung, dass in einem einzelnen Schritt nur ein endliches Pensum von Arbeit erledigt werden kann.

Um dieses Phänomen zu verstehen, betrachte man die analoge Situation bei Programmiersprachen. Viele davon wie z. B. PASCAL und LISP unterscheiden sich in Stil und Struktur voneinander. Gibt es also Algorithmen, die wir in der einen Sprache programmieren können, in anderen aber nicht? Natürlich nicht. Wir können LISP in PASCAL übersetzen und umgekehrt, d.h. beide Sprachen beschreiben exakt dieselbe Klasse von Algorithmen. Die weit verbreitete Äquivalenz von Computermodellen beruht auf derselben Ursache. Irgendzwei Modelle, die bestimmte vernünftige Forderungen erfüllen, können sich gegenseitig modellieren und sind daher gleichmächtig. Dieses Äquivalenzphänomen hat eine wichtige philosophische Folgerung. Obwohl es viele Berechnungsmodelle gibt, ist die Klasse der Algorithmen, die davon beschrieben wird, eindeutig bestimmt. Während die Definition eines Berechnungsmodells relativ beliebig ist, ist die von ihm beschriebene Klasse von Algorithmen in gewissem Sinne natürlich, da sie dieselbe ist wie die von anderen Modellen. Das hat – wie wir im nächsten Abschnitt sehen werden – schwerwiegende Implikationen für die Mathematik.

2.4 Algorithmen

Informal gesehen ist ein Algorithmus eine Ansammlung von einfachen Anweisungen, um eine bestimmte Aufgabe auszuführen. In der Umgangssprache heißen Algorithmen manchmal auch Prozeduren oder Konzepte. In der Mathematik spielen sie ebenfalls eine große Rolle. So ist die historische Mathematikliteratur voll von Beschreibungen von Algorithmen für die verschiedenste Aufgaben, wie z.B. für das Finden von Primzahlen oder für die Berechnung des größten gemeinsamen Teilers. In der gegenwärtigen Literatur gibt es sie ebenfalls im Überfluss.

Allerdings wurde der Begriff des Algorithmus nicht vor dem 20. Jahrhun-

dert genau definiert. Davor gab es nur ein intuitives Verständnis und davon hing es ab, wie man mit Algorithmen umging. Dieser intuitive Begriff war allerdings nicht ausreichend, um ein tieferes Verständnis von Algorithmen zu bekommen. Wie wir heute wissen, gibt es Funktionen, die nicht berechenbar sind. Zu beweisen, dass man für eine Problemlösung kein mechanisches Verfahren angeben kann, ist aber unmöglich, wenn man die Klasse der Algorithmen nicht beschreiben kann, während man zum Finden eines Algorithmus nicht unbedingt mehr als eine gute Idee von der Sache braucht.

2.4.1 Historisches

Im Jahr 1900 hielt David Hilbert (1862 – 1942) eine programmatische Rede auf dem Internationalen Mathematikerkongress in Paris, in der er eine Liste von 23 Problemen vorstellte, die die vordringlichsten Aufgaben des neuen Jahrhunderts sein sollten. Die Bearbeitung dieser Probleme beeinflusste die mathematische Forschung erheblich. Das zehnte Problem auf dieser Liste hat mit Algorithmen zu tun. Um es zu erklären, brauchen wir den Begriff eines Polynoms.

Definition 8. *Ein Polynom ist eine Summe von Termen, wobei jeder Term ein Produkt bestimmter Variablen und einer Konstanten (Koeffizienten) ist.*

Beispiel 5.

$$6 \cdot x \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

ist ein Term mit dem Koeffizienten 6

$$P = 6x^3yz^2 + 3xy^2 – x^3 – 10$$

ist ein Polynom mit vier Termen in den Variablen x, y *und* z*.*

Definition 9. *Die Wurzel eines Polynoms ist eine Belegung der Variablen, so dass der Wert des Polynoms* 0 *ist.*

Beispiel 6. $x = 5, y = 3, z = 0$ *ist eine Wurzel von* P*.*

Wir sagen, dass eine Wurzel ganzzahlig ist, wenn die Belegung mit ganzen Zahlen erfolgt.

Hilberts zehntes Problem war die Aufgabe, einen Algorithmus zu entwerfen, der feststellt (testet), ob ein Polynom ganzzahlige Lösungen hat. Ohne das Wort Algorithmus zu verwenden, suchter er nach einer Prozedur, die es erlaubt, in endlich vielen Schritten (nach einer endlichen Anzahl von Operationen) zu bestimmen, ob es eine solche Lösung gibt. Interessanterweise stellte er die Aufgabe, eine solche Prozedur zu entwerfen, scheint also angenommen zu haben, dass es eine solche gibt und dass sie nur noch jemand finden muss.

2.4.2 Church-Turing-These

Wie wir heute wissen, existiert kein solcher Algorithmus. Zu dieser Schlussfolgerung zu kommen war aber aufgrund des unklaren Algorithmusbegriffes von 1900 für die damaligen Mathematiker unmöglich. Dass tatsächlich kein Lösungsverfahren für den allgemeinen Fall existiert, wurde erst 1970 durch Juri Matijasevich bewiesen. Um den Beweis zu führen brauchte dieser u.a. auch eine Definition des Begriffes Algorithmus. Eine solche Definition wurde aber erst 1936 mit Artikeln von Alan Turing und Alonzo Church gegeben. Turing verwendete dafür die später so genannten Turing-Maschinen, Church entwarf ein Beschreibungssystem, den so genannten λ-Kalkül. Es konnte gezeigt werden, dass beide Definitionen äquivalent sind.

Die Verbindung zwischen dem intuitiven Berechenbarkeitsbegriff und der genauen Definition nennt man die Church-Turing-These.

Intuitive Berechenbarkeit ist äquivalent zu Turing-Berechenbarkeit.

Wir werden im Kapitel 2 die Techniken kennenlernen, mit denen man zeigen kann, dass ein bestimmtes Problem algorithmisch nicht lösbar ist. Wie sieht Hilberts zehntes Problem in unserer Begriffswelt aus?
Sei

$$D = \{p \mid p \text{ ist ein Polynom mit einer ganzzahligen Wurzel}\}$$

Hilberts zehntes Problem besteht darin, zu fragen, ob die Menge D entscheidbar ist. Die Antwort ist negativ. Andererseits können wir aber zeigen, dass D ist Turing-akzeptierbar ist. Bevor wir das tun, betrachten wir zunächst ein einfacheres Problem, das analog zu Hilberts Problem für Polynome mit einer Variablen ist, nämlich

$$D_1 = \{p \mid p \text{ ist ein Polynom in } x \text{ mit einer ganzzahligen Wurzel.}\}$$

Eine Turingmaschine, die D_1 akzeptiert, sieht folgendermaßen aus:
$M_1 =$

„Auf Eingabe p, wobei p ein Polynom in x ist:

1. Belege x der Reihe nach mit den Werten $0, -1, 1, -2, 2, -3, \ldots$ und werte $p(x)$ aus. Falls an einer Stelle das Polynom 0 ergibt, accept.

“

Falls p keine ganzzahlige Wurzel hat, wird die Maschine niemals anhalten, aber wenn es eine Wurzel hat, dann wird sie sie finden.

Für den Fall eines Polynoms in mehreren Variablen können wir eine ähnliche Turingmaschine angeben, die D akzeptiert. Dabei muss M durch alle möglichen Belegungen der Variablen mit ganzen Zahlen geben. Sowohl M als auch M_1 sind keine Entscheider. Allerdings kann M_1 in einen Entscheider für D_1 umgewandelt werden, da wir Schranken berechnen können, innerhalb derer die Wurzeln eines Polynoms in einer Variablen liegen müssen, wenn sie existieren. Man kann zeigen (Übungsaufgabe!), dass die Wurzeln zwischen den Werten

$$\pm k \frac{c_{max}}{c_1}$$

liegen müssen. Dabei ist k die Anzahl der Terme des Polynoms, c_{max} ist der Koeffizient mit dem größten Absolutbetrag und c_1 ist der Koeffizient des Terms mit der höchsten Potenz. Wenn innerhalb der Schranken keine Wurzel gefunden wurde, hält die Maschine an und lehnt ab. Der Beweis von Matijasevich beruht darauf, dass er beweist, dass das Berechnen solcher Schranken für allgemeine Polynome unmöglich ist.

2.5 Sprechen über Turingmaschinen

Im Folgenden werden wir zwar von Turingmaschinen sprechen, aber unser Hauptaugenmerk liegt auf den durch sie zu beschreibenden Algorithmen. D.h. die Turingmaschine dient nur als genaues Modell für die Definition eines Algorithmus. Bevor wir uns von der Möglichkeit, bestimmte häufig wiederkehrende Konstruktionen mit Turingmaschinen zu programmieren überzeugen, wollen wir einen Standard angeben, mit dem Turingmaschinen-Algorithmen beschrieben werden sollen. Wie viele Details muss man bei der Beschreibung einer Turingmaschine angeben? Es gibt drei Möglichkeiten:

1. Die formale Beschreibung: Darstellung der Zustände, Überführungsfunktion usw.,

2. Implementierungsbeschreibung: in Umgangssprache wird beschrieben, wie sich der Kopf bewegt und in welcher Weise die Daten auf dem Band gespeichert werden. (Dazu muss man nicht die Details der Zustände und der Überführungsfunktion angeben.)

3. High-level Beschreibung: es wird allein der Algorithmus angegeben (unter Ignorieren des konkreten Modells) also keine Angaben über Bewegung der Köpfe und Bänder.

Im Folgenden werden wir uns meist auf die High-Level-Beschreibung beschränken. Dabei sind folgende Dinge zu beachten.
Eingaben sind immer **Zeichenketten**. Falls wir ein anderes Objekt brauchen, müssen wir dieses Objekt als Zeichenkette darstellen. Man kann Polynome, Graphen, Grammatiken, Automaten und jede Kombination davon als String darstellen. Eine Turingmaschine kann programmiert werden, diese Darstellung zu dekodieren, um sie so zu interpretieren, wie es beabsichtigt war.
Notation: O für das Objekt, $< O >$ für dessen Kodierung als Zeichenkette

Falls wir **mehrere Objekte** $O_1, \ldots O_n$ haben, können wir sie auch als eine Zeichenkette $< O_1, \ldots O_n >$ darstellen. Die Kodierung selbst kann auf unterschiedliche Weise erfolgen, es spielt keine Rolle, welche man wählt, da eine Turingmaschine jede Kodierung in eine andere übersetzen kann.

2.5.1 Beschreibung von Turingmaschinen

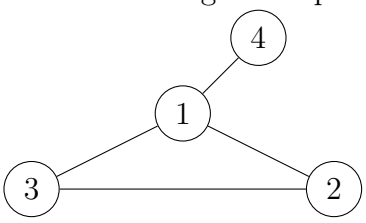
- Wir beschreiben den Turingmaschinen-Algorithmus innerhalb von Anführungszeichen,
- Numerierung erfolgt nach Stadien.
- Blockstruktur wird durch Einrücken erkennbar gemacht.
- Erste Zeile = Eingabe – die Eingabe ist einfach eine Zeichenkette. Falls die Eingabe die Kodierung eines Objektes ist, z.B. $< A >$ wird im ersten Schritt geprüft, ob die Kodierung korrekt ist, falls nicht, wird die Eingabe abgelehnt.

Das folgende Beispiel hat mit Graphen zu tun. Dafür brauchen wir den Begriff eines verbundenen Graphen.

Definition 10. *Ein Graph heißt verbunden, wenn jeder Knoten von jedem anderen aus durch Verfolgen der Kanten erreichbar ist.*

Beispiel 7. $A = \{< G > |\ G$ *ist verbundener ungerichteter Graph*$\}$. *Abbildung 5 zeigt einen verbundenen Graphen und seine Darstellung als Zeichenkette.*

Abbildung 5: Graph $< G >= (1,2,3,4),((1,2),(2,3),(3,1),(1,4))$



Wir geben eine Turingmaschine M an, die A entscheidet:
$M =$

„Auf Eingabe $< G >$

1. *Wähle den ersten Knoten und markiere ihn,*
2. *Wiederhole folgende Anweisungen bis keine neuen Knoten mehr markiert werden,*
3. *Gehe alle Knoten in G durch und markiere sie, wenn sie von irgendeinem bereits markierten Knoten erreichbar sind,*
4. *Überprüfe, ob alle Knoten von G markiert sind. Falls ja,* accept*; falls nein,* reject*.*

“

Implementierungsdetails

Eingabecheck: *Um die Wohlgeformtheit zu überprüfen, scannt die Turingmaschine die Eingabe. Diese sollte aus zwei Listen bestehen, deren erste ein Liste von verschiedenen Zahlen ist und deren zweite eine Liste von Paaren aus den Zahlen der ersten Liste ist. Um zu überprüfen, ob die Knotenliste Wiederholungen enthält, verwende man einen Algorithmus für die Sprache L_3 aus Beispiel 4. Eine ähnliche Methode verwendet man für die zweite Bedingung. Falls w beide Tests erfüllt, ist w die Kodierung eines Graphen und M geht zu Schritt* 1 *über.*

Schritt 1: *Der erste Knoten (das linkeste Zeichen auf dem Band) wird mit einem Punkt markiert(punktiert),*

Schritt 2(und 3) *M durchsucht die Knotenliste nach einem unmarkierten Knoten n_1 und kennzeichnet ihn durch eine Markierung (verschieden*

von der Punktmarkierung, z.B. Unterstrich) danach sucht M einen punktierten Knoten n_2 und unterstreicht ihn ebenfalls; für jede Kante wird daraufhin überprüft, ob sie das unterstrichene Paar ist. Falls ja, bekommt n_1 ebenfalls eine Punktmarkierung. Die Unterstreichungen werden gelöscht und es geht weiter mit Schritt 2. Falls nicht, prüft M die nächste Kante in der Liste. Wenn die Kante $\{n_1, n_2\}$ nicht vorkommt, ist sie keine Kante des Graphen, und der Unterstrich unter n_2 wird zum nächsten punktierten Knoten verschoben. M wiederholt die Schritte wie zuvor und prüft, ob das neue Paar $\{n_1, n_2\}$ eine Kante ist. Falls es keine punktierten Knoten mehr gibt, dann ist n_1 nicht mit einem punktierten Knoten verbunden. M setzt die Unterstreichungen so, dass n_1 der nächste unpunktierte Knoten und n_2 der erste punktierte Knoten ist und wiederholt die Schritte in diesem Abschnitt. Wenn es keine unpunktierten Knoten mehr gibt, konnte M keine neuen Knoten zum Punktieren finden und geht über zu Schritt 4.

Schritt 4: *M überprüft die Liste der Knoten, um festzustellen, ob alle punktiert sind. Falls ja* accept*, falls nein* reject.

2.5.2 Speichern von Daten in Zuständen

Die endliche Steuerung (die Zustandsmenge) kann nicht nur dazu verwendet werden, eine bestimmte Stelle im Programm der Turingmaschine zu markieren, sondern auch um (eine endliche Menge von) Daten zu speichern. Dazu stellt man sich den Zustand als Paar vor, wobei der erste Teil die eigentliche Steuerung darstellt (also die Position im Programmablauf) und der zweite Teil als Speicher dient. Zustandsübergänge können so systematischer beschrieben werden.

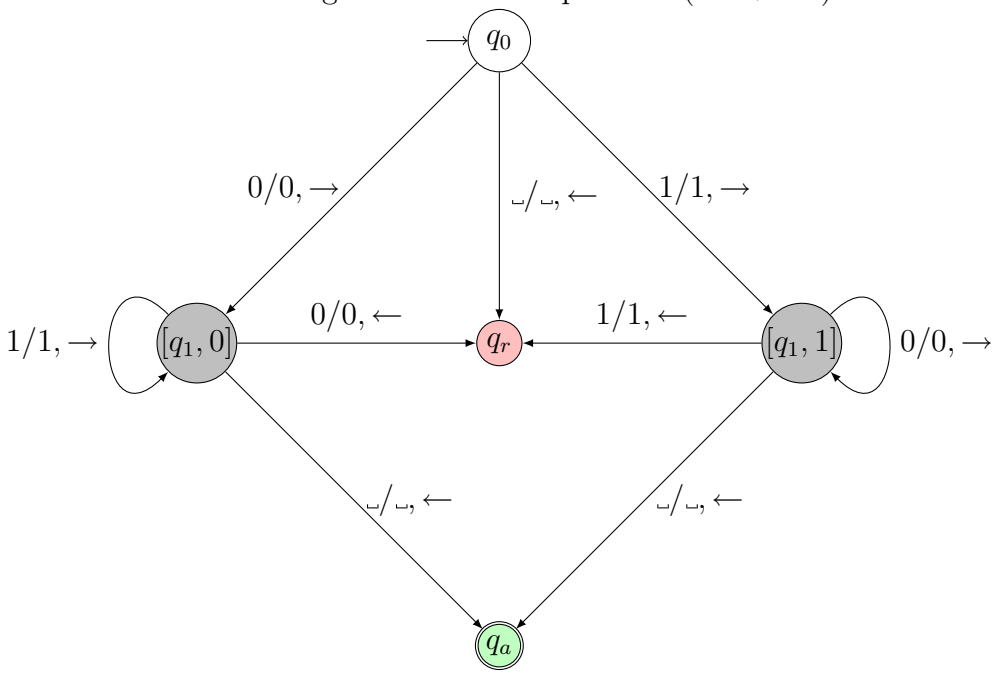
Beispiel 8. *Eine Turingmaschine, die die Sprache $L(10^* + 01^*)$ akzeptiert. Die Turingmaschine liest das erste Symbol, dann geht sie durch die ganze Eingabe und prüft, ob es später noch einmal vorkommt. Falls nicht, dann wird das Wort akzeptiert, sonst wird es abgelehnt. Abbildung 6 zeigt das Übergangsdiagramm für die Maschine*

$$M = (\{0,1\}, \{0,1,\textvisiblespace\}, \{q_0, q_a, q_r, [q_1,0], [q_1,1]\}\delta, q_0, q_a, q_r)$$

wobei δ durch die folgende Tabelle beschrieben wird:

δ	0	1	$\sqcup$
q_0	$([q_1, 0], R)$	$([q_1, 1], R)$	q_r
$[q_1, 0]$	(q_r, R)	$([q_1, 0], R)$	(q_a, R)
$[q_1, 1]$	$([q_1, 1], R)$	(q_r, R)	(q_a, R)
q_a	-	-	-
q_r	-	-	-

Abbildung 6: TM für die Sprache $L(10^* + 01^*)$



2.5.3 Abhaken von Symbolen

Bei Sprachen, die sich durch Wiederholung von Strings auszeichnen, z.B. $L_1 = \{w\sharp w \mid w \in \Sigma^*\}$ und $L_2 = \{a^i b^i \mid i \in \mathbb{N}\}$ ist es manchmal notwendig „mitzuzählen“ wenn man feststellen will, ob ein eingegebenes Wort zur Sprache gehört. Man kann dies erleichtern, indem man das Alphabet erweitert und für jedes Zeichen aus dem Eingabealphabet zusätzlich eine „abgehakte“ Variante im Bandalphabet bereithält.

- Man erweitert das Bandalphabet Γ um zusätzliche Symbole, indem man für jedes Σ-Zeichen a eine „abgehakte “ Variante, also z.B. ein Zeichen $\overline{a}$ hinzufügt. $\Gamma = \Sigma \cup \{\overline{a} \mid a \in \Sigma\}$

- Einige Zustände werden als Paare beschrieben, wobei der erste Teil den Stand der Abarbeitung im Algorithmus kodiert und der zweite Teil als

Speicherplatz für ein eingelesenes Symbol dient.

Beispiel 9. *Turingmaschine für* $L = \{w\sharp w \mid w \in L(\{0,1\}^+)\}$*, siehe auch Abbildung 7*

q_0 *Ist der Startzustand, das erste Zeichen* X *der Eingabe wird gelesen und markiert. Übergang nach* (q_1, X)*. Bei Ungültigkeit der Eingabe, also falls* $X \notin \{0,1\}$ *Übergang zu* q_r*.*

(q_1, X) *Der Kopf der Turingmaschine bewegt sich solange nach rechts bis er* $\sharp$ *gefunden hat, dann geht die Turingmaschine in* (q_2, X) *über, enthält die Eingabe kein* $\sharp$ *Übergang zu* q_r*.*

(q_2, X) *Nach rechts gehen bis zum ersten unmarkierten Zeichen hinter* $\sharp$*, vergleichen mit dem gespeicherten Zeichen* X*, bei Übereinstimmung, Übergang nach* (q_3, X)*, bei Nichtübereinstimmung zu* q_r*.*

(q_3, X) *Nach links gehen bis zum* $\sharp$*, Wechsel in Zustand* $(q_4, \sharp)$*.*

$(q_4, \sharp)$ *falls es keine unmarkierten Zeichen links von* $\sharp$ *gibt, gehe in* q_7 *über; falls es noch welche gibt, nach* q_5*.*

q_5 *Gehe zum am weitesten links stehenden unmarkierten Zeichen, wenn das erreicht ist, Übergang nach* q_6*.*

q_6 *Einlesen des Zeichens* X*, markieren und Übergang nach* (q_1, X)*.*

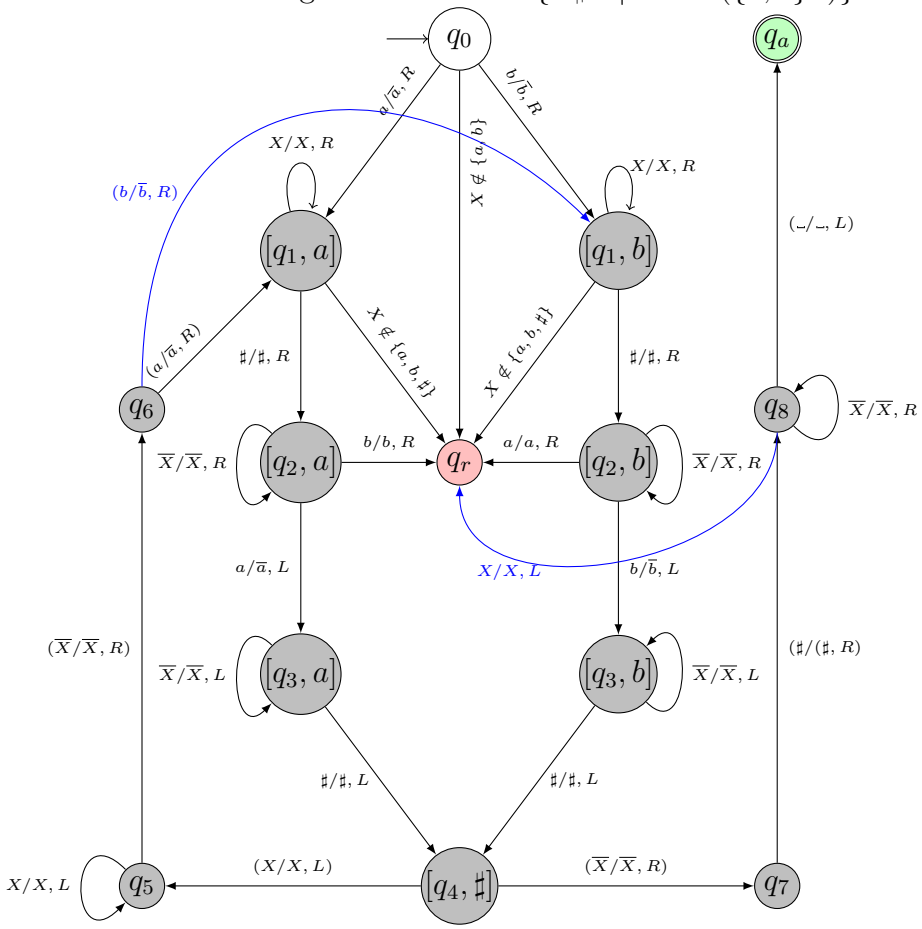
q_7 *Lese* $\sharp$*, gehe nach rechts und gehe zu* q_8 *über.*

q_8 *Gehe nach rechts, lies alle markierten Zeichen, wenn noch unmarkierte Zeichen da sind, Übergang zu* q_r *sonst zu* q_a*.*

2.5.4 Verschiebungen

Um Platz auf dem Band zu schaffen, ist es manchmal hilfreich, alle von Leerzeichen verschiedenen Zeichen auf dem Band um eine feste Anzahl x nach rechts zu verschieben. Das Verfahren dazu ist einfach. Man legt sich mit Hilfe der Zustände einen Zwischenspeicher an, in dem man von der einen Seite Zeichen hinein-, auf der anderen Seite herausschieben kann. Wir dürfen voraussetzen, daß dabei keine Leerzeichen innerhalb des zu verschiebenden Teils erlaubt sind.

Abbildung 7: TM für $L = \{w\sharp w \mid w \in L(\{0,1\}^+)\}$



Beispiel 10. *Sei* $\Sigma = \{a, b\}$ *und* $M = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$ *mit*

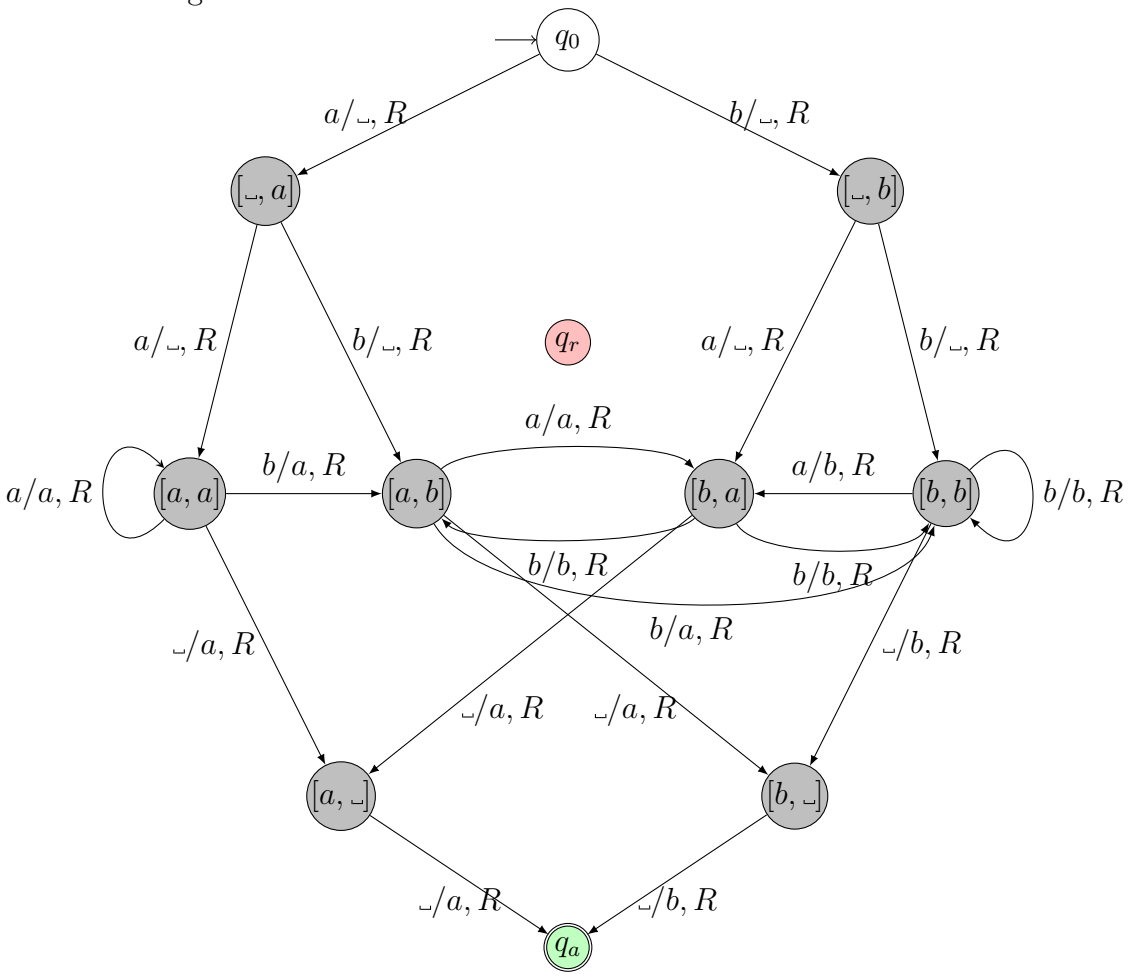
$$Q = \{q_0, [\sqcup, a], [\sqcup, b], [a, a], [a, b][b, a], [b, b][a, \sqcup], [b, \sqcup][q_A], [q_R]\}$$

als Menge der Zustände. Das Übergangsdiagramm in Abbildung 8 gehört zu einer Turingmaschine M*, die den Bandinhalt um zwei Zeichen nach rechts verschiebt.*

2.5.5 Unterprogramme

Damit man bereits programmierte Teile weiterverwenden kann, braucht man die Möglichkeit eine Turingmaschine M_1 als Teil in andere Turingmaschine M_2 einzubauen. Beim Übergang zum Unterprogramm muss der gewünschte

Abbildung 8: Bandinhalt um zwei Zeichen nach rechts verschieben



Anfangszustand hergestellt werden und die beiden Haltezustände müssen zu einem vorher festgelegten Rückkehrzustand führen. Die Aufrufe können dann sowohl rekursiv als auch nicht rekursiv erfolgen.

Beispiel 11. *Es soll die Multiplikation von natürlichen Zahlen berechnet werden. Dabei wird eine Zahl m durch den String 0^m und das Paar (m, n) durch 0^m10^n dargestellt.*

$M =$

„Auf Eingabe von 0^m10^n

1. Hinter das rechte Ende wird eine Eins gesetzt.

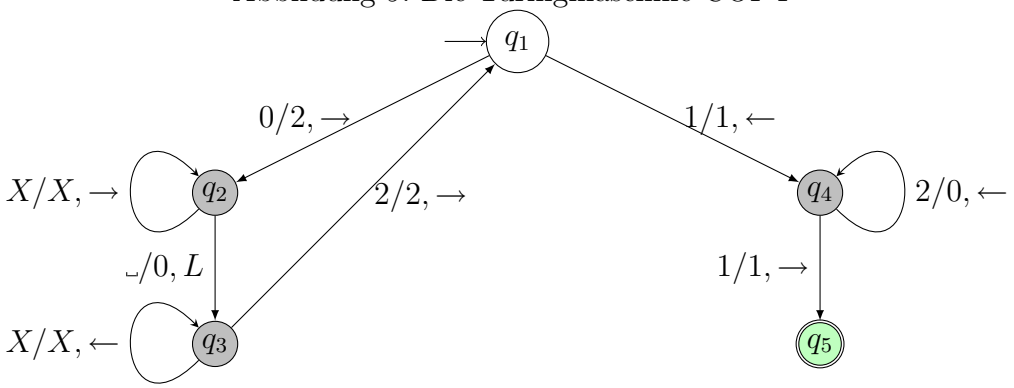
2. Für jede der Nullen aus dem vorderen Block wird der hintere Block aus Nullen einmal kopiert und die vordere Null gelöscht.

3. Wenn keine vordere Null mehr vorhanden ist, steht das Ergebnis hinter der zweiten Eins, die Maschine akzeptiert.

“

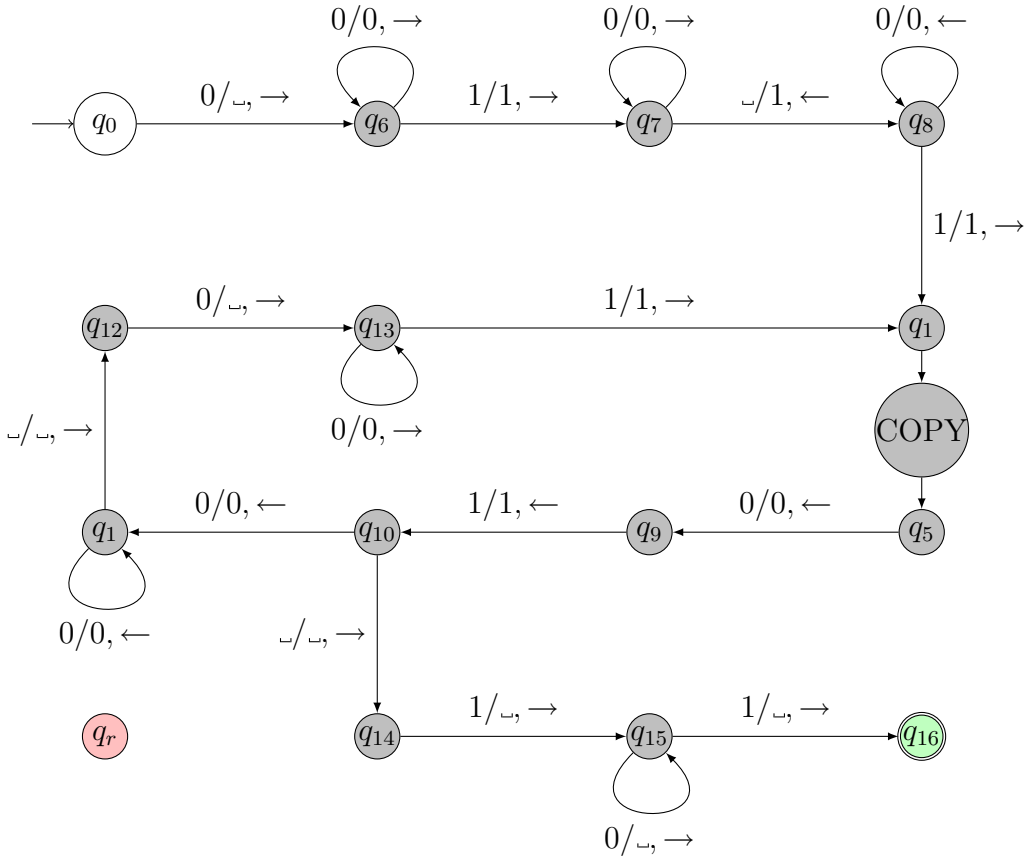
Wir definieren zunächst die Maschine COPY. COPY startet in der Konfiguration $0^m 1 q_1 0^n 1 0^i$ und endet mit der Konfiguration $0^m 1 q_5 0^n 1 0^{i+n}$.

Abbildung 9: Die Turingmaschine COPY



Anschließend setzen wir das ganze Programm zusammen.

Abbildung 10: Ganzes Programm für Multiplikation



3 Entscheidbarkeit

Bisher haben wir Turingmaschinen als ein Modell für universelle Rechenmaschinen kennengelernt. Wir haben den Begriff des Algorithmus definiert, indem wir uns auf Turingmaschinen gestützt haben, wozu uns die Church-Turing-These berechtigt. Mit dem λ-Kalkül haben wir ein weiteres Modell zur Berechnung von Funktionen betrachtet. Jetzt wollen wir uns den Grenzen algorithmischen Problemlösens zuwenden. Uns interessieren die Art und der Umfang der Probleme, die mit Algorithmen lösbar sind. Dabei ist es für eine Reihe von Problemen offensichtlich, dass und wie man sie lösen kann. Die Tatsache, dass es (durchaus praktische) Probleme gibt, für die das nicht möglich ist, ist vielleicht weniger offensichtlich. Das gilt zum Beispiel auch für Hilberts zehntes Problem (konnte erst in den 1970er Jahren bewiesen werden).

Wenn wir keinen Einfluss darauf haben, ob ein Problem lösbar ist oder nicht (wenn also die Lösbarkeit im Problem begründet liegt und nicht in der Mächtigkeit der verwendeten Computer) warum sollte man sich damit beschäftigen, ob es lösbar ist oder nicht, wenn man doch eine Lösung (praktikable) finden muss? Dafür gibt es hauptsächlich zwei Gründe:

1. Wenn man festgestellt hat, dass man das Problem, so wie es gestellt ist, nicht lösen kann, muss man es vereinfachen oder abändern (solche Abschwächungen müssen unter Umständen gerechtfertigt werden), um dann für die neue Version eine Lösung zu finden. Die Untersuchung zur Lösbarkeit liefert dabei oft auch einen Hinweis, wie man abschwächen muss.

2. Der zweite Grund ist eher kulturell. Selbst wenn man mit Problemen zu tun hat, die offenbar lösbar sind, hilft ein Blick auf die unlösbaren Dinge, um Anregungen für die Lösung zu bekommen.

Schließlich ist es auch aus philosophischer Sicht interessant, sich mit den Grenzen der maschinellen Berechenbarkeit zu beschäftigen.

Wir erinnern uns an folgende Definitionen: Eine Menge M heißt Turing-akzeptierbar, wenn es eine Turingmaschine gibt, die akzeptiert und anhält wenn die Eingabe zu M gehört. M heißt entscheidbar, wenn es eine Turingmaschine gibt, die immer (bei jeder Eingabe) anhält. M ist aufzählbar, wenn es eine Turingmaschine gibt, die alle Elemente von M aufzählt.

3.1 Entscheidbare Mengen

Im letzten Semester haben Sie verschiedene Klassen von formalen Sprachen sowie die zugehörigen Maschinenmodelle kennengelernt. Jedes dieser Modelle

lässt sich durch Turingmaschinen simulieren. Bei unserer Aufzählung beginnen wir mit regulären Sprachen.

3.1.1 Probleme regulärer Sprachen

Sei $A_{DEA} = \{< B, w > |\ B \text{ ist ein endlicher Automat, der } w \text{ akzeptiert}\}$

Satz 4. *Die Menge A_{DEA} ist entscheidbar.*

Beweis: Wir bauen eine Turingmaschine M, die A_{DEA} entscheidet.
$M =$

„Auf Eingabe $< B, w >$ mit DEA B und Wort w:

1. Simuliere B auf Eingabe w.
2. Falls B in einem akzeptierenden Zustand anhält – *accept*;
   falls B in einem nicht akzeptierenden Zustand anhält– *reject*.

“

Ein paar Details zur Turingmaschine M: Wir werfen zuerst einen Blick auf die Eingabe. Sie ist die Darstellung eines endlichen Automaten B zusammen mit einem Wort w. Eine mögliche Darstellung für B ist z.B. die Liste der fünf Komponenten $Q, \Sigma, \delta, q_0, F$. Als erstes prüft M die Eingabe daraufhin, ob sie die richtige Form hat und lehnt ab, falls nicht. Danach führt M die Simulation von B direkt aus. Sie verfolgt den aktuellen Zustand und die aktuelle Position bei der Abarbeitung von w, indem sie diese Informationen aufs Band schreibt. Am Anfang ist der Zustand von B ihr Startzustand und ihre Eingabeposition ist das am weitesten links stehende Symbol von w. Zustand und Position werden entsprechend der Überführungsfunktion δ aktualisiert. Wenn M die Bearbeitung des letzten Zeichens von w abgeschlossen hat, dann akzeptiert M das Paar (B, w), wenn B in einem Endzustand ist – sonst wird das Paar zurückgewiesen. □

Satz 5. *A_{NEA} ist eine entscheidbare Menge.*

Beweis: Konvertieren des NEA in einen DEA. □

Satz 6.

$$A_{REX} = \{< R, w > |\ R \textit{ ist regulärer Ausdruck, der } w \textit{ generiert}\}$$

ist entscheidbar.

Beweis: Konvertieren von R in einen endlichen Automaten. □

Die Sätze 4, 5 und 6 illustrieren, dass für Entscheidbarkeitszwecke das Angeben einer Turingmaschine für DEA, NEA und reguläre Ausdrücke äquivalent ist, da eine Turingmaschine jedes Modell in eines der anderen „übersetzen“ kann.

Wir kommen nun zu einem anderem Problem regulärer Sprachen, dem der Leerheit. In den bisher betrachteten Problemen mussten wir feststellen, ob ein gegebener Automat ein bestimmtes Wort akzeptiert; jetzt müssen wir untersuchen, ob es überhaupt ein Wort gibt, das er akzeptiert. Wir definieren:

$$E_{DEA} = \{< A >| \ A \text{ ist ein DEA und } L(A) = \emptyset\}$$

Satz 7. *E_{DEA} ist entscheidbar.*

Beweis: Ein endlicher Automat akzeptiert eine Zeichenkette genau dann, wenn es einen Pfad vom Startzustand zu einem akzeptierenden Zustand gibt. $M =$

„Auf Eingabe $< A >$

1. Markiere den Startzustand von A.
2. Wiederhole bis keine neuen Zustände markiert werden:
   Markiere jeden Zustand, der von einem bereits markierten Zustand aus erreichbar ist.
3. Falls kein Endzustand markiert ist, *reject*; sonst *accept*.

“

□

Satz 8. *Sei $EQ_{DEA} = \{< A, B >| \ A$ und B sind DEA und $L(A) = L(B)\}$. EQ_{DEA} ist entscheidbar.*

Beweisidee: Wir bauen aus A und B einen neuen Automaten C, der ein Wort genau dann akzeptiert, wenn es von A, aber nicht von B oder von B, aber nicht von A akzeptiert wird. Falls A und B dieselbe Sprache akzeptieren, dann akzeptiert C gar nichts. Die Sprache von C ist $L(A) \triangle L(B)$

1. Konstruiere C.
2. Lasse M aus dem Beweis von Satz 7 auf C laufen.
3. *accept*, wenn M akzeptiert; sonst, *reject*.

3.1.2 Entscheidbare Probleme kontextfreier Sprachen

Satz 9. *Sei*

$$A_{CFG} = \{< G, w > | \; G \textit{ist eine kontextfreie Grammatik, die } w \textit{ erzeugt}\}.$$

Die Menge A_{CFG} ist entscheidbar.

Die naive Idee ist, alle Ableitungen aus G auszuprobieren, ob sie zu w führen. Da es unendlich viele solcher Ableitungen gibt, funktioniert das nicht, falls G das Wort w nicht erzeugt. Man bekommt auf diesem Weg nur einen Akzeptierer für G. Um daraus einen Entscheider zu konstruieren, muss man sicherstellen, dass die Turingmaschine nur endlich viele Ableitungen ausprobiert.
Für diesen Zweck machen wir uns die Chomsky-Normalform(CNF) zunutze. Bei einer Grammatik in CNF haben alle Produktionen die Form $A \rightarrow BC$ oder $A \rightarrow a$ und jede Ableitung von w hat $2n$–1 Schritte, wobei n die Länge von w ist. Man muss also nur alle Ableitungen der Länge $2n - 1$ testen, um festzustellen, ob w von G erzeugt wird. Davon existieren aber nur endlich viele.

Wie aus der Vorlesung „Automaten und Sprachen“ bekannt ist, gibt es einen Algorithmus, mit dem man jede kontextfreie Grammatik G in endlich vielen Schritten in eine Grammatik in CNF überführen kann.

Beweis: Die Turingmaschine S, die A_{CFG} entscheidet, sieht aus wie folgt:
$S =$

„Auf Eingabe $< G, w >$, wobei G eine CFG und w ein Wort ist:

1. Konvertiere G in eine äquivalente Grammatik in CNF.
2. Zähle alle Ableitungen mit $2n - 1$ Schritten auf, wobei $n = |w|$, außer wenn $n = 0$, dann betrachte alle Ableitungen der Länge 1.
3. Falls eine der Ableitungen w erzeugt, *accept*– sonst *reject*.

“

□

Das Testen, ob eine kontextfreie Grammatik ein bestimmtes Wort akzeptiert, ist verwandt mit dem Problem der Kompilierung. Der Algorithmus von S ist sehr ineffizient und würde in der Praxis niemals benutzt werden. Aber er ist einfach zu beschreiben und für Laufzeit und Platzverbrauch eines Algorithmus' werden wir uns erst in einem späteren Kapitel interessieren. Noch

schneller wäre eine Turingmaschine, die das Verfahren von Cocke-Younger-Kasami benutzt. Effizienz beschäftigt uns aber erst im Abschnitt Komplexität.
Wie in der Vorlesung über Automaten und Sprachen gezeigt wurde, kann man Verfahren angeben, wie man aus einem PDA A eine kontextfreie Grammatik entwickelt, die die von A akzeptierte Sprache erzeugt und umgekehrt einen PDA angeben, der dieselbe Sprache akzeptiert wie eine vorgegebene kontextfreie Grammatik. Das bedeutet, dass alles, was wir über die Entscheidbarkeit von kontextfreien Sprachen sagen können auch für Kellerautomaten gilt.

Folgerung: Die Menge

$$A_{PDA} = \{< A, w > | \ A \text{ ist ein Kellerautomat und } w \text{ ein Wort}\}$$

ist entscheidbar.

Nun betrachten wir das Leerheitsproblem für kontextfreie Sprachen.

Satz 10. *$E_{CFG} = \{< G > | \ G$ ist kontextfreie Grammatik und $L(G) = \emptyset\}$ ist entscheidbar.*

Beweis: Um einen Algorithmus für dieses Problem zu finden, könnte man die Idee haben, die Maschine S aus Satz 9 zu benutzen, die feststellt, ob ein bestimmtes Wort von einer Grammatik erzeugt wird. Aber um festzustellen, ob $L(G)$ leer ist, müssten wir alle Wörter w nacheinander ausprobieren können. Da es aber unendlich viele Wörter gibt, können wir Satz 9 nicht verwenden (bzw. eine entsprechende Turingmaschine). Wir brauchen also einen anderen Ansatz. Um herauszufinden, ob die Sprache zu einer Grammatik leer ist, muss man testen, ob vom Startsymbol aus eine Kette von Terminalsymbolen erzeugt werden kann. Der folgende Algorithmus geht sogar noch allgemeiner vor; er testet für alle Nichterminalsymbole, ob sie zur Erzeugung von Terminalen führen.

$R =$

„Auf Eingabe $< G >$, wobei G eine kontextfreie Grammatik ist:

1. Markiere alle Terminale von G.
2. Wiederhole bis keine neuen Nichtterminale markiert werden:
3. - Markiere jedes Nichtterminal A, für das G eine Produktion $A \rightarrow U_1 U_2 \ldots U_k$ enthält und $U_1, \ldots, U_k$ bereits markiert sind.
4. Wenn das Startsymbol nicht markiert ist, *accept*; sonst *reject*.

“

□

Als nächstes wollen wir entscheiden, ob zwei kontextfreie Grammatiken dieselbe Sprache generieren. Sei

$$EQ_{CFG} = \{< G, H > |\ G, H \text{ kontextfreie Grammatiken und } L(G) = L(H)\}.$$

In Satz 8 haben wir einen Algorithmus angegeben, der das analoge Problem für endliche Automaten entscheidet. Wir haben die Entscheidungsprozedur für E_{DEA} benutzt um zu zeigen, dass EQ_{DEA} entscheidbar ist. Da wir für E_{CFG} ebenfalls eine Entscheidungsprozedur haben, könnte man auf die Idee kommen, eine ähnliche Strategie zu verwenden um zu beweisen, dass EQ_{CFG} entscheidbar ist. Diese Idee funktioniert aber nicht, da kontextfreie Sprachen unter Durchschnittsbildung oder Mengendifferenz *nicht* abgeschlossen sind. Tatsächlich ist EQ_{CFG} nicht entscheidbar, wie wir später im Kapitel über unentscheidbare Probleme noch beweisen werden.

Betrachten wir nun das Wortproblem für kontextfreie Sprachen. Wie bei regulären Sprachen ist auch das Wortproblem für kontextfreie Sprachen entscheidbar:

Satz 11. *Jede kontextfreie Sprache ist entscheidbar.*

Beweisidee: Sei A eine kontextfreie Sprache. Unser Ziel ist es zu zeigen, dass A entscheidbar ist. Eine naive (und schlechte) Idee wäre es, einen PDA für A direkt in eine Turingmaschine umzuwandeln. Das sollte möglich sein, da das Band auch als Stack behandelt werden kann. Der PDA für A kann nichtdeterministisch sein, aber wir wissen bereits, dass für jede nichtdeterministische Turingmaschine auch eine deterministische Turingmaschine existiert, die die gleiche Sprache akzeptiert und wir haben auch einen Algorithmus, der eine solche erzeugt. Allerdings gibt es eine Schwierigkeit. Es kann Pfade des PDA geben, wo in unendlicher Folge auf den Stack geschrieben und von ihm gelesen wird. Die entsprechende simulierende Turingmaschine hat dann ebenfalls nichthaltende Zweige und kann damit kein Entscheider sein. Stattdessen verwenden wir den Entscheider S für die Sprache A_{CFG} aus Satz 9.

Beweis: Sei G eine kontextfreie Grammatik für A.
$M_G =$

„Auf Eingabe w

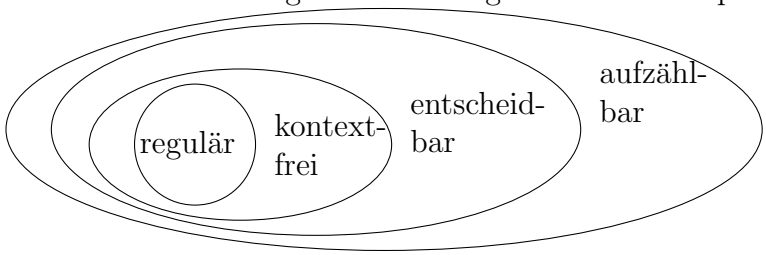
1. Starte die Turingmaschine S auf der Eingabe $< G, w >$.
2. *accept* wenn S akzeptiert– *reject* sonst.

“

□

Die folgende Abbildung 11 zeigt die Relation zwischen den vier hauptsächlich bisher von uns betrachteten Klassen von Problemen bzw. Sprachen.

Abbildung 11: Beziehung zwischen den Sprachklassen



3.2 Das Halteproblem

In diesem Abschnitt werden wir eine der philosophisch wichtigsten Aussagen der Berechenbarkeitstheorie kennenlernen. Es gibt Probleme, die algorithmisch nicht lösbar sind. Computer erscheinen so mächtig, dass man denken könnte, dass sie früher oder später mit jedem Problem fertig würden. Die zentrale Aussage, die hier präsentiert werden soll, zeigt, dass Computer in einer sehr grundlegenden Weise beschränkt sind. Nun könnte man denken, dass die unlösbaren Probleme erkünstelt, weltfremd oder irgendwie esoterisch sind. Aber das ist nicht der Fall. Sogar einige ganz gewöhnliche Probleme, die Leute sehr gern automatisch lösen würden, fallen in diese Gruppe.

In einem Typ unlösbarer Probleme werden ein Computerprogramm und eine genaue Beschreibung dessen, was das Computerprogramm tun soll (z.B. eine Liste mit Zahlen sortieren) gegeben. Man soll überprüfen, ob das Programm die Aufgabe erfüllt (d.h. ob das Programm korrekt ist). Da sowohl das Programm als auch die Spezifikation mathematisch genau beschreibbare Objekte sind, sollte man erwarten, dass man den Prozess der Verifikation

automatisieren kann. D.h. man übergibt beide Objekte einem speziellen Computerprogramm zur Auswertung. Und man wird enttäuscht! Das allgemeine Problem zur Software-Verifikation ist nicht durch einen Computer lösbar.

In diesem und dem nächsten Kapitel werden wir noch einige weitere unlösbare Probleme kennenlernen. Das Ziel ist dabei, ein Gefühl dafür zu bekommen, welche Art von Problemen unlösbar sind und Beweistechniken für Unentscheidbarkeit zu lernen.

3.2.1 Formulierung des Halteproblems

In Anlehnung an die Bezeichnung für Entscheidbarkeitsprobleme bezeichnen wir die zu entscheidende Menge als A_{TM}, wobei:

$$A_{TM} = \{< M, w > |\ M \text{ ist Turingmaschine und } M \text{ akzeptiert } w\}$$

Satz 12. *A_{TM} ist unentscheidbar.*

Bevor wir an den Beweis gehen, überzeugen wir uns davon, dass A_{TM} aufzählbar ist (also Turing-akzeptierbar). Damit beweisen wir gleichzeitig, dass Entscheider weniger mächtig als Akzeptierer sind (vergleiche Bemerkung nach Definition 6). Wir konstruieren eine Turingmaschine U, die A_{TM} akzeptiert wie folgt:
$U =$

„Auf Eingabe $< M, w >$, mit Turingmaschine M und Wort w:

1. Simuliere M auf w.
2. Falls M das Wort w akzeptiert, accept; falls M ablehnt, reject.

“

Die Turingmaschine U gerät in eine Schleife, wenn M bei Eingabe von w in eine Schleife gerät, ist also kein Entscheider. Falls der Algorithmus eine Möglichkeit hätte festzustellen, dass M auf w niemals anhält, würde er das Paar ablehnen. Daher heißt A_{TM} auch das Halteproblem. Wie wir zeigen werden, gibt es aber für keinen Algorithmus die Möglichkeit festzustellen, ob eine beliebige Maschine auf einer Angabe jemals anhalten wird.

U ist aber auch für sich selbst interessant, sie ist ein Beispiel für eine universelle Turingmaschine, wie sie von Turing vorgeschlagen wurde. Universell heißt sie deswegen, weil sie die Arbeit einer beliebigen Turingmaschine aus deren Beschreibung simulieren kann. Die universelle Turingmaschine spielte eine wichtige Rolle als Anstoß für die Entwicklung von Computern mit gespeicherten Programmen.

3.2.2 Die Methode der Diagonalisierung

Für den Beweis der Unentscheidbarkeit werden wir eine Technik von Georg Cantor, einem Mathematiker des 19. Jahrhunderts, verwenden. Cantor beschäftigte sich mit der Messung der Größe von unendlichen Mengen. Unter anderem wollte er wissen, wie man bei zwei unendlich großen Mengen feststellen kann, welche von beiden größer ist. Bei endlichen Mengen zählt man einfach die Elemente und deren Anzahl ist die Größe der Menge; bei unendlichen Mengen wird man damit nicht fertig. Man kann zum Beispiel die Menge der ganzen Zahlen und die Menge der Zeichenketten über $\{0, 1\}$ betrachten. Beide Mengen sind unendlich groß. Die Frage ist nun, ob man in irgendeiner Weise feststellen kann, ob eine von beiden „größer“ in einem vernünftig definierten Sinn ist, und wenn ja, wie. Cantor machte einen eleganten Vorschlag: Er stellte fest, dass endliche Mengen genau dann die gleiche Größe haben, wenn man ihre Elemente paarweise einander zuordnen konnte und übertrug diese Idee von „gleich groß“ auf unendliche Mengen.

Definition 11. *Seien A und B Mengen und $f : A \to B$ eine Funktion.*

1. *f heißt eineindeutig oder injektiv, wenn es keine Elemente von A gibt, die unter f dasselbe Bild haben, $f(a) \neq f(a')$ für $a \neq a'$.*

2. *f heißt Funktion auf B oder surjektiv, wenn für alle $b \in B$ ein Element $a \in A$ existiert, so dass $f(a) = b$.*

3. *A und B haben dieselbe Größe, wenn es eine eineindeutige Funktion f von A auf B gibt. f heißt dann auch Bijektion.*

Bei einer Bijektion wird jedem Element von A ein eindeutig bestimmtes Element von B zugeordnet, und für jedes Element von B gibt es ein Element von A, dessen Bild es ist.

Eine Bijektion ist also einfach ein Weg, Elemente von A und B einander paarweise zuzuordnen.

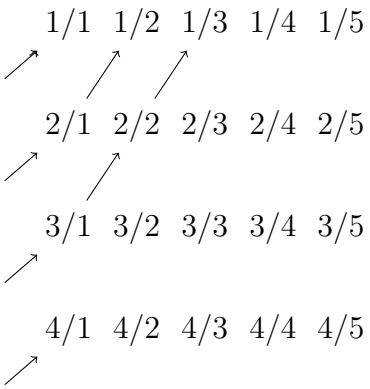
Beispiel 12. *Die Menge $\mathbb{N}$ der natürlichen Zahlen und die Menge $\mathbb{E}$, der geraden Zahlen sind gleich groß. Die Funktion $f : \mathbb{N} \to \mathbb{E}$ mit $n \mapsto 2n$ ist eine Bijektion.*

Definition 12. *Eine Menge A heißt abzählbar, wenn sie endlich ist oder dieselbe Größe wie $\mathbb{N}$ hat.*

Beispiel 13. *Sei $\mathbb{Q}^+$ die Menge der positiven rationalen Zahlen, d.h.*

$$\mathbb{Q}^+ = \{\frac{m}{n} \mid m, n \in \mathbb{N} \setminus \{0\}\}.$$

Intuitiv würde man wegen $\mathbb{N} \subseteq \mathbb{Q}^+$ vermuten, dass $\mathbb{Q}^+$ größer als $\mathbb{N}$ ist. Wir geben eine Bijektion an, um zu zeigen, dass sie gleich groß sind. Zunächst ordnen wir die rationalen Zahlen in der folgenden Matrix an, so dass dabei Zahlen in derselben Zeile jeweils den gleichen Zähler, in derselben Spalte jeweils den gleichen Nenner haben.



Diese Matrix lässt sich in eine Liste transformieren, indem wir die Zahlen entlang der Nebendiagonalen unter Auslassen von Wiederholungen aufschreiben und wir bekommen:

$$\frac{1}{1}, \frac{2}{1}, \frac{1}{2}, \frac{3}{1}, \frac{1}{3}, \frac{4}{1}, \frac{3}{2}, \frac{2}{3}, \frac{1}{4}, \ldots$$

Es gibt aber auch Mengen, die „zu groß“ sind, um abzählbar zu sein– sogenannte überabzählbare Mengen. Beispiele dafür sind die Reellen Zahlen und die Menge der Funktionen auf den natürlichen Zahlen. Dabei sind reelle Zahlen solche Zahlen, die eine (möglicherweise unendliche) Dezimaldarstellung besitzen wie z.B. $\pi = 3.1415926$ oder $\sqrt{2} = 1.4142135$.

Satz 13. *$\mathbb{R}$ ist überabzählbar.*

Beweis: (mit Hilfe der Diagonalisierungsmethode, indirekt)
Wir nehmen an, wir hätten eine Bijektion $f : \mathbb{N} \to \mathbb{R}$ und beweisen, dass es Zahlen gibt, die davon nicht erfasst werden. Sei f die Liste

$$(1, \pi), (2, 55.555), (3, 0.1234), (4, 0.5000) \ldots$$

einer solchen hypothetischen Bijektion. Wir konstruieren eine Zahl $x = 0.a_1 a_2 \ldots a_i \ldots$, die nicht in der Liste auftaucht, in dem wir die i-te Nachkommastelle wie folgt definieren [2] durch

$$a_i := \begin{cases} 2 & \text{falls die } i\text{-te Stelle von } f(i) \text{ ungleich 2 ist,} \\ 3 & \text{sonst.} \end{cases}$$

[2] Es geht auch allgemeiner, dann muss man aber beachten, dass es Zahlen mit verschiedenen Dezimaldarstellungen gibt wie $1.999\ldots = 2.000\ldots$

Die so definierte Zahl x unterscheidet sich an jeder Stelle von mindestens einer Zahl in der Liste, kann also selbst nicht in der Liste gewesen sein. Da wir eine solche Zahl konstruieren konnten, kann f nicht surjektiv und damit auch keine Bijektion gewesen sein. □

Der obige Satz hat eine wichtige Anwendung in der Berechenbarkeitstheorie. Er zeigt, dass manche Sprachen nicht entscheidbar oder nicht einmal Turing-akzeptierbar sind, da es überabzählbar viele Sprachen, aber nur abzählbar viele Turingmaschinen gibt. Da nämlich jede Turingmaschine jeweils nur eine Sprache akzeptiert und es mehr Sprachen als Turingmaschinen gibt, muss es auch Sprachen geben, die nicht von einer Turingmaschine akzeptiert werden.

Folgerung: Es gibt Sprachen, die nicht Turing-akzeptierbar sind.

Beweis: Für den Beweis formulieren wir zunächst drei Teilziele:

1. Die Menge aller Turingmaschinen ist abzählbar.
2. Die Menge aller Turingmaschinen ist aufzählbar.
3. Die Menge aller Sprachen ist überabzählbar.

1. Um zu zeigen, dass die Menge aller Turingmaschinen abzählbar ist, stellt man zunächst fest, dass die Menge aller Zeichenketten Σ^* über einem Alphabet Σ abzählbar ist. Mit nur endlich vielen Zeichenketten jeder Länge können wir eine Aufzählung von Σ^* angeben, indem wir alle Zeichenketten der Länge 0, der Länge 1, der Länge 2 usw. aufschreiben.

2. Die Menge der Turingmaschinen ist dann ebenfalls aufzählbar, da sich jede Turingmaschine M durch eine endliche Zeichenkette $< M >$ darstellen lässt. Lassen wir von den allgemeinen Zeichenketten also alle weg, die keine gültige Kodierung einer Turingmaschine darstellen, erhalten wir eine Aufzählung aller Turingmaschinen.

3. Um zu zeigen, dass die Menge aller Sprachen überabzählbar ist, überlegt man sich zuerst, dass die Menge $\mathbb{B}$ aller unendlichen binären Folgen überabzählbar ist (Übungsaufgabe).

   Sie $\mathcal{L}$ die Menge aller Sprachen über Σ. Die Überabzählbarkeit von $\mathcal{L}$ zeigt man, indem man eine Bijektion von $\mathcal{L}$ zu $\mathbb{B}$ angibt. Sei

$$\Sigma^* = \{s_1, s_2, \ldots\}$$

die Menge aller Wörter über Σ. Jeder Sprache $A \in \mathcal{L}$ kann man eine Folge χ_A zuordnen: [3], die so genannte charakteristische Folge mit

$$\chi_A(s_i) = \begin{cases} 1 & \text{falls } s_i \in A \\ 0 & \text{sonst.} \end{cases}$$

Die Zuordnung $f : \mathcal{L} \to \mathbb{B}$ mit $A \mapsto \chi_A$ ist injektiv und surjektiv, also eine Bijektion. Da $\mathbb{B}$ überabzählbar ist, ist auch $\mathcal{L}$ überabzählbar.

Damit haben wir gezeigt, dass es keine Bijektion zwischen der Menge aller Turingmaschinen und der Menge aller Sprachen geben kann. Daraus folgt, dass es Sprachen geben muss, die von keiner Turingmaschine akzeptiert werden können. □

3.2.3 Unentscheidbarkeit des Halteproblems

Wir können nun die Unentscheidbarkeit von A_{TM} beweisen.

Satz 14. *$A_{TM} = \{< M, w > |$ M ist Turingmaschine und M akzeptiert $w\}$ ist nicht entscheidbar.*

Beweis: (indirekt) Wir nehmen an, A_{TM} wäre entscheidbar. Sei H ein Entscheider für A_{TM}. Dann akzeptiert H die Eingabe $< M, w >$, falls M auf w anhält und lehnt die Eingabe ab, falls M auf Eingabe w ablehnt oder nicht anhält.

Wir konstruieren eine Turingmaschine D, die H als Unterprogramm enthält. Bei Eingabe von M wird H mit der Eingabe $< M, < M >>$ gestartet.
$D =$

„Auf Eingabe $< M >$, wobei M eine Turingmaschine ist:

1. Starte H auf der Eingabe $< M, < M >>$.
2. Gebe das Gegenteil von der Ausgabe von H aus, d.h. wenn H akzeptiert–reject wenn H ablehnt–accept.

“

Man sollte sich an dieser Stelle nicht von der Idee verwirren lassen, eine Maschine mit ihrer eigenen Beschreibung als Eingabe laufen zu lassen. Dieser

[3] Zum Beispiel hat über $\Sigma = \{0, 1\}$ mit $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \ldots\}$ die Sprache $A = \{0, 00, 01, 000, 001 \ldots\}$ die charakteristische Folge $\chi_A = (0, 1, 0, 1, 1, 0, 0, 1, 1, 1 \ldots)$.

Abbildung 12: Der Eintrag bei (i, j) ist accept, falls M_i die Eingabe $< M_j >$ akzeptiert.

	$< M_1 >$	$< M_2 >$	$< M_3 >$	$< M_4 >$	...
M_1	accept		accept		
M_2	accept	accept	accept	accept	
M_3				...	
M_4	accept	accept			
$\vdots$		$\vdots$			

Vorgang ist ähnlich zu dem, ein Programm mit seinem eigenen Quelltext als Eingabe laufen zu lassen. So etwas kommt in der Praxis durchaus vor. Zum Beispiel ist ein Compiler ein Programm, das andere Programme übersetzt. Und ein Pascal-Compiler kann auch selbst in Pascal geschrieben sein, kann also auch selbst kompiliert werden.

Zusammengefasst kann man nun folgende Beobachtung über das Verhalten von D machen:

$$D(< M >) = \begin{cases} accept, & \text{falls } M \text{ die Eingabe } < M > \text{ nicht akzeptiert,} \\ reject, & \text{falls } M \text{ die Eingabe } < M > \text{ akzeptiert.} \end{cases}$$

Als nächstes überlegen wir uns, was passiert, wenn D als Eingabe seine eigene Beschreibung bekommt:

$$D(< D >) = \begin{cases} accept, & \text{falls } D \text{ die Eingabe } < D > \text{ nicht akzeptiert.} \\ reject, & \text{falls } D \text{ die Eingabe } < D > \text{ akzeptiert.} \end{cases}$$

Was auch immer D tut, nach ihrer Definition muss sie das genaue Gegenteil tun, was offensichtlich ein Widerspruch ist. Das heißt D und demzufolge auch H können nicht existieren. □

Betrachten wir noch einmal die Argumentation im Beweis:
Zuerst nehmen wir an, wir hätten eine Turingmaschine H, die A_{TM} entscheidet. Mit deren Hilfe konstruieren wir eine Maschine D, die bei Eingabe der Beschreibung $< M >$ einer Turingmaschine M genau dann akzeptiert, wenn M bei Eingabe von $< M >$ nicht akzeptiert. Zuletzt lassen wir D mit der Eingabe $< D >$ laufen und erhalten ein Ergebnis, das der Definition von D widerspricht. Was hat das alles mit der Diagonalisierungsmethode aus dem Beweis von Satz 13 zu tun?
Das wird offensichtlich, wenn man das Verhalten aller Turingmaschinen in einer Tabelle auflistet:

Die Tabelle in Abbildung 13 zeigt das Verhalten von H mit den Eingaben $< M_i, < M_j >>$ entsprechend der Tabelle in Abbildung 12 oben. Wenn M_3

Abbildung 13: Der Eintrag an der Stelle (i, j) ist das Ergebnis von H auf $< M_i, < M_j >>$

	$< M_1 >$	$< M_2 >$	$< M_3 >$	$< M_4 >$	...
M_1	accept	reject	accept	reject	
M_2	accept	accept	accept	accept	
M_3	reject	reject	reject	reject	
M_4	accept	accept	reject	reject	
$\vdots$		$\vdots$			

Abbildung 14: Ergebnisse von H auf $< M_i, < M_j >>$ einschließlich

	$< M_1 >$	$< M_2 >$	$< M_3 >$	$< M_4 >$	...	$< D >$	...
M_1	accept	reject	accept	reject			
M_2	accept	accept	accept	accept	...		
M_3	reject	reject	reject	reject			
M_4	accept	accept	reject	reject			
$\vdots$		$\vdots$					
D	reject	reject	accept	reject		?	
$\vdots$		$\vdots$					

die Eingabe $< M_2 >$ nicht akzeptiert, ist der Eintrag für Zeile 3 und Spalte 2 *reject*, da H die Eingabe von $< M_3, < M_2 >>$ ablehnt.

In der Tabelle in Abbildung 14 haben wir die Maschine D in die obere Tabelle eingefügt. Nach unserer Annahme ist D eine Turingmaschine, kommt also unter den aufgezählten M_i vor. Man beachte, dass D stets das Gegenteil von den Einträgen in der Diagonale berechnet. Der Widerspruch taucht an der Stelle mit dem Fragezeichen auf, wo der Eintrag das Gegenteil von sich selbst sein muss.

3.2.4 Eine nicht-Turing-akzeptierbare Sprache

Wie wir gerade gesehen haben, gibt es nicht entscheidbare Sprachen, z. B. A_{TM}. Nun wollen wir demonstrieren, dass es auch Sprachen gibt, die nicht einmal Turing-akzeptierbar sind. A_{TM} ist für diesen Zweck nicht geeignet, da wir (im Beweis von Satz 12) schon gezeigt haben, dass A_{TM} Turing-akzeptierbar ist.

Der folgende Satz demonstriert, dass wenn eine Menge A und auch ihr Komplement $\overline{A}$ Turing-akzeptierbar sind, dann ist diese Menge entscheidbar. Da es nicht entscheidbare Mengen gibt, folgt daraus, dass entweder die-

se Mengen oder deren Komplemente nicht Turing-akzeptierbar sind. Man beachte dabei, dass das Komplement einer Menge die Elemente aus dem Universum enthält, die nicht in der Menge sind. Wir sagen eine Menge ist co-Turing-akzeptierbar, wenn sie das Komplement einer Turing-akzeptierbaren Menge ist.

Satz 15. *Eine Menge A ist genau dann entscheidbar, wenn A und $\overline{A}$ Turing-akzeptierbar sind.*

Beweis: Wir müssen zwei Richtungen beweisen:

1. Wenn A entscheidbar ist, dann sind sowohl A als auch $\overline{A}$ Turing-akzeptierbar.

2. Wenn sowohl A als auch $\overline{A}$ Turing-akzeptierbar sind, dann ist A entscheidbar.

Für die erste Richtung überlegt man sich, dass jede entscheidbare Sprache Turing-akzeptierbar ist, da ein Entscheider ja insbesondere auch ein Akzeptierer ist. Für das Komplement kann man aus diesem Entscheider ebenfalls einen Akzeptierer bauen, indem man die negativen Antworten des Entscheiders als *accept* für das Komplement betrachtet.

Um die zweite Richtung zu beweisen, konstruieren wir aus den Akzeptierern für A und $\overline{A}$ einen Entscheider für A. Sei also M_1 eine Turingmaschine, die A akzeptiert und M_2 eine Turingmaschine, die $\overline{A}$ akzeptiert, dann wird M wie folgt definiert:
$M =$

„Auf Eingabe w:

1. Starte M_1 und M_2 parallel auf w.

2. Wenn M_1 akzeptiert, *accept*; wenn M_2 akzeptiert, *reject*.

“

Das parallele Starten von M_1 und M_2 bedeutet, dass M zwei Bänder hat, eines um M_1 und eines um M_2 zu simulieren. M führt jeweils auf jedem Band einen Schritt aus solange, bis eine der beiden Maschinen anhält. Wir zeigen nun, dass M tatsächlich ein Entscheider für A ist. Jede Zeichenkette w ist entweder Element in A oder im Komplement. Daher muss entweder M_1 oder M_2 die Eingabe w akzeptieren. Also hält M in jedem Falle und auf allen Eingaben an, ist also ein Entscheider. Außerdem akzeptiert M alle Strings

aus A und lehnt alle Strings aus $\overline{A}$ ab. D.h. M ist ein Entscheider für A, also ist A entscheidbar. □

Folgerung: $\overline{A_{TM}}$ ist nicht Turing-akzeptierbar.

Beweis: Wir wissen, dass A_{TM} Turing-akzeptierbar ist. Wenn $\overline{A_{TM}}$ auch Turing-akzeptierbar wäre, dann wäre A_{TM} entscheidbar, was ein Widerspruch zu Satz 14 ist. □

4 Reduzierbarkeit

Im Kapitel 2 haben wir Turingmaschinen als Modell eines allgemeinen Computers eingeführt. Wir haben verschiedene lösbare Probleme präsentiert und ein Beispiel für ein unentscheidbares (algorithmisch unlösbares) Problem angegeben. In diesem Kapitel soll es um weitere unlösbare Probleme gehen. Um die Beweise zu führen, brauchen wir eine allgemeine Methode: *Reduzierbarkeit*.

Eine Reduktion ist ein Weg, ein Problem so in ein anderes umzuwandeln, dass die Lösung des zweiten Problems die Lösung des ersten impliziert. Solche Problemreduktionen kommen auch im täglichen Leben vor, wenn sie auch dort nicht so genannt werden.

So ist zum Beispiel das Finden eines Weges in einer unbekannten Stadt kein Problem mehr, wenn man eine Karte hat. Es reduziert sich also auf das Problem, eine Karte zu besorgen.

Reduzierbarkeit hat immer mit zwei Problemen zu tun, die A und B genannt werden sollen. Wenn A sich auf B reduzieren lässt, kann man die Lösung von B dazu verwenden A zu lösen. In unserem Beispiel ist A das Problem, den Weg zu finden, B das Problem, eine Karte zu besorgen. Man beachte dabei, dass Reduzierbarkeit nichts über die Lösbarkeit von A oder B allein aussagt, es geht nur um die Lösbarkeit von A bei Vorhandensein einer Lösung von B.

Reduzierbarkeit spielt eine große Rolle bei der Klassifizierung von Problemen in Bezug auf Entscheidbarkeit und später in der Komplexitätstheorie in Bezug auf den Berechnungsaufwand. Wenn A auf B reduzierbar ist, dann kann die Lösung von A nicht schwerer als die Lösung von B sein, da jede Lösung von B eine Lösung von A liefert. In Begriffen der Berechenbarkeitstheorie heißt das, wenn A sich auf B reduzieren lässt und B entscheidbar ist, dann ist auch A entscheidbar. Umgekehrt gilt, wenn A unentscheidbar ist und A sich auf B reduzieren lässt, dann ist auch B unentscheidbar. Letzteres werden wir als Beweismethode für Unentscheidbarkeit verwenden: Um zu beweisen dass B unentscheidbar ist, zeigen wir, dass ein bekanntermaßen unentscheidbares Problem A sich auf B reduzieren lässt.

4.1 Unentscheidbare Probleme der Sprachtheorie

Bisher haben wir die Unentscheidbarkeit von A_{TM}, dem Problem festzustellen, ob eine Turingmaschine ein gegebenes Wort akzeptiert, bewiesen. Wir wollen nun das Problem

$HALT_{TM} = \{< M, w >, \text{wobei } M \text{ eine Turingmaschine ist, die auf } w \text{ anhält}\}$

betrachten, bei dem es darum geht, festzstellen, ob eine Turingmaschine auf einer bestimmten Eingabe anhält oder nicht[4].

Satz 16. *$HALT_{TM}$ ist unentscheidbar.*

Beweisidee: Wir wollen zeigen, dass A_{TM} sich auf $HALT_{TM}$ reduzieren lässt, also eine Lösung für $HALT_{TM}$ eine Lösung für A_{TM} liefert. Wir nehmen also an, wir hätten eine Turingmaschine R die $HALT_{TM}$ entscheidet und bauen daraus einen Entscheider S für A_{TM}. Um ein Gefühl dafür zu bekommen, wie S konstruiert werden muss, stelle man sich vor, was S tun soll. S bekommt eine Eingabe der Form $< M, w >$ und muss *accept* ausgeben, falls M die Eingabe w akzeptiert und *reject*, falls M in eine Schleife kommt (vergleiche Bemerkung auf Seite 11) oder ablehnt. Wir starten zunächst M mit Eingabe w. Wenn M akzeptiert oder ablehnt, tut S dasselbe. Allerdings könnte es sein, dass wir nicht feststellen können, ob M in eine Schleife gerät. In diesem Fall würde unsere Simulation nicht terminieren, was für einen Entscheider nicht zulässig ist.

Wie kann uns R dabei helfen, A_{TM} zu entscheiden? Mit R könnte man feststellen, ob M auf w anhält. Falls R feststellt, dass M auf w nicht anhält, dann wird $< M, w >$ nicht akzeptiert, denn dann wäre $< M, w > \notin A_{TM}$. Falls R sagt, dass M auf w anhält, können wir die Simulation von oben ohne Gefahr ausführen.

Beweis: (indirekt) Wir nehmen an, R sei ein Entscheider für $HALT_{TM}$. Wir konstruieren einen Entscheider S für A_{TM} durch:
$S =$

„Auf Eingabe $< M, w >$, wobei M eine Turingmaschine, w ein Wort ist

1. Starte R auf $< M, w >$.
2. Wenn R ablehnt, *reject*.
3. Wenn R akzeptiert, simuliere M auf w bis M anhält.
4. Falls M akzeptiert, *accept*. Falls M ablehnt, *reject*.

“

Falls R das Problem $HALT_{TM}$ entscheidet, dann wäre S ein Entscheider für A_{TM}. Da wir wissen, dass S nicht existieren kann, kann auch kein

[4]Im Abschnitt 3.2 habe wir den Begriff Halteproblem schon für A_{TM} verwendet, während das eigentliche Halteproblem $HALT_{TM}$ ist. Von hier an werden wir A_{TM} konsequenterweise als Akzeptanzproblem bezeichnen, um die beiden Probleme voneinander zu unterscheiden.

Entscheider R für $HALT_{TM}$ existieren, also ist $HALT_{TM}$ unentscheidbar. □

Der Beweis illustriert die übliche Methode für Unentscheidbarkeitsbeweise. Diese Methode wird für die meisten Unentscheidbarkeitsbeweise verwendet, außer für A_{TM}, dessen Unentscheidbarkeit wir direkt mit Hilfe der Diagonalisierungsmethode bewiesen haben. Im folgenden werden wir eine Reihe weiterer Probleme auf diese Art beweisen. Als nächstes betrachten wir das Leerheitsproblem für Turingmaschinen, das sich wie folgt formulieren lässt.

$$E_{TM} = \{< M > \text{wobei } M \text{ eine Turingmaschine ist und } L(M) = \emptyset\}$$

Satz 17. *E_{TM} ist unentscheidbar.*

Beweisidee: Wieder nehmen wir an, wir hätten einen Entscheider R für E_{TM} und bauen daraus einen Entscheider S für A_{TM}. Wie würde S arbeiten? Eine Idee wäre, R auf Eingabe $< M >$ zu starten und zu sehen, ob R akzeptiert. Wäre das der Fall, dann wüssten wir dass $L(M)$ leer ist und kein Wort w von M akzeptiert wird. Im anderen Fall, falls R ablehnt, können wir allerdings nichts darüber sagen, ob M die Eingabe w akzeptiert oder nicht.

Wir brauchen eine andere Idee: Wir modifizieren M so zu M_w, dass M_w alle Wörter außer w zurückweist. Nur auf w funktioniert M_w wie gewöhnlich M. Dann nutzen wir R um festzustellen, ob die Sprache von M_w leer ist. Die einzige Eingabe, die M_w überhaupt akzeptieren könnte, wäre w. Also ist $L(M_w)$ genau dann nicht leer wenn w von M akzeptiert wird. Wenn R die Beschreibung einer dementsprechend modifizierten Turingmaschine akzeptiert, dann wissen wir, dass M das Wort w nicht akzeptiert.

Beweis: Wir beschreiben zuerst die modifizierte Maschine:
$M_w =$

„Auf Eingabe x

1. Falls $x \neq w$, *reject*.
2. Falls $x = w$, starte M mit Eingabe w und *accept*, falls M akzeptiert.

“

Für die Turingmaschine M_w ist w ein Teil ihrer Beschreibung. Sie führt den Test, ob $w = x$ auf dem offensichtlichen Weg durch, indem sie die Eingabe Zeichen für Zeichen mit w vergleicht. Fassen wir nun die Vorbereitungen zusammen. Unter der Annahme, wir hätten einen Entscheider R für E_{TM} konstruieren wir den Entscheider S für A_{TM}:

$S =$

„Auf Eingabe $< M, w >$, wobei M eine Turingmaschine, w ein Wort ist:

1. Verwende Beschreibung von M und w um die Maschine M_w zu konstruieren.
2. Starte R mit Eingabe $< M_w >$.
3. Falls R akzeptiert, *reject*. Falls R zurückweist, *accept*.

“

Man beachte, dass S tatsächlich in der Lage sein muss, eine Beschreibung von M_w aus der Beschreibung von M und w zu konstruieren. Das ist möglich durch Hinzufügen von Zuständen zu M, um den Vergleich $x = w$ auszuführen.

Wäre R wirklich ein Entscheider für E_{TM}, dann wäre S ein Entscheider für A_{TM}, da ein solcher aber nicht existieren kann, kann auch R nicht existieren und wir haben einen Widerspruch zur Annahme. Also ist E_{TM} unentscheidbar. □

Das nächste Problem, das uns interessiert, ist die Frage, ob man (automatisch) feststellen kann, ob die von einer gegebenen Turingmaschine akzeptierte Sprache auch durch ein einfacheres Modell akzeptiert werden kann. Als Beispiel betrachten wir:

$$REGULAR_{TM} = \{< M > |\ M \text{ ist eine Turingmaschine, } L(M) \text{ ist regulär}\}.$$

Satz 18. *$REGULAR_{TM}$ ist unentscheidbar.*

Beweisidee: Wir nehmen wieder an, R wäre Entscheider für $REGULAR_{TM}$ und bauen einen Entscheider S für A_{TM}. Allerdings ist hier weniger offensichtlich, wie man R dafür benutzen soll. Die Idee für S ist, die Eingabe $< M, w >$ zu verwenden um eine modifizierte Maschine M' zu konstruieren, so dass M' genau dann eine reguläre Sprache akzeptiert, wenn M die Eingabe w akzeptiert. M' wird so konstruiert, dass sie die nichtreguläre Sprache $\{0^n 1^n \mid n \geq 0\}$ akzeptiert, wenn w von M nicht akzeptiert wird und die reguläre Sprache $\{0, 1\}^*$ wenn w von M akzeptiert wird. Wir müssen angeben, wie S aus der Beschreibung von M und w eine solche Maschine M' bauen kann. M' akzeptiert automatisch alle Wörter der Form $0^n 1^n$ und, falls w von M akzeptiert wird, zusätzlich alle anderen Wörter.

Beweis:
Sei R ein Entscheider für $REGULAR_{TM}$. Wir konstruieren einen Entscheider S für A_{TM}.
$S =$

„Auf Eingabe $< M, w >$ wobei M Turingmaschine, w ein Wort ist:

1. Konstruiere $M'_w =$
   „Auf Eingabe x:
   (a) Falls $x = 0^n 1^n$ für ein n, *accept*.
   (b) Falls x nicht von dieser Form ist, starte M auf w und *accept*, wenn w von M akzeptiert wird.“
2. Starte R mit der Eingabe $< M'_w >$.
3. Falls R akzeptiert, dann *accept*; falls R ablehnt, *reject*.

“

□

Auf ähnliche Weise können die Probleme, ob die von einer Turingmaschine akzeptierte Sprache kontextfrei, entscheidbar oder sogar endlich ist, als unentscheidbar bewiesen werden.

Es gibt sogar ein allgemeines Resultat, den Satz von Rice, der besagt, dass das Testen auf irgendeine Eigenschaft, die die von Turingmaschinen akzeptierten Sprachen haben können, unentscheidbar ist.

Satz 19 (von Rice)**.** *Sei P ein Problem(eine Menge) von Turingmaschinen, das(die) die beiden folgenden Eigenschaften besitzt:*

1. *P ist extensional: Für zwei Turingmaschinen M_1 und M_2 mit $L(M_1) = L(M_2)$ gilt $< M_1 > \in P$ gdw. $< M_2 > \in P$ (d.h. ob eine Maschine zu P gehört, hängt nur von der Sprache, nicht von der Beschreibung von M ab).*
2. *P ist nichttrivial: Es existieren Turingmaschinen M_1 und M_2 wobei $< M_1 > \in P$ und $< M_2 > \notin P$.*

Dann ist P nicht entscheidbar.

Beweis: (Nachschlagen, Hausaufgabe) □

Bis jetzt haben wir als Methode, um die Unentscheidbarkeit eines Problems zu beweisen immer eine Reduktion von A_{TM} benutzt. Manchmal ist es aber einfacher, von einer anderen unentscheidbaren Sprache zu reduzieren.

Der folgende Satz besagt, dass das Problem, ob zwei Turingmaschinen dieselbe Sprache akzeptieren, unentscheidbar ist. Man kann das mit Hilfe einer Reduktion von A_{TM} zeigen, aber wir werden hier eine Reduktion von E_{TM} konstruieren.

Sei

$$EQ_{TM} = \{< M_1, M_2 > |\ \ M_1, M_2 \text{ Turingmaschinen mit } L(M_1) = L(M_2)\}.$$

Satz 20. *EQ_{TM} ist unentscheidbar.*

Die Idee für den Beweis ist einfach. E_{TM} ist das Leerheitsproblem. Wenn eine der zu untersuchenden Sprachen M_1 oder M_2 leer wäre, dann müsste man nur noch feststellen, ob die andere auch leer ist. In einem gewissen Sinn ist also E_{TM} ein Spezialfall von EQ_{TM}, wobei eine der beiden Maschinen aus der Eingabe von EQ_{TM} die ist, die gar kein Wort (also die leere Sprache) akzeptiert.

Beweis: Sei R ein Entscheider für EQ_{TM}; wir konstruieren einen Entscheider S für E_{TM} wie folgt:

$S =$

„Auf Eingabe $< M >$, wobei M eine Turingmaschine ist:

1. Starte R auf der Eingabe $< M, M_1 >$, wobei M_1 die Turingmaschine ist, die alle Eingaben zurückweist.
2. Falls R akzeptiert, *accept*; sonst *reject*.

“

Würde R tatsächlich EQ_{TM} entscheiden, dann wäre S ein Entscheider für E_{TM}. Nach Satz 17 ist E_{TM} aber unentscheidbar, also war die Annahme, dass es einen Entscheider für EQ_{TM} gibt falsch. □

4.2 Reduktion über Berechnungshistorie

Die Verwendung der Berechnungshistorie ist eine wichtige Methode, um die Reduzierbarkeit von A_{TM} auf bestimmte Sprachen zu beweisen. Sie ist oft dann sinnvoll, wenn das Problem, dessen Unentscheidbarkeit zu zeigen ist, etwas mit der Existenz von Zeugen zu tun hat. Mit ihrer Hilfe wurde zum Beispiel bewiesen, dass Hilberts zehntes Problem – die Feststellung, ob ein Polynom ganzzahlige Wurzeln hat – unentscheidbar ist. Die Berechnungshistorie einer Turingmaschine auf einer Eingabe ist einfach die Folge der Konfigurationen, die die Maschine von der Eingabe an durchlaufen hat.

Definition 13. *Sei M eine Turingmaschine und w ein Eingabewort.*

1. *Eine akzeptierende Berechnungshistorie für M auf w ist eine Folge von Konfigurationen $C_1, C_2, \ldots C_l$, wobei C_1 die Startkonfiguration von M auf w ist und C_l eine akzeptierende Konfiguration von M und jedes C_i eine Nachfolgefonfiguration von C_{i-1} ist.*

2. *Eine ablehnende Berechnungshistorie ist eine Folge von Konfigurationen*
   $C_1, C_2, \ldots C_l$ wie oben, nur dass C_l eine ablehnende Konfiguration ist.

Was Berechnungshistorien so vielseitig verwendbar macht, ist, dass sie endliche Folgen und für deterministische Maschinen eindeutig bestimmt sind. Nichtdeterministische Turingmaschinen können für eine Eingabe auch mehrere Berechnungshistorien haben, entsprechend den verschiedenen möglichen Berechnungszweigen, aber bis auf weiteres konzentrieren wir uns auf deterministische Turingmaschinen.

Unser erster Unentscheidbarkeitsbeweis mit Hilfe von Berechnungshistorien betrifft den Typ der linear beschränkten Automaten.

Definition 14. *Linear beschränkte Automaten(LBA) sind Turingmaschinen, bei denen der Kopf sich nur auf dem Teil des Bandes bewegen kann, auf dem die Eingabe steht.*

Bei der Programmierung behandelt man das rechte Bandende genauso wie in unserem Modell bisher schon das linke Bandende. Ergibt die Übergangsfunktion am rechten Ende, dass der Kopf nach rechts bewegt werden soll, dann wird diese Anweisung ignoriert.

Linear beschränkte Automaten haben also nur beschränkten Speicherplatz zur Verfügung. Indem man ein Bandalphabet verwendet, das größer ist als das Eingabealphabet kann man den zur Verfügung stehenden Platz um einen konstanten Faktor vervielfachen. D.h. bei Eingabelängen n ist der verfügbare Platz linear in n – daher der Name *linear beschränkt*. Trotz dieser Beschränkung sind linear beschränkte Automatens ziemlich mächtig, so sind z.B. die Entscheider von A_{DEA}, A_{GG}, E_{DEA} und E_{CFG} alle linear beschränkte Automaten. (Tatsächlich ist es schwierig, eine entscheidbare Sprache zu finden, die nicht von einem linear beschränkte Automaten entschieden werden kann.)

Das Wortproblem A_{LBA} mit

$$A_{LBA} = \{< M, w > | \ M \text{ ist ein linear beschränkter Automat, der } w \text{ akzeptiert}\}$$

von linear beschränkte Automaten ist entscheidbar. D.h. für einen beliebigen linear beschränkte Automaten kann man entscheiden, ob er ein bestimmtes Wort akzeptiert.

Um diese Behauptung zu beweisen, benötigen wir noch ein Lemma, das eine Obergrenze der möglichen Anzahl von Konfigurationen in Abhängigkeit von der Länge der Eingabe angibt:

Lemma 1. *Sei M ein linear beschränkter Automat mit q Zuständen und g Symbolen im Bandalphabet. Dann gibt es bei Bandlänge n genau $g^n qn$ verschiedene Konfigurationen von M.*

Beweis: Jede Konfiguration ist eine Art „Schnappschuss bei der Arbeit“. Eine Konfiguration besteht aus dem aktuellen Zustand, der Position und dem Bandinhalt. Bei Bandlänge n und Anzahl der Zeichen des Bandalphabetes gibt es g^n mögliche Bandinhalte. Der Kopf kann an n verschiedenen Stellen stehen und dabei in q verschiedenen Zuständen sein. Alle diese Anzahlen miteinander multipliziert ergeben die angegebene obere Schranke. □

Satz 21. *A_{LBA} ist entscheidbar.*

Beweisidee: Um zu entscheiden, ob der linear beschränkte Automat M auf der Eingabe w anhält, simulieren wir M auf w. Um festzustellen, ob M in eine Schleife kommt, „beobachtet“ man die Berechnung um festzustellen, ob eine Konfiguration schon einmal aufgetaucht ist. Nach dem Lemma wissen wir, dass das mindestens nach qng^n Schritten der Fall sein muss.

Beweis: $L =$

„Auf Eingabe $< M, w >$, wobei M ein linear beschränkter Automat ist und w ein Wort:

1. Simuliere M auf w für qng^n Schritte oder bis M anhält.
2. Falls M anhält und akzeptiert, dann *accept*; wenn M anhält und ablehnt, dann *reject*. Falls M nicht anhält, muss es eine Schleife geben. Daher wird die Eingabe w zurückgewiesen(*reject*).

“

□

Dieser Satz zeigt, dass linear beschränkte Automaten und Turingmaschinen grundverschieden sind. Für linear beschränkte Automaten ist das Akzeptanzproblem entscheidbar – für Turingmaschinen nicht. Allerdings sind auch für linear beschränkte Automatens manche Probleme unentscheidbar wie z.B. das Leerheitsproblem.

$$E_{LBA} = \{< M > |\ M \text{ ist ein linear beschränkter Automat und } L(M) = \emptyset\}$$

Satz 22. *E_{LBA} ist unentscheidbar.*

Beweisidee: Wir konstruieren eine Reduktion von A_{TM} auf E_{LBA}. Wir nehmen an wir hätten einen Entscheider R für E_{LBA} konstruieren daraus einen Entscheider für A_{TM}.

Für eine Turingmaschine M und ein Wort w konstruieren wir einen linear beschränkten Automaten $B_{M,w}$, der als Sprache die akzeptierenden Berechnungshistorien von M bei Eingabe w hat. Wenn M das Wort w akzeptiert, dann enthält $L(B_{M,w})$ genau ein Wort, wenn nicht, dann ist $L(B_{M,w})$ leer. Zuerst zeigen wir, wie $B_{M,w}$ konstruiert werden muss und überzeugen uns, dass diese Konstruktion von einer Turingmaschine (also automatisch) gemacht werden kann.

Eine Eingabe x wird von $B_{M,w}$ genau dann akzeptiert, wenn sie eine akzeptierende Berechnungshistorie von M auf w ist. Wir erinnern uns, das eine Folge von Konfigurationen $C_1, C_2 \dots C_l$ ist. Für Beweiszwecke wollen wir annehmen, dass die Eingabe für $B_{M,w}$ als String in der folgenden Form gegeben ist:

$$\sharp \underbrace{\quad}_{C_1} \sharp \underbrace{\quad}_{C_2} \sharp \underbrace{\quad}_{C_3} \sharp \dots \sharp \underbrace{\quad}_{C_l} \sharp$$

$B_{M,w}$ arbeitet folgendermaßen. Wenn er die Eingabe x bekommt, zerlegt er x zuerst mit Hilfe der Trennzeichen in Wörter $C_1, \dots C_l$. Anschließend über prüft er die drei Bedingungen einer Berechnungshistorie:

1. C_1 ist Startkonfiguration von M auf w.
2. Jedes C_{i+1} ist eine Nachfolgekonfiguration von C_i.
3. C_l ist eine akzeptierende Konfiguration von M.

C_1 ist eine Startkonfiguration für M auf w, wenn $C_1 = q_0 w_1 w_2 \dots w_n$, wobei q_0 der Startzustand von M ist. Sowohl q_0 als auch w sind direkt in $B_{M,w}$ eingebaut, also kann $B_{M,w}$ diese Bedingung einfach testen. Ebenso einfach geht das Testen, ob C_l eine akzeptierende Konfiguration ist, da nur geprüft werden muss, ob q_{accept} in C_l vorkommt.

Um die zweite Bedingung zu überprüfen, muss man testen, ob C_{i+1} und C_i übereinstimmen bis auf die Zeichen unter und neben dem Kopf bei C_i. Diese beiden Stellen müssen entsprechened der Übergangsfunktion von M angepasst worden sein. $B_{M,w}$ überprüft das, indem er im Zickzack über beide Zeichenfolgen läuft und diese vergleicht (vergleiche Beispiel 1 auf Seite 3).

Bemerkung: Man beachte, dass der LBA $B_{M,w}$ nicht dazu konstruiert wurde, um tatsächliche Berechnungen anzustellen. Wir konstruieren ihn nur, um seine Beschreibung als Eingabe für den hypothetischen Entscheider von

E_{LBA} zu verwenden. Wenn der seine Antwort gegeben hat, können wir diese in eine Antwort auf die Frage der Zugehörigkeit von w zu $L(M)$ umrechnen. Damit würden wir A_{TM} entscheiden können, was ein Widerspruch zu Satz 14 wäre.

Beweis: (von Satz 22) Wir nehmen an, E_{LBA} wäre entscheidbar und R wäre der Entscheider. Der Entscheider S für A_{TM} sieht aus wie folgt:

$S =$

„Auf Eingabe $< M, w >$, mit Turingmaschiene M, Wort w:

1. Konstruiere einen linear beschränkten Automaten $B_{M,w}$, sd.

   $$L(B_{M,w}) = \{\text{akzeptierende Berechnungshistorien von } M \text{ auf } w\}.$$

2. Starte R mit der Eingabe $< B >$.

3. Falls R ablehnt, accept; falls R akzeptiert, reject.

“

□

Die Methode, eine Reduktion mit Hilfe von Berechnungshistorien zu konstruieren, kann auch verwendet werden, um die Unentscheidbarkeit von Problemen kontextfreier Grammatiken und Kellerautomaten(PDA) zu beweisen. In Satz 10 haben wir einen Algorithmus präsentiert, mit dessen Hilfe man entscheiden kann, ob eine gegebene Grammatik G überhaupt ein Wort erzeugt, d.h. ob $L(G) = \emptyset$. Wir wissen also, dass das Leerheitsproblem für kontextfreie Grammatiken entscheidbar ist. Das verwandte Problem ALL_{CFG} hingegen ist unentscheidbar, wie wir gleich sehen werden. Sei

$$ALL_{CFG} = \{< G > |\ G \text{ ist eine kontextfreie Grammatik und } L(G) = \Sigma^*\}$$

Satz 23. *ALL_{CFG} ist unentscheidbar.*

Beweisidee: Der Beweis erfolgt durch Widerspruch und Reduktion von $A_T M$. Wir nehmen an, wir hätten einen Entscheider R für ALL_{CFG}. Wenn wir von A_{TM} reduzieren wollen, müssen wir aus einer Eingabe $< M, w >$ für A_{TM} eine kontextfreie Grammatik G konstruieren, die genau dann alle Strings von Σ^* erzeugt, wenn w nicht von M akzeptiert wird. Falls $w \in L(M)$, dann erzeugt G einen bestimmtes Wort **nicht**. Dieses Wort ist die akzeptierende Berechnungshistorie von M bei Eingabe w.

Mit anderen Worten: G wird so konstruiert, dass sie alle Wörter erzeugt, die keine akzeptierende Berechnungshistorie von M für w sind. Um das zu erreichen, verwenden wir die folgende Strategie: Eine akzeptierende Berechnungshistorie von M für w ist ein Wort $x = \sharp C_1 \sharp C_2 \sharp \dots \sharp C_l$, wobei C_i die Konfiguration von M im i-ten Schritt der Berechnung bei Eingabe von w ist. G soll alle Wörter erzeugen, die

1. **nicht** mit C_1 beginnen, oder
2. **nicht** mit C_l enden, oder
3. es ein C_{i+1}in der Folge gibt, das **keine Nachfolgekonfiguration** von C_i gemäß der Überführungsfunktion von M ist.

Wenn M das Wort w nicht akzeptiert, gibt es keine akzeptierende Konfiguration, d.h. alle Wörter verletzen eine der nötigen Bedingungen an eine akzeptierende Konfiguration, d.h. G würde ganz Σ erzeugen wie gewünscht.

Nun kommen wir zur Konstruktion von G. Dafür konstruieren wir einen PDA D. Wie wir wissen, lässt sich jeder PDA automatisch(!) in eine kontextfreie Grammatik „umwandeln“.

Unser PDA D muss im Startzustand nichtdeterministisch verzweigen, um zu entscheiden, welche Bedingung geprüft werden soll. Ein Zweig prüft, ob die Eingabe mit C_1 beginnt, ein Zweig prüft, ob sie mit einer akzeptierenden Konfiguration endet und einer prüft, ob die Übergänge von C_i nach C_{i+1} korrekt sind. in der folgenden Tabelle sehen wir das Verhalten von D in den Zweigen:

Zweig 1	Beginnt x mit C_1?	wenn nein, *accept*
Zweig 2	Endet x mit akzeptierender Konfiguration?	wenn nein, *accept*
Zweig 3	Gibt es einen ungültigen Übergang	wenn ja, *accept*

Um den dritten Zweig auszuführen, geht D über die Eingabe, wählt nichtdeterministisch ein C_i aus und legt es auf den Stack bis er zum Trennsymbol $\sharp$ kommt. Anschließend wird der Stack mit C_{i+1} verglichen. C_i und C_{i+1} müssten bis auf die Kopfposition und einen Nachbarn übereinstimmen, wobei der Unterschied durch die Überführungsfunktion von M bestimmt sein muss. Damit das Aufschreiben und Vergleichen mit dem Stack klappt, schreiben wir die Konfigurationen ein bisschen anders auf – nämlich wie folgt:

$$\sharp \underbrace{\rightarrow}_{C_1} \sharp \underbrace{\leftarrow}_{C_2^R} \sharp \underbrace{\rightarrow}_{C_3} \sharp \underbrace{\leftarrow}_{C_4^R} \sharp \dots \sharp \underbrace{\qquad}_{C_l} \sharp$$

Der so modifizierte PDA kann eine Konfiguration auf dem Stack direkt mit der Nachfolgekonfiguration vergleichen und ist so eingerichtet, dass jeder String der keine akzeptierende Berechnungshistorie ist, akzeptiert wird.

Die dazugehörige Grammatik G erzeugt dann alle Wörter bis auf die akzeptierende Berechnungshistorie, falls diese existiert. Könnte man ALL_{CFG} entscheiden, dann wüsste man, ob eine solche existiert, d.h. ob w zu $L(M)$ gehört.

Mit Hilfe von Satz 23 kann man auch die Unentscheidbarkeit von EQ_{CFG} beweisen.

4.3 Das Postsche Korrespondenzproblem

In diesem Abschnitt werden wir illustrieren, dass es auch unentscheidbare Probleme gibt, die nichts mit Automaten zu tun haben. Dazu geben wir ein Beispiel für ein unentscheidbares Problem an, das mit der Manipulation von Zeichenketten zu tun hat. Es heißt Postsches Korrespondenzproblem, abgekürzt PCP, und läßt sich als eine Art Spiel beschreiben.

Gegeben ist eine Menge von Dominosteinen, die oben und unten mit je einem Wort beschriftet sind. Ein einzelner Stein sieht z.B. so aus:

$$\left[\frac{a}{ab}\right]$$

Eine Menge solcher Steine sieht dann so aus:

$$\left\{\left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right]\right\}$$

Die Aufgabe besteht darin, die Steine so in einer Reihe anzuordnen, dass eine Korrespondenz (ein Match) entsteht, d.h. die obere und die untere Reihe, als String gelesen, dasselbe Wort ergeben. Dabei sind Wiederholungen erlaubt. Zum Beispiel ist:

$$\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right]$$

ein Match für die obige Menge, da das obere Wort *abcaaabc* mit dem unteren Wort übereinstimmt. Man kann die Korrespondenz deutlicher sichtbar machen, wenn man die Dominosteine deformiert, so dass die zueinander gehörenden Symbole übereinander stehen.

Für manche Mengen von Steinen ist es gar nicht möglich, eine Korrespondenz zu finden. So gibt es zum Beispiel für die Menge:

$$\left\{\left[\frac{abc}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{acc}{ba}\right]\right\}$$

keine Korrespondenz, da die oberen Strings stets länger als die unteren sind.

Das PCP besteht darin, für eine festgelegte Menge von Dominosteinen zu entscheiden, ob es ein Match gibt oder nicht. Dieses Problem ist algorithmisch unlösbar. Bevor wir diese Aussage und ihren Beweis formal darstellen, wollen wir das Problem genau definieren und wie die Automatenprobleme als Menge ausdrücken.

Definition 15. *(i) Eine Instanz des Postschen Korrespondenzproblems ist eine Menge P von Dominosteinen.*

$$P = \left\{ \left[\frac{o_1}{u_1}\right], \left[\frac{o_2}{u_2}\right], \ldots, \left[\frac{o_k}{u_k}\right] \right\}.$$

(ii) Eine Korrespondenz ist eine Folge $\{i_1 \ldots i_l\}$, wobei $o_{i_1} o_{i_2} \ldots o_{i_l} = u_{i_1} u_{i_2} \ldots u_{i_l}$ ist.

(iii) $PCP = \{P$ ist eine Instanz des PCP mit einer Korrespondenz (Match)$\}$

Satz 24. *PCP ist unentscheidbar.*

Von der Idee her ist der Beweis einfach, obwohl er eine Menge technische Feinheiten braucht. Die verwendete Technik ist die Reduktion von A_{TM} über akzeptierende Berechnungshistorien. Wir zeigen, dass wir für jede Turingmaschine M und jede Eingabe w eine Instanz des PCP konstruieren können, so dass ein Match einer akzeptierenden Berechnungshistorie von M für w entspricht. Könnten wir entscheiden, ob die PCP-Instanz ein Match hat, dann wüßten wir, ob w von M akzeptiert wird.

Wie konstruiert man eine Instanz P so, dass ein Match eine akzeptierende Berechnungshistorie ist? Dazu wählen wir die Dominosteine von P so, dass ein Match einer Simulation von M auf w entspricht. In dem Match verbindet jedes Domino eine oder mehrere Positionen einer Konfiguration mit der/den entsprechenden Position(en) in der nächsten.

Bevor wir mit der eigentlichen Konstruktion beginnen, betrachten wir zwei kleine technische Finessen. Um die Konstruktion von P zu erleichtern, nehmen wir zunächst an, dass M bei der Eingabe w niemals die Anweisung L erhält, wenn der Kopf bereits am linken Bandende steht. Dazu müssen wir das Verhalten von M leicht abändern. Als zweites modifizieren wir PCP so, dass ein Match immer mit dem ersten Dominostein

$$\left[\frac{o_1}{u_1}\right]$$

anfängt. Das so modifizierte PCP soll $MPCP$ heißen.

$$MPCP = \{P \text{ ist eine Instanz des } PCP \text{ mit einem Match},$$

das mit dem ersten Stein beginnt.}

Beweis:

Sei R ein Entscheider für PCP; wir konstruieren einen Entscheider S für A_{TM}. Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ die gegebene Turingmaschine mit den üblichen Komponenten. Wir wollen, dass S (der zu bauende Entscheider für A_{TM}) eine Instanz von PCP konstruiert , die genau dann ein Match hat, wenn $w \in L(M)$. Dazu konstruieren wir zuerst eine Instanz P' von $MPCP$. Wir beschreiben die Konstruktion in sieben Teilen, von denen jeder einen bestimmten Aspekt der Simulation von M auf w behandelt.

Teil 1: Nimm als ersten Stein $\left[\frac{o_1}{u_1}\right]$ den Stein

$$\left[\frac{\sharp}{\sharp q_0 w_1 w_2 \dots w_n \sharp}\right]$$

in P' auf. Da P' eine Instanz von $MPCP$ ist, muss ein Match mit diesem Stein beginnen. Daher muss der untere String mit $q_0 w_1 w_2 \dots w_n$, der ersten Konfiguration einer akzeptierenden Berechnungshistorie von M auf w, beginnen. Um ein Match zu bekommen, muss der obere String so erweitert werden, dass er zum unteren passt. Dafür müssen weitere Steine hinzugefügt werden. Die zusätzlichen Steine bedingen, dass die nächste Konfiguration von M als Erweiterung des unteren Strings auftreten, indem sie eine Einschrittsimulation von M erzwingen.

In den Teilen 2, 3 und 4 ergänzen wir die Dominosteine von P', die den Hauptteil der Simulation ausführen. Teil 2 behandelt die Kopfbewegung nach rechts, Teil 3 die Kopfbewegung nach links, und Teil 4 behandelt die Bandzellen, die nicht neben dem Kopf liegen.

Teil 2:

Für jedes $a, b \in \Gamma$ und jedes $q, r \in Q$, wobei $q \neq q_{reject}$:

$$\text{Ergänze } P' \text{ um } \left[\frac{qa}{br}\right], \text{falls } \delta(q, a) = (r, b, R).$$

Teil 3:

Für jedes $a, b, c \in \Gamma$ und jedes $q, r \in Q$, wobei $q \neq q_{reject}$:

$$\text{Ergänze } P' \text{ um } \left[\frac{cqa}{rcb}\right], \text{falls } \delta(q, a) = (r, b, L)$$

Teil 4:

Für jedes $a \in \Gamma$

$$\text{ergänze } P' \text{ um } \left[\frac{a}{a}\right].$$

An dieser Stelle fügen wir ein kleines Beispiel ein, um den Faden im Beweis nicht zu verlieren:

Beispiel 14. *Sei* $\Gamma = \{0, 1, 2, \sqcup\}$ *und* $w = 0100$. *Der Startzustand von* M *heiße* q_0. *Im Zustand* q_0, *beim Lesen von* 0 *soll die Überführungsfunktion verlangen, dass* M *nach* q_7 *übergeht, eine* 2 *schreibt und den Kopf nach rechts bewegt, also* $\delta(q_0, 0) = (q_7, 2, R)$.
Teil 1 liefert den ersten Stein

$$\left[\frac{\sharp}{\sharp q_0 0100 \sharp}\right] = \left[\frac{o_1}{u_1}\right]$$

Teil 2 liefert

$$\left[\frac{q_0 0}{2 q_7}\right]$$

und Teil 4

$$\left[\frac{0}{0}\right], \left[\frac{1}{1}\right], \left[\frac{2}{2}\right], \left[\frac{\sqcup}{\sqcup}\right]$$

Damit sieht die beginnende Korrespondenz aus wie folgt:

$$\left[\frac{\sharp}{\sharp q_0 0100 \sharp}\right] \left[\frac{q_0 0}{2 q_7}\right] \left[\frac{1}{1}\right] \left[\frac{0}{0}\right] \left[\frac{0}{0}\right]$$

Die Steine aus den Teilen 2,3 und 4 erlauben uns also das Match zu erweitern durch Anfügen der zweiten Konfiguration an die erste. Dieser Prozess soll durch Hinzufügen der dritten, vierten und weiteren Konfigurationen weiter fortgesetzt werden. Dazu brauchen wir noch Steine, die das Trennsymbol kopieren.

Teil 5:

$$\text{Ergänze } P' \text{ um } \left[\frac{\sharp}{\sharp}\right] \text{ und } \left[\frac{\sharp}{\sqcup \sharp}\right].$$

Der erste dieser Steine erlaubt uns, das Trennsymbol für die Konfigurationen zu kopieren und der zweite, ein Blanksymbol an das Ende einer Konfiguration zu schreiben, um die unendlich vielen Blanksymbole auf dem Band darzustellen, die beim Schreiben der Konfigurationen weggelassen werden.

Beispiel 15. *(Fortsetzung)Nehmen wir an im Zustand* q_7 *würde die Maschine* M *beim Lesen von 1 in den Zustand* q_5 *wechseln, eine* 0 *schreiben und den Kopf nach rechts bewegen, also* $\delta(q_7, 1) = (q_5, 0, R)$, *dann muss nach Teil 2 der Stein*

$$\left[\frac{q_7 1}{0 q_5}\right]$$

zu P' hinzugefügt werden. Danach sei $\delta(q_5, 0) = (q_9, 2, L)$ und wir müssen nach Teil 3 die Steine

$$\left[\frac{0q_50}{q_902}\right], \left[\frac{1q_50}{q_912}\right], \left[\frac{2q_50}{q_922}\right] \text{ und } \left[\frac{\sqcup q_50}{q_9\sqcup 2}\right]$$

hinzufügen, von denen der erste relevant ist, da der Kopf eine 0 *liest.*

Man beachte, dass man bei der Konstruktion des Matches gezwungen ist, das Verhalten von M auf w zu simulieren. Dieser Prozess geht solange, bis wir einen Haltezustand erreichen. Wenn das eintritt, möchten wir den oberen Teil des partiellen Matches mit dem unteren so ergänzen, dass es vollständig ist. Wir tun das durch folgende zusätzliche Dominosteine:

Teil 6:

Ergänze P' für jedes $a \in \Gamma$ durch

$$\left[\frac{aq_{accept}}{q_{accept}}\right] \text{ und } \left[\frac{q_{accept}a}{q_{accept}}\right].$$

Dieser Teil hat den Effekt, „Pseudoschritte“ von M hinzuzufügen, nachdem die Turingmaschine gehalten hat, in denen noch verbleibende Symbole „gelöscht“ werden können.

In unserem Beispiel bedeutet das folgendes: Wenn die Maschine im akzeptierenden Zustand anhält, d.h das untere Wort z.B. mit $\sharp 21q_{accept}02$ und das obere mit $\sharp$ endet, dann können wir mit Hilfe der in Teil 6 definierten Steine mit der Folge:

$$\left[\frac{2}{2}\right], \left[\frac{1}{1}\right], \left[\frac{q_{accept}0}{q_{accept}}\right], \left[\frac{2}{2}\right] \ldots.$$

fortfahren. Das Spiel wird fortgesetzt, bis der untere String auf dem letzte Stein mit q_{accept} endet. Danach kommt noch ein Stein der Form

$$\left[\frac{\sharp}{\sharp}\right]$$

Teil 7: Zuletzt brauchen wir noch einen Stein der Form:

$$\left[\frac{q_{accept}\sharp\sharp}{\sharp}\right]$$

um das Match zu vervollständigen.

Wir erinnern uns, dass P' als Instanz von $MPCP$ konstruiert wurde, wobei das Match der Simulation von M auf w entsprach. Um den Beweis zu beenden, müssen wir uns daran erinnern, dass sich $MPCP$ von PCP

dadurch unterscheidet, dass ein Match mit dem ersten Stein aus der Liste beginnen soll. Wenn wir P' als Instanz von PCP betrachten, dann hat es immer ein Match, unabhängig davon, ob M auf w anhält oder nicht.

Wir müssen nun zeigen, wie man P' in eine Instanz P von PCP umwandelt, die immer noch die Arbeit von M auf w simuliert. Dazu bedient man sich eines technischen Tricks.

Anstatt explizit zu fordern, dass mit dem ersten Dominostein begonnen wird, wird dieser Beginn in das Problem eingebaut. Zu diesem Zweck müssen wir noch ein bißchen Notation einführen:

Sei $u = u_1 u_2 \dots u_n$ ein Wort der Länge n. Wir definieren drei Strings $\bigstar u, u\bigstar$ und $\bigstar u\bigstar$ durch:

$$\bigstar u = *u_1 * u_2 * \dots * u_n \tag{1}$$

$$u\bigstar = u_1 * u_2 * \dots * u_n* \tag{2}$$

$$\bigstar u\bigstar = *u_1 * u_2 * \dots * u_n* \tag{3}$$

Um P' in eine Instanz P von PCP umzuwandeln tun wir folgendes: Wenn P' aus der Folge :

$$\left\{ \left[\frac{o_1}{u_1}\right], \left[\frac{o_2}{u_2}\right], \dots, \left[\frac{o_k}{u_k}\right] \right\}$$

besteht, dann soll P die folgende Menge sein:

$$\left\{ \left[\frac{\bigstar o_1}{\bigstar u_1\bigstar}\right], \left[\frac{\bigstar o_1}{u_1\bigstar}\right], \left[\frac{\bigstar o_2}{u_2\bigstar}\right], \left[\frac{\bigstar o_3}{u_3\bigstar}\right], \dots, \left[\frac{\bigstar o_k}{u_k\bigstar}\right], \left[\frac{\bigstar \lozenge}{\lozenge}\right] \right\}$$

Betrachtet man P als Instanz von PCP, so sieht man, dass eine Korrespondenz nur erreicht werden kann, wenn man mit dem ersten Stein beginnt. Der letzte Stein dient dazu, ein zusätzliches Stern-Symbol am Ende der oberen Reihe hinzuzufügen. Außer dem Effekt, dass nur der erste Stein eine Korrespondenz beginnen kann, haben die zusätzlichen Sterne keinen Einfluss auf eventuell existierende Korrespondenzen. Die Originalsymbole eines Matches treten dadurch nur an den geraden Stellen der Strings auf.

Mit der Konstruktion einer Instanz P von PCP haben wir eine Eingabe für den Entscheider R für PCP gefunden. Da P so gebaut ist, dass es genau dann ein Match gibt, wenn w von M akzeptiert wird, können wir mit Hilfe von R auch A_{TM} entscheiden, was ein Widerspruch zum Satz 14 ist. Also ist unsere Annahme, dass PCP entscheidbar ist falsch, d.h. PCP ist nicht entscheidbar. □

4.4 Funktionale Reduzierbarkeit

Nachdem wir gezeigt haben, wie man die Methode der Reduzierbarkeit dazu verwenden kann, Unentscheidbarkeit von Problemen zu beweisen, geht es nun darum, den Begriff der Reduzierbarkeit formal zu erfassen. Indem wir das tun, können wir Reduzierbarkeit auf differenziertere Art verwenden, z.B. um zu zeigen, dass bestimmte Sprachen nicht Turing-akzeptierbar sind und auch später, in der Komplexitätstheorie, um Algorithmen nach ihrer Platz- oder Zeitkomplexität zu klassifizieren. Reduzierbarkeit lässt sich auf mehrere Arten definieren – je nach Anwendungsgebiet. Eine davon ist die folgende:

Definition 16. *Seien A und B Teilmengen von Wörtern über einem Alphabet Σ. Die Sprache A ist funktional reduzierbar auf die Sprache B, geschrieben $A \leq_t B$, wenn es eine total berechenbare Funktion $t : \Sigma^* \to \Sigma^*$ gibt, so dass $w \in A$ genau dann, wenn $t(w) \in B$. Die Funktion t heißt Reduktion von A auf B.*

Mit Hilfe von t können Instanzen von A in Instanzen von B transformiert werden. Um festzustellen, ob eine Eingabe x zu A gehört ($x \in A$), können wir t auf x anwenden und dann die Frage:„$t(x) \in B$? “ stellen. Der Ausdruck funktionale Reduzierbarkeit stammt daher, dass die Reduktion in Form einer Abbildung/Funktion ausgeführt wird.

Satz 25. *Falls $A \leq_t B$ gilt und B entscheidbar ist, dann ist A entscheidbar.*

Beweis: Sei M ein Entscheider für B und $t : \Sigma^* \to \Sigma^*$ eine Reduktion $N =$

„Auf Eingabe w:

1. Berechne $t(w)$.
2. Starte M auf $t(w)$ und gebe das Ergebnis aus.

“

Offenbar gilt: Falls $w \in A$, dann ist $t(w) \in B$, da t eine Reduktion ist. Daher akzeptiert M die Eingabe $t(w)$, also ist N ein Entscheider für A. □
Folgerung: Falls $A \leq_t B$ und A unentscheidbar ist, dann ist B unentscheidbar.

Um Beispiele für Reduktionsfunktionen zu bekommen, betrachten wir noch mal einige der Beweise, in denen wir Reduzierbarkeit verwendet haben.

Beispiel 16. *1. Beim Beweis von Satz 16 haben wir eine Reduktion von A_{TM} auf $HALT_{TM}$ verwendet. Um die funktionale Reduzierbarkeit nachzuweisen, müssen wir eine berechenbare Funktion t angeben, die eine Eingabe $< M, w >$ in eine Eingabe $< M', w' >$ für $HALT_{TM}$ transformiert, so dass $< M, w > \in A_{TM}$ gdw. $< M', w' > = t(< M, w >) \in HALT_{TM}$.*

Die folgende Turingmaschine berechnet t:
$T =$

„Auf Eingabe $< M, w >$, wobei M Turingmaschine und w Wort:

(a) Konstruiere die Maschine M' mit:
$M =$„Auf Eingabe von x:

i. Starte M auf x.

ii. Falls M akzeptiert, accept.

iii. Falls M zurückweist, beginne eine Schleife“.

(b) Gib $< M', w >$ aus.

“

2. *Der Unentscheidbarkeitsbeweis von PCP enthält zwei Reduktionen. Zuerst wird gezeigt, dass $A_{TM} \leq_t MPCP$ und dann dass $MPCP \leq_t PCP$. In beiden Fällen lässt sich die Reduktionsfunktion einfach bestimmen. Die funktionale Reduktion ist transitiv, man bekommt also $A_{TM} \leq_t PCP$.*

3. *Der Beweis der Unentscheidbarkeit von E_{TM} (Satz 17) zeigt den Unterschied zwischen dem formalen Begriff der funktionalen Reduzierbarkeit und dem intuitiven der Reduzierbarkeit. Aus der Eingabe $< M, w >$ für A_{TM} hatten wir eine Turingmaschine (M_w) konstruiert, mit $L(M_w) = \emptyset$ genau dann, wenn w von M nicht akzeptiert wird. Unsere Funktion t ist also eine Reduktion von A_{TM} auf $\overline{E_{TM}}$! Bei Entscheidbarkeitsbeweisen ist das allerdings kein Problem. Der Beweis ist trotzdem gültig. (Tatsächlich gibt es keine funktionale Reduktion $A_{TM} \leq_t E_{TM}$.)*

Die feine Unterscheidung von funktionaler Reduktion auf Komplementbildung ist manchmal wichtig, um die Nicht-Turing-Akzeptierbarkeit bestimmter Sprachen zu beweisen.

Analog zu Satz 25 kann man den folgenden Satz formulieren:

Satz 26. *Falls $A \leq_t B$ und B Turing-akzeptierbar ist, dann ist A Turing-akzeptierbar.*

Der Beweis geht genauso wie der für Satz 25, nur werden anstelle von Entscheidern Akzeptierer verwendet.

Folgerung: Wenn A nicht Turing-akzeptierbar ist und $A \leq_t B$, dann ist B nicht Turing-akzeptierbar.

Eine typische Anwendung der Folgerung ist eine Reduktion von $\overline{A_{TM}}$, um zu beweisen, dass ein bestimmtes Problem nicht Turing-akzeptierbar ist.

5 Der Rekursionssatz

In diesem Kapitel geht es um ein wichtiges Resultat, das eine bedeutende Rolle bei Untersuchungen zur Berechenbarkeit spielt, den Rekursionssatz. Dieser Satz hat Verbindungen zur mathematischen Logik, der Theorie von sich selbst reproduzierenden Systemen und sogar zu Computerviren. Ein wichtiges Resultat, dass sich als Konsequenz des Rekursionssatzes beweisen lässt, ist der Gödelsche Unvollständigkeitssatz. Um den Rekursionssatz einzuführen, betrachten wir zunächst das folgende Problem.

Man betrachte Maschinen, die andere Maschinen konstruieren, z.B. eine automatische Fabrik, die Autos herstellt. Sie bekommt Rohstoffe, die Roboter führen eine Reihe von Anweisungen aus, und am Ende stehen fertige Autos.

Wir behaupten, dass die Fabrik „komplexer“sein muss als die gefertigten Autos, da das Entwerfen der Fabrik komplexer ist, als das Entwerfen der Autos. Wir begründen das damit, dass der Entwurf für die Fabrik, neben dem Entwurf für die Roboter auch den Entwurf für die Autos beinhalten muss.

Dieselbe Überlegung lässt sich auf jede Maschine A anwenden, die eine Maschine B konstruiert: A muss komplexer sein als B. Allerdings kann eine Maschine nicht komplexer sein als sie selbst. Daraus würde folgen, dass Selbstreproduktion für Maschinen nicht möglich ist. Allerdings ist diese Schlussfolgerung falsch. Der Rekursionssatz zeigt warum.

5.1 Selbstreferenz

Zunächst wollen wir eine Turingmaschine konstruieren, die ihre Eingabe ignoriert und eine Kopie ihrer eigenen Beschreibung ausdruckt. Der Name für diese Maschine ist SELF. Um die Beschreibung übersichtlicher zu machen, beweisen wir zuerst das folgende Lemma.

Lemma 2. *Es gibt eine berechenbare Funktion $q : \Sigma^* \to \Sigma^*$, die für jede Zeichenkette w die Beschreibung einer Turingmaschine P_w ausgibt, die w ausdruckt und hält.*

Beweis: Die Maschine Turingmaschine Q, die die Funktion q berechnet wird so gebaut:
$Q =$

„Auf Eingabe w:

1. Konstruiere $P_w =$ „Auf Eingabe x:
   (a) Lösche x.
   (b) Schreibe w auf das Band.
   (c) Halte an.“
2. Gebe $< P_w >$ aus.

“

□

Die Turingmaschine SELF, die wir bauen wollen, besteht aus zwei Teilen, die wir A und B nennen wollen und die wir als voneinander getrennte Prozeduren betrachten, die zusammen SELF ergeben: $< SELF >=< AB >$. A beginnt und übergibt danach an B. Die Aufgabe von A ist es, eine Beschreibung von B auszugeben und die von B ist es, eine Beschreibung von A auszugeben. Das Ergebnis ist die gewünschte Beschreibung von SELF.

Obwohl die Aufgaben von A und B ähnlich sind, werden sie auf unterschiedliche Weise ausgeführt. Wir zeigen zuerst, wie man Teil A bekommt. Für A verwenden wir die Maschine $P_{<B>}$, beschrieben von $q(< B >)$, das Ergebnis der Anwendung von q auf $< B >$. D.h. Teil A ist eine Turingmaschine, die $< B >$ ausgibt. Allerdings hängt ihre Definition davon ab, dass wir eine Beschreibung von B haben und wir können sie nicht abschließen, bevor wir eine Konstruktion von B haben.

Um B zu beschreiben, können wir nicht $q(< A >)$ verwenden, denn dann hätten wir eine zirkuläre Beschreibung. Stattdessen definieren wir B auf einem anderen Weg so, dass B die Maschine A berechnet, indem B die Konstruktion von A mit Hilfe der Ausgabe von A berechnet.

Wir haben A definiert durch $< A >= q(< B >)$. Jetzt kommt der knifflige Teil: Falls B Zugriff auf $< B >$ hätte, dann könnte man q anwenden und hätte $< A >$. Aber wie bekommt B diesen Zugriff? Das ist einfach, weil $< B >$ genau das ist, was A auf dem Band hinterlässt, also muss B nur auf dem Band nachsehen. Nach Berechnung von $q(< B >) =< A >$ ergänzt B dieses auf dem Anfang des Bandes und zuletzt enthält das Band $< AB >=< SELF >$.

Alles zusammengefasst haben wir folgendes:

$A = P_{<B>}$ und

$B =$

„Auf Eingabe $< M >$ wobei M Teil einer Turingmaschine ist:

1. Berechne $q(< M >)$.
2. Kombiniere das Resultat mit $< M >$ um die Beschreibung zu vervollständigen.
3. Drucke die Beschreibung aus und halte an.

“

Wenn wir SELF starten, bekommen wir das folgende Verhalten:

1. Zuerst startet A und schreibt $< B >$ auf das Band.
2. Start von B mit Eingabe $< B >$.
3. B berechnet $q(< B >) =< A >$ und schreibt $< A >$ vor $< B >$ auf Band
4. B druckt den Bandinhalt und hält an.

Diese Konstruktion lässt sich in jeder Programmiersprache durchführen. Sogar in natürlicher Sprache kann man eine Anweisung schreiben, die ihren eigenen Code ausdruckt: „Drucke diesen Satz!“

Leider lässt sich das nicht unmittelbar in eine Programmiersprache übersetzen, da es dort keinen selbstreferentiellen Ausdrucke „dieser“ gibt. Ein solcher Ausdruck ist aber auch nicht nötig. Die folgende Variante funktioniert auch:

Drucke zwei Kopien des Folgenden, das Zweite in Anführungszeichen
„Drucke zwei Kopien des Folgenden, das Zweite in Anführungszeichen“

In diesem Satz ist die Selbstreferenz durch denselben Trick ersetzt worden, der bei SELF verwendet wurde. Dabei ist B der Teil ohne die Anführungszeichen. A ist der Teil nur mit den Anführungszeichen. A stellt eine Kopie von B für B bereit, so dass B diese Kopie ausführen kann.

Das Rekursionstheorem dient u.a. dazu, so etwas wie das selbstreferentielle „dieser “ (this) in eine Programmiersprache einzubauen. Damit hat man in jedem Programm die Möglichkeit, auf den eigenen Code zu verweisen. Der Rekursionssatz erweitert die Möglichkeiten der Technik, die für die

Konstruktion von SELF verwendet wurde, so dass das Programm auf seine eigene Beschreibung zugreifen und damit rechnen kann, anstatt nur den Code auszugeben.

Satz 27 (Rekursionssatz)**.** *Sei T eine Turingmaschine, die die Funktion $t : \Sigma^* \times \Sigma^* \to \Sigma^*$ berechnet. Dann gibt es eine Turingmaschine R, die die Funktion $r : \Sigma^* \to \Sigma^*$ berechnet mit*

$$r(w) = t(< R >, w)$$

für alle $w \in \Sigma^*$.

Obwohl der Satz sehr technisch wirkt, ist die Aussage des Rekursionssatzes einfach: Um eine Turingmaschine zu bauen, die Zugriff auf ihre eigene Beschreibung hat und damit rechnen kann, konstruiert man nur eine Turingmaschine (im Satz hat sie den Namen T) mit einer zusätzlichen Eingabe für die Beschreibung der Turingmaschine. Dann garantiert der Rekursionssatz eine neue Maschine R, die genauso wie T funktioniert, aber mit der Beschreibung von R automatisch an die erste Stelle gesetzt.

Beweis: Der Beweis funktioniert so ähnlich wie die Konstruktion von $SELF$. Wir konstruieren R in drei Teilen: A, B und T wobei T durch die Voraussetzung des Theorems vorgegeben ist. A ist jetzt die Turingmaschine $P_{<BT>}$, beschrieben durch $q(< BT >)$. Nach der Arbeit von A enthält das Band $< BT >$. B liest wieder den Bandinhalt und wendet q darauf an, das Ergebnis ist wieder $< A >$. Anschließend kombiniert B die Teile A, B und T zu einer einzigen Maschine, schreibt deren Beschreibung auf das Band und übergibt die Steuerung an T. □

Das Rekursionstheorem macht die Aussage, dass Turingmaschinen die Fähigkeit haben, auf ihre eigene Beschreibung zuzugreifen und damit zu rechnen. Auf den ersten Blick mag das wie eine völlig überflüssige Aufgabe aussehen, aber wie wir sehen werden, ist der Rekursionssatz ein mächtiges Werkzeug, um Probleme der Algorithmentheorie zu lösen. Wir verwenden es, indem wir den Satz „beziehe die eigene Beschreibung $< M >$“ in die Beschreibung einer Turingmaschine M einfügen. Wir dürfen dann fortfahren und $< M >$ verwenden wie jeden anderen Eingabewert. Die Beschreibung von $SELF$ sieht dann so aus:

$SELF =$

„Auf Eingabe x:

1. Beziehe die eigene Beschreibung $< SELF >$.
2. Drucke $< SELF >$ aus.

“

5.2 Anwendungen

Computerviren sind Programme, deren Aufgabe beinhaltet, sich selbst weiterzuverbreiten. Wenn man sie betrachtet, sieht man, dass sie Vieles mit biologischen Viren gemeinsam haben. Solange sie als Code-Schnipsel irgendwo gespeichert sind, sind sie inaktiv, aber sowie sie in der richtigen Umgebung sind, z.B. einem Host-Rechner, der damit befallen ist, können sie aktiviert werden und Kopien von sich selbst an andere erreichbare Rechner schicken. Dabei können den Transport verschiedene Medien übernehmen, etwa ein USB-Stick oder das Internet. Um die Hauptaufgabe, nämlich die Vervielfältigung des eigenen Codes zu übernehmen, kann der Virus z.B. die im Beweis des Rekursionstheorems vorgeführte Konstruktion enthalten.

Im Folgenden werden wir drei Sätze angeben, die man mit Hilfe des Rekursionssatzes beweisen kann. Damit wollen wir illustrieren, wieviel einfacher die Beweise durch dieses Werkzeug werden. Als erstes betrachten wir noch einmal die Unentscheidbarkeit von A_{TM}, die wir in Kapitel 4 mit Hilfe des Diagonalisierungsverfahrens bewiesen hatten.

Satz 28. *(=14) A_{TM} ist unentscheidbar.*

Beweis:: Wir nehmen an, wir hätten eine Turingmaschine H, die A_{TM} entscheidet. Wir konstruieren die folgende Turingmaschine B:
$B =$

„Auf Eingabe w:

1. Erzeuge (mit Rekursionstheorem) die Beschreibung $< B >$ von B.
2. Starte H auf der Eingabe $< B, w >$
3. Falls H akzeptiert, reject; falls H ablehnt, accept.

“

Starten von B auf w bewirkt das Gegenteil dessen, was H behauptet, dass es tun wird, also kann es H nicht geben.

□

Eine weitere Anwendung des Rekursionstheorems betrifft die Minimalität von Turingmaschinen. Für eine gegebene Maschine ist die *Länge* der Beschreibung $< M >$ von M die Anzahl der Symbole im String der M beschreibt.

Definition 17. *Eine Turingmaschine M heißt* minimal*, falls es keine Turingmaschine M' gibt, die zu M äquivalent ist und deren Beschreibung $< M' >$ kürzer ist als $< M >$.*

Sei

$$MIN_{TM} = \{< M > | M \text{ ist minimale Turingmaschine}\}$$

Satz 29. *MIN_{TM} ist nicht Turing-akzeptierbar.*

Beweis: Sei E ein Turingmaschine, die MIN_{TM} aufzählt. Wir konstruieren eine Maschine C
$C =$

„Auf Eingabe w:

1. Beziehe die eigene Beschreibung $< C >$.
2. Starte den Aufzähler E und lasse ihn laufen, bis eine Turingmaschine D auftritt, mit $< D > \geq < C >$
3. Simuliere D auf Eingabe w.

“

Da MIN_{TM} unendlich ist, muss die von E erzeugte Liste eine Turingmaschine enthalten, deren Beschreibung länger als die von C ist. Schritt 2 terminiert also. Anschließend simuliert C die Arbeit von D, also erkennen C und D dieselbe Sprache. Da die Beschreibung von C kürzer als die von D ist, kann D nicht minimal sein. Nach Voraussetzung ist E aber ein Aufzähler, der nur die minimalen Turingmaschinen aufzählt. Also kann es ein solches E nicht geben. □

Eine letzte Anwendung des Rekursionssatzes ist ein Fixpunktsatz. Ein Fixpunkt einer Funktion ist ein Wert, der bei Anwendung einer Funktion auf sich selbst abgebildet wird. (bzw. ein Punkt in einem dynamischen System, der sich in der zeitlichen Entwicklung des Systems nicht ändert.) In

unserem Fall betrachten wir Funktionen, die berechenbare Transformationen von Turingmaschinenbeschreibungen sind. Wir zeigen, dass für jede solche Transformation eine Turingmaschine existiert, deren Verhalten unter der Transformation unverändert bleibt.

Satz 30. *Sei $t : \Sigma^* \to \Sigma^*$ eine total berechenbare Funktion. Dann gibt es eine Turingmaschine F, so dass $t(< F >)$ eine zu F äquivalente Turingmaschine beschreibt.*

Bemerkung: In diesem Satz spielt t die Rolle der Transformation, F ist ein Fixpunkt.

Beweis: Sei F die folgende Turingmaschine

$F =$

„Auf Eingabe w:

1. Beziehe mit Hilfe des Rekursionssatzes die eigene Beschreibung $< F >$ von F.
2. Berechne $t(< F >)$, um die Beschreibung einer Turingmaschine G zu erhalten.
3. Simuliere G auf w.

“

Offenbar beschreiben $< F >$ und $t(< F >)$ äquivalente Turingmaschinen da F in Schritt 3 das Verhalten von G simuliert. □

5.3 Entscheidbarkeit logischer Theorien

Mathematische Logik ist ein Zweig der Mathematik, der die Mathematik selbst untersucht. Mathematische Logik beschäftigt sich mit Fragen wie:

- Was ist ein Beweis?
- Was ist ein Theorem?
- Was ist Wahrheit?
- Kann ein Algorithmus entscheiden, welche Behauptungen wahr sind?
- Kann man jede wahre Aussage beweisen?

Im folgenden werden wir mit ein paar dieser Fragen in Berührung kommen. Hauptsächlich wollen wir uns darauf konzentrieren zu bestimmen, ob mathematische Aussagen wahr oder falsch sind und die Entscheidbarkeit dieses Problems untersuchen. Die Antwort wird davon abhängen, aus welchem Gebiet der Mathematik die Aussagen genommen werden. Wir untersuchen zwei Gebiete: Eines, wo man einen Entscheidungsalgorithmus angeben kann und eines, bei dem das nicht möglich ist. Zunächst brauchen wir aber eine präzise formale Sprache, um die entsprechenden Probleme zu formulieren. Wir wollen Aussagen der folgenden Form untersuchen:

1. Es gibt unendlich viele Primzahlen. Als wahr bekannt seit Euklid ca. 2300 Jahre.

$$\forall q \exists p \forall x, y[p > q \wedge (x, y > 1 \Rightarrow xy \neq p)]$$

2. Fermats letzter Satz, bewiesen ca. 1993

$$\forall a, b, c, n[(a, b, c > 0 \wedge n > 2) \Rightarrow a^n + b^n \neq c^n]$$

3. Primzahlzwillingsvermutung

$$\forall q \exists p \forall x, y[p > q \wedge (x, y > 1 \Rightarrow (xy \neq p \wedge xy \neq p + 2)]$$

Um festzustellen, ob wir den Prozess, herauszufinden, ob solche Aussagen wahr sind, automatisieren können, müssen wir die Aussagen als Strings behandeln und wir bilden die Sprache, die aus allen wahren Aussagen besteht. Dann können wir fragen, ob diese Sprache entscheidbar ist.

Zunächst beschreiben wir das Alphabet der Sprache:

$$\{\wedge, \vee, \neg, (,), \forall, \exists, x, R_1, \ldots R_k\}$$

Dabei sind $\wedge, \vee, \neg$ die sogenannten Boolesche Operatoren und „(“ und „)“ Klammern. Die Symbole $\forall$ und $\exists$ heißen Quantoren, x wird verwendet um Variablen zu beschreiben, werden mehere Variablen gebraucht verwendet man weitere Symbole. Die Symbole R_1 bis R_n heißen Relationssymbole.

Eine *Formel* ist eine wohlgeformte Zeichenkette über diesem Alphabet. Die formale Definition für Wohlgeformtheit kann man z.B. in [1] nachschlagen. Eine Zeichenkette der Form $R_i(x_1 \ldots x_n)$ ist eine *atomare Formel*, n ist dabei die Stelligkeit des Relationssysmbols R_i. Alle Vorkommen desselben Relationssymbols in einer wohlgeformten Formel müssen dieselbe Stelligkeit haben.

Definition 18. *Eine Zeichenkette φ ist eine* Formel*, falls sie entweder*

1. *eine atomare Formel ist, oder*
2. *die Form $\varphi_1 \wedge \varphi_2$ oder $\varphi_1 \vee \varphi_2$ oder $\neg\varphi_1$ hat, wobei φ_1 und φ_2 kleinere Formeln sind, oder*
3. *die Form $\forall x[\varphi_1]$ oder $\exists x[\varphi_1]$ wobei φ_1 eine kleinere Formel ist.*

Ein Quantor darf dabei an einer beliebigen Stelle auftreten. Sein Wirkungsbereich ist das Stück der Formel, das innerhalb des zugehörigen Klammerpaares steht, die der quantifizierten Variablen folgt. Eine Formel ist ein *Pränexnormalform*, wenn alle Quantoren am Anfang stehen. Eine Variable, die nicht im Wirkungsbereich des Quantors steht, heißt *freie Variable*. Formeln ohne freie Variablen sind *Aussagen*.

Beispiel 17. *Nur die dritte Formel ist eine Aussage*

1. $R_1(x_1) \wedge R_2(x_1, x_2, x_3)$
2. $\forall x_1[R_1(x_1) \wedge R_2(x_1, x_2, x_3)]$
3. $\forall x_1 \exists x_2 \exists x_3[R_1(x_1) \wedge R_2(x_1, x_2, x_3)]$

Nachdem wir die Syntax der Formeln festgelegt haben, kommen wir zur Bedeutung. Die Booleschen Operatoren und Quantoren haben ihre übliche Bedeutung. Um die Bedeutung der Variablen und Relationssymbole festzulegen, müssen noch zwei Dinge geklärt werden: Erstens das *Universum*, aus dem die Werte für die Variablen genommen werden und zweitens eine Zuordnung spezieller Relationen zu den Relationssymbolen. Dabei ist eine Relation eine Abbildung (eine Funktion) von k-Tupeln über dem Universum in die Menge $\{true, false\}$. Die Stelligkeit des Relationssymbols muss dabei zu der ihr zugeordneten Relation passen.

Ein *Modell* ist ein Universum zusammen mit einer Zuordnung von Relationen zu Relationssymbolen. Formal ist ein Modell $\mathcal{M}$ ein Tupel

$$(U, P_1, \ldots, P_n),$$

wobei U das Universum ist und $P_1, \ldots, P_n$ Relationen, die den Relationssymbolen $R_1, \ldots, R_n$ zugeordnet sind. Die Sprache eines Modells ist die Menge der Formeln, die nur die im Modell beschriebenen Relationssymbole verwenden.

Falls φ eine Aussage in der Sprache eines Modells ist, dann ist φ entweder wahr oder falsch in diesem Modell. Wenn sie wahr ist, dann sagen wir $\mathcal{M}$ ist ein *Modell* dieser Aussage.

Damit man sich angesichts der Menge dieser Definitionen nicht überfordert fühlt, hilft es, sich auf das eigentliche Ziel zu konzentrieren. Was wir mit den Definitionen errreicht haben, ist die Schaffung einer formalen Sprache, die geeignet ist, mathematische Aussagen genau auszudrücken. Die folgenden Beispiele sollen helfen, mit dieser Sprache umzugehen.

Beispiel 18. *Sei φ die Aussage*

$$\forall x \forall y [R_1(x, y) \vee R_1(y, x)].$$

Sei $\mathcal{M} = (\mathbb{N}, \leq)$ die Interpretation, bei der das Universum die Natürlichen Zahlen sind und die Relation, die R_1 zugeordnet wird, die übliche kleinergleich-Relation ist. Offenbar ist φ wahr in $\mathcal{M}$, da für alle Zahlen $a, b \in \mathbb{N}$ gilt $a \leq b$ oder $b \leq a$. Wählt man jedoch als Interpretation $\mathcal{M}_1 = \mathbb{N}, <)$ dann ist φ nicht wahr, da z.B. für $a = b$ weder $a < b$ noch $b < a$ gilt.

Falls man im Voraus weiß, welche Relation gemeint ist, kann man auch die Infix-Schreibweise verwenden, dh. man könnte φ schreiben als

$$\forall x \forall y [x \leq y) \vee (y \leq x)].$$

Sei $\mathcal{M}_2$ die Interpretation, die als Universum die reellen Zahlen und als dreistellige Relation die Relation $PLUS$ hat mit $PLUS(a, b, c) = true$ gdw. $a + b = c$. $\mathcal{M}_2$ ist dann ein Modell für die Aussage

$$\psi = \forall y \exists x [R_1(x, x, y)].$$

Würde man statt $\mathbb{R}$ jedoch $\mathbb{N}$ benutzen, dann wäre ψ keine wahre Aussage. Wie schon im vorigen Beispiel kann man, wenn man weiß wie R_1 interpretiert wird, auch die Infixschreibweise verwenden, womit ψ dann so aussieht:

$$\psi = \forall y \exists x [x + x = y].$$

Wie wir im zweiten Beispiel gesehen haben lassen sich Funktionen als Relationen schreiben. Dies trifft auch auf Konstanten zu, wenn wir nullstellige Relationen erlauben.

Definition 19. *Die* Theorie eines Modells $\mathcal{M}$ *, bezeichnet durch $Th(\mathcal{M})$, ist die Menge aller wahren Aussagen in der Sprache eines Modells.*

5.3.1 Eine entscheidbare Theorie

Die Zahlentheorie ist einer der ältesten Zweige der Mathematik und einer der schwierigsten. Viele harmlos aussehende Aussagen über natürliche Zahlen sowie + und × haben Mathematiker für Jahrhunderte auf Trab gehalten,

wie z.B. die Primzahlzwillingsvermutung. Bevor wir uns dem Ergebnis von Church zuwenden, dass es keinen Algorithmus geben kann, der allgemein entscheidet, ob eine Aussage in Zahlentheorie wahr oder falsch, wollen wir zunächst eine entscheidbare Theorie betrachten.

Sei $\mathcal{M} = (\mathbb{N}, +)$ das Modell, dessen Universum $\mathbb{N}$ ist und dessen einzige Relation durch $PLUS$ interpretiert wird. Die zugehörige Theorie ist $Th(\mathbb{N}, +)$, zu der z. B. $\forall x \exists y[x + x = y]$ gehört, aber nicht $\forall y \exists x[x + x = y]$.

Satz 31. *$Th(\mathbb{N}, +)$ ist entscheidbar.*

Beweisidee: In einer Übungsaufgabe im letzten Semester sollte ein endlicher Automat angegeben werden, der solche Strings aus Buchstaben der Form

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

akzeptiert, bei denen die untere Zeile als Summe der oberen beiden verstanden werden kann. Zum Beispiel gehört

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

zur Sprache.

Wir geben einen Algorithmus an, mit dem wir feststellen können, ob eine Aussage φ aus der Sprache von $(\mathbb{N}, +)$ in diesem Modell wahr ist.

In diesem Beweis sind die Buchstaben des Alphabets i-zeiligen Bitvektoren, d.h. das Alphabet hat 2^i Zeichen.

Sei

$$\varphi = Q_1 x_1 Q_2 x_2 \dots Q_l x_l[\psi].$$

Dabei sind $Q_1 Q_2 \dots Q_l$ Quantoren, also entweder $\forall$ oder $\exists$ und ψ ist eine quantorfreie Formel in den Variablen $x_1, \dots, x_l$.

Für jedes i von 0 bis l definieren wir eine Formel φ_i mit i freien Variablen durch

$$\varphi_i = Q_{i+1} x_{i+1} Q_{i+2} x_{i+2} \dots Q_l x_l[\psi].$$

Es gilt daher $\varphi_0 = \varphi_i$ und $\varphi_l = \psi$.

Für natürliche Zahlen $a_1, \dots, a_i$ schreiben wir $\varphi_i(a_1, \dots, a_i)$ für die Aussage, die entsteht, wenn die Konstanten $a_1, \dots a_i$ für die Variablen $x_1 \dots x_i$ eingesetzt wurden.

Für jedes i von 0 bis l konstruiert der Algorithmus einen endlichen Automaten A_i , der die Menge von Strings akzeptiert, die die i-Tupel darstellen, die ψ wahr machen. Der Algorithmus beginnt mit der direkten Konstruktion von A_l, wobei eine Verallgemeinerung der Methode aus der Übungsaufgabe verwendet wird. In den folgenden Schritten für jedes i von l bis zu 1 verwendet er A_i um A_{i-1} zu konstruieren. Am Ende, wenn er A_0 konstruiert hat, kann er feststellen, ob A_0 den leeren String akzeptiert. Falls ja, dann ist φ wahr und der Algorithmus akzeptiert.

Beweis: Für $i > 0$ definieren wir das Alphabet

$$\Sigma_i = \left\{ \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 1 \end{bmatrix} \right\}$$

der i-zeiligen Bitvektoren. Jeder String aus Σ_i^* ist also ein i-Tupel binärer Wörter. Für $i = 0$ definieren wir $\Sigma_0 = \{[]\}$, wobei [] ein Symbol ist.

Wir geben nun den Algorithmus an, der $Th(\mathbb{N}, +)$ entscheidet. Auf Eingabe einer Aussage φ arbeitet der Algorithmus wie folgt:

Schreibe φ und definiere φ_i für alle i von 0 bis l. Konstruiere wie folgt entsprechend der Beweisidee für jedes i aus φ_i einen DEA A_i der die Strings aus Σ_i^* akzeptiert, die den i-Tupeln $a_1, \ldots, a_i$ entsprechen, für die $\varphi_i(a_1, \ldots, a_i)$ wahr ist.

Um A_l zu konstruieren, beachte man, dass $\varphi_l = \psi$ eine Boolsche Kombination von atomaren Formeln ist. Eine atomare Formel in der Sprache von $Th(\mathbb{N}, +)$ ist eine einfache Addition. Man kann für jede solche Addition einen DEA bauen. Indem man die Konstruktionen für Vereinigung, Durchschnitt und Komplementbildung für reguläre Sprachen verwendet, kann man die entstandenen DEAs zum gewünschten Automaten A_l kombinieren.

Als nächstes zeigen wir, wie man aus einem DEA A_{i+1} den Automaten A_i bauen kann:

Falls $\varphi_i = \exists x_i \varphi_{i+1}$, dann konstruieren wir A_i so, dass er genauso arbeitet wie A_{i+1}, außer dass er anstatt den Wert a_{i+1} als Eingabe zu bekommen, diesen nichtdeterministisch „rät“. Genauer gesagt enthält A_i einen Zustand für jeden Zustand von A_{i+1} und außerdem einen neuen Startzustand. Jedes Mal wenn A_i ein Symbol

$$\begin{bmatrix} b_1 \\ \ldots \\ b_{i-1} \\ b_i \end{bmatrix}$$

liest, wobei jedes $b_j \in \{0,1\}$ ein Bit der Zahl a_j ist, dann rät er nichtdeterministisch ein $z \in \{0,1\}$ und simuliert A_{i+1} auf dem Eingabesymbol

$$\begin{bmatrix} b_1 \\ \dots \\ b_{i-1} \\ b_i \\ z \end{bmatrix}$$

Am Anfang „rät“ A_i die führende Bits von z, die den nicht mitgeschriebenen führenden Nullen von b_1 bis b_i entsprechen sollen, durch nichtdeterministische Verzweigungen von seinem neuen Startzustand zu allen möglichen Zuständen, die A_{i+1} von seinem Startzustand bei Eingabe von Strings der Symbole

$$\{ \begin{bmatrix} 0 \\ \dots \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \dots \\ 0 \\ 1 \end{bmatrix} \}$$

von Σ_{i+1}. Offenbar akzeptiert A_i die Eingabe $(a_1, \dots, a_i)$, wenn ein a_{i+1} existiert sodass $(a_1, \dots, a_{i+1})$, von A_{i+1} akzeptiert wird. Falls $\varphi_i = \forall x_i varphi_{i+1}$, dann ist φ_i äquivalent zu $\neg\exists x_i \neg varphi_{i+1}$ und wir konstruieren den DEA, der das Komplement der Sprache von A_{i+1} akzeptiert, wenden die obige Konstruktion für den Existenzquantor an und bilden anschließend das Komplement, um A_i zu erhalten.

Der endliche Automat A_0 akzeptiert eine Eingabe, falls φ_0 wahr ist. Also ist der letzte Schritt zu testen, ob A_0 das leere Wort ϵ akzeptiert. Wenn ja, dann ist φ wahr und der Algorithmus akzeptiert; wenn nicht, wird abgelehnt. □

5.3.2 Eine unentscheidbare Theorie

Wie schon erwähnt, ist $Th(\mathbb{N}, +, x)$ unentscheidbar. Das heißt, es gibt keinen Algorithmus, mit dessen Hilfe „automatisch“ geprüft werden kann, ob eine beliebige mathematische Aussage wahr oder falsch ist. Nicht einmal dann, wenn man sich auf die Sprache $(\mathbb{N}, +, \times)$ beschränkt. Diese Aussage ist deshalb wichtig, weil sie unter anderem zeigt, dass die Mathematik nicht mechanisierbar ist. Wir werden den Beweis nicht in seiner vollen Länge angeben, sondern nur skizzieren.

Satz 32. *$Th(\mathbb{N}, +, x)$ ist nicht entscheidbar.*

Der Beweis funktioniert im Wesentlichen wie die anderen Unentscheidbarkeitsbeweise aus Kapitel 4. Wir zeigen, dass sich A_{TM} auf $Th(\mathbb{N}, +, \times)$

reduzieren lässt. Für den Beweis der Existenz der Reduktion brauchen wir folgendes Lemma.

Lemma 3. *Sei M eine Turingmaschine, w ein Wort. Dann kann man aus M und w eine Formel $\varphi_{M,w}$ der Sprache $Th(\mathbb{N}, +, \times)$ berechnen, die eine einzige freie Variable x enthält, so dass $\exists x \varphi_{M,w}$ genau dann wahr ist, wenn M die Eingabe w akzeptiert.*

Beweisidee: Die Formel „besagt“, dass x eine (passend kodierte) akzeptierende Berechnungshistorie von M auf w ist. Natürlich ist x ist Wirklichkeit eine ziemlich große ganze Zahl, in der die Berechnungshistorie in einer Form kodiert ist, dass sie mit Benutzung von $+$ und x Operationen überprüft werden kann.

Die tatsächliche Konstruktion ist zu kompliziert, um sie hier darzustellen. Sie extrahiert bestimmte Symbole aus der Berechnungsgeschichte mit Hilfe von $+$ und $\times$, um die Startkonfiguration für M auf w zu überprüfen, zu testen, ob jeder Konfigurationsübergang gültig ist und ob die letzte Konfiguration akzeptierend ist.

Beweis:(von Satz 32) Wir geben eine funktionale Reduktion von A_{TM} zu $Th(\mathbb{N}, +, \times)$ an. Für jede Eingabe $< M, w >$ wird die Formel $\varphi_{M,w}$ (mit Hilfe des Lemmas) konstruiert und die Formel $\exists x \varphi_{M,w}$ ausgegeben. □

Als Nächstes skizzieren wir den Beweis des Gödelschen Unvollständigkeitssatzes. Dieser Satz besagt, dass es in jedem vernünftigen (reasonable) System, in dem der Begriff der Beweisbarkeit in der Zahlentheorie formalisiert ist, Sätze gibt, die wahr aber nicht beweisbar sind.

Ein *formaler Beweis* π für eine Aussage φ ist eine endliche Folge von Aussagen $S_1, S_2, \ldots S_n$ mit $S_n = \varphi$. Dabei folgt jedes S_i aus den vorhergehenden Aussagen und bestimmten grundlegenden Axiomen über Zahlen unter Benutzung von einfachen und genau festgelegten Ableitungsregeln.

Wir können das Konzept eines Beweises hier nicht im Einzelnen erklären, aber die folgenden beiden Eigenschaften sind wesentlich:

1. Die Korrektheit des Beweises einer Aussage kann von einer Maschine überprüft werden. Formal bedeutet das $\{< \varphi, \pi > | \pi$ ist ein Beweis für $\varphi\}$ ist entscheidbar.

2. Das Beweissystem ist korrekt: Falls eine Aussage beweisbar ist (d.h. wenn es einen Beweis für sie gibt), dann ist sie auch wahr.

Es gelten die folgenden drei Sätze.

Satz 33. *Die Menge der beweisbaren Aussagen in $Th(\mathbb{N}, +, \times)$ ist Turing-akzeptierbar.*

Beweis: Der folgende Algorithmus P akzeptiert Eingabe φ falls φ beweisbar ist. P testet jedes mögliche Wort des Alphabets darauf, ob es ein Beweis für φ ist, indem er den in Eigenschaft 1 geforderten Proof-Checker benutzt. Falls irgendeiner der Kandidaten erfolgreich ist, wird φ akzeptiert. □

Mit Hilfe dieses Satzes können wir eine Variante des Unvollständigkeitssatzes beweisen:

Satz 34. *Es gibt wahre Aussagen in $Th(\mathbb{N}, +, \times)$, die nicht beweisbar sind.*

Beweis: (indirekt) Wir nehmen an, alle wahren Aussagen wären beweisbar. Wir beschreiben nun einen Algorithmus D, der entscheidet, ob eine Aussage wahr ist. Sei φ die Eingabe für D. Wir starten den Algorithmus P von Satz 33 parallel auf φ und $\neg\varphi$. Eine der beiden Aussagen muss wahr sein, also nach unserer Annahme auch beweisbar. Also hält P an. Wegen Eigenschaft 2 gilt, dass wenn φ beweisbar ist, dann ist φ auch wahr; wenn $\neg\varphi$ beweisbar ist, dann ist φ falsch. Damit kann D entscheiden, ob eine Aussage wahr oder falsch ist, was ein Widerspruch zu Satz 32 ist. Also muss es wahre aber nicht beweisbare Aussagen in $Th(\mathbb{N}, +, \times)$ geben. □

Wir verwenden nun den Rekursionssatz, um einen wahren aber unbeweisbaren Satz anzugeben.

Satz 35. *Die im Beweis definierte Aussage $\varphi_{unprovable}$ ist nicht beweisbar.*

Beweisidee: Wir konstruieren mit Hilfe des Rekursionssatzes einen Satz der Art: „Dieser Satz ist nicht beweisbar“.

Beweis: Es wird folgende Turingmaschine S gebaut.

$S =$

„Auf Eingabe w, einem beliebigen Wort:

1. Beziehe die eigene Beschreibung $< S >$ von S mit Hilfe des Rekursionssatzes.
2. Konstruiere mit Hilfe des Lemmas unter Satz 32 die Aussage
$$\psi = \neg\exists c[\varphi_{S,0}].$$
3. Lasse P (den proofchecker) aus dem Beweis von Satz 33 auf ψ laufen.
4. Falls der vorhergehende Schritt akzeptiert, *accept*; falls P hält und ablehnt*reject* .

“

Sei $\varphi_{unprovable}$ die Aussage aus Schritt 2. Die Aussage $\varphi_{unprovable}$ ist genau dann wahr, wenn S die Eingabe 0 (die 0 wurde beliebig ausgewählt) nicht akzeptiert. Falls S einen Beweis für $\varphi_{unprovable}$ findet, dann akzeptiert S die Eingabe 0 und $\varphi_{unprovable}$ wäre demnach falsch. Eine falsche Aussage kann aber nicht beweisbar sein, also kann diese Situation nicht auftreten. D.h. die einzige verbleibende Möglichkeit ist, dass S keinen Beweis für $\varphi_{unprovable}$ findet, also dass S die 0 nicht akzeptiert. Dann jedoch wäre $\varphi_{unprovable}$ wahr – wie behauptet. □

Teil II
Komplexitätstheorie

6 Grundbegriffe

6.1 Aufgaben und Ziele der Komplexitätstheorie

Die wichtigste Aufgabe der Komplexitätstheorie ist es, die Berechnungskomplexität von Problemen (oder synonym dazu, ihre „Härte“) so genau wie möglich zu bestimmen. Auch wenn ein Problem entscheidbar und damit im Prinzip mechanisch lösbar ist, kann es praktisch unlösbar sein, da es übermäßig viel Platz oder Zeit braucht. Man betrachte die Menge

$$S = \{x2^{|x|}x \mid x \in \{0,1\}^*\}$$

Wie schwierig ist es zu entscheiden, ob ein Wort $w \in \{0,1,2\}^*$ in S liegt oder nicht?

Die Antwort hängt von verschiedenen Faktoren ab. Hier ist eine Liste von drei möglichen Antworten.

1. Turingmaschinen mit zwei Bändern, eins nur zum Lesen der Eingabe, eins zum Lesen und Schreiben können S in Echtzeit lösen, die Anzahl der Berechnungsschritte entspricht der Länge der Eingabe.

2. Einbandturingmaschinen (unser übliches Modell) brauchen mindestens quadratischen Zeitaufwand (in der Eingabegröße).

3. DEA (spezielle Turingmaschinen) können S gar nicht entscheiden.

Soll die Komplexität eines Problems bestimmt werden, müssen drei Merkmale festgelegt werden, um genau einzugrenzen, welche Art von Komplexität gemeint ist:

- das verwendete *Berechnungsmodell*, also das Werkzeug, mit dem die Aufgabe gelöst werden soll,

- das *Berechnungsparadigma* oder der Akzeptierungsmodus, also zum Beispiel deterministische oder probabilistische oder nichtdeterministische Turingmaschine,

- das verwendete *Komplexitätsmaß* oder die *Ressource*–z.B. die zur Lösung nötige Zeit oder der benötigte Speicherplatz.

Komplexitätstheorie untersucht wichtige interessante Probleme aus vielen Wissensgebieten unter anderem aus Logik, Algebra, Graphentheorie, Physik, Spieltheorie und den Wirtschaftswissenschaften um sie hinsichtlich ihrer Komplexität zu klassifizieren. Eine andere Aufgabe der Komplexitätstheorie

besteht darin, die Berechnungskraft verschiedener algorithmischer Werkzeuge und Automaten und der zugehörigen Berechnungsparadigmata zu vergleichen und die „Trade-offs“ zu bestimmen. Dazu gehören die Zeit-versus-Raum-Frage und die Determinismus-versus-Nichtdeterminismus-Frage. Wir werden uns hauptsächlich auf die Komplexitätsmaße Zeit und Platz im Worst-Case-Komplexitätsmodell konzentrieren und entsprechende Komplexitätsklassen einführen.

Eine Komplexitätsklasse ist eine Menge von Problemen, die gemäß einem gegebenen Berechnungsmodell und -paradigma, durch Algorithmen gelöst werden können, welche höchstens den vorgegebenen Betrag der jeweiligen Komplexitätsressource verbrauchen. Wir werden Sätze über lineare Beschleunigung und Raumkompression sowie Hierarchiesätze für Raum und Zeit kennenlernen. In diesen Resultaten geht es darum, um wieviel eine Komplexitätsressource vergrößert werden muss, damit echt mehr Probleme von einem auf diese Ressourcen beschränkten Algorithmus gelöst werden können.

Um die Berechnungskomplexität von zwei Problemen zu vergleichen, werden wir wieder funktionale Reduktion benutzen, wie schon in den Kapiteln über Berechenbarkeit. Im Zusammenhang damit werden wir die Begriffe Härte und Vollständigkeit eines Problems in einer Komplexitätsklasse einführen, die charakterisieren, ob ein Problem zu einer gewissen Klasse gehört und ob es zu den schwersten Problemen dieser Klasse gehört.

6.2 Komplexitätsmaße und Komplexität

Als Algorithmenmodell werden wir im folgenden wie bisher auch Turingmaschinen betrachten. Wenn wir darüber sprechen wollen, dass ein bestimmtes Problem eine gewisse Komplexität hat, verlangen wir, dass die zugehörige Turingmaschine nach endlich vielen Schritten terminiert, bzw. wenn es sich um eine nichtdeterministische Maschine handelt, dass es mindestens einen akzeptierenden Pfad gibt.

Definition 20 (Deterministische Komplexitätsmaße)**.** *Sei M eine Deterministische Turingmaschine und $\varphi_M : \Sigma^* \to \Sigma^*$, die von M berechnete Funktion. Wir definieren die Zeitfunktion und die Platzfunktion $Time_M, Space_M : \Sigma^* \to \mathbb{N}$ durch:*

$$Time_M(w) = \begin{cases} m & \text{wenn } M \text{ auf } w \text{ nach } m+1 \text{ Konfigurationen terminiert,} \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

$$Space_M(w) = \begin{cases} \text{Anzahl der Bandzellen in einer} \\ \text{größten Konfiguration von } M \text{ auf } w & \text{falls } M \text{ auf } w \text{ terminiert,} \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Natürlich wollen wir nicht nur das Verhalten von M auf einer bestimmten Eingabe beschreiben, sondern auf allen. Wir definieren daher zwei weitere Funktionen $time_M, space_M : \mathbb{N} \to \mathbb{N}$ wie folgt

Definition 21.

$$time_M(n) = \begin{cases} max_{|w|=n} Time_M(w) & \textit{falls } Time_M \textit{ für alle } w \textit{ der Länge } n \textit{ definiert ist,} \\ \textit{undefiniert} & \textit{sonst.} \end{cases}$$

$$space_M(n) = \begin{cases} max_{|w|=n} Space_M(w) & \textit{falls } Space_M \textit{ für alle } w \textit{ der Länge } n \textit{ definiert ist,} \\ \textit{undefiniert} & \textit{sonst.} \end{cases}$$

Dass die $space_M$-Funktion nur dann definiert ist, wenn M terminiert, ist eine Entscheidung, die man auch anders treffen könnte. Schließlich könnte eine Maschine ja durchaus auch mit beschränkten Platzressourcen ewig laufen. Wir sehen unsere Entscheidung deswegen als vernünftig an, da wir uns auf den Standpunkt stellen, dass eine Berechnung genau dann eine Komplexität haben möge, wenn sie terminiert.

Nachdem wir gesehen haben, wie man für einen einzigen Algorithmus die verwendeten Ressourcen beschreibt, wollen wir Probleme, die sich bezüglich ihres Ressourcenverbrauchs ähnlich verhalten, in Klassen versammeln. Von einer solchen Einteilung in Klassen verlangen wir wie schon bei Entscheidbarkeit eine gewisse Robustheit, d.h die Einteilung sollte weder zu fein noch zu grob gewählt werden.

Definition 22 (Deterministische Zeit-und Raumklassen)**.** *Seien $s, t : \mathbb{N} \to \mathbb{N}$ total berechenbar. Wir definieren die deterministischen Komplexitätsklassen mit den Funktionen s und t wie folgt :*

$$\begin{aligned} DTIME(t) = \{A \mid A = L(M) \textit{ für eine DTM } M \textit{ und} \\ \textit{für alle } n \in \mathbb{N} \textit{ gilt: } time_M(n) \leq t(n)\} \\ DSPACE(s) = \{A \mid A = L(M) \textit{ für eine DTM } M \textit{ und} \\ \textit{für alle } n \in \mathbb{N} \textit{ gilt: } space_M(n) \leq s(n)\} \end{aligned}$$

Als nächstes wollen wir Komplexitätsmaße und -klassen für nichtdeterministische Turingmaschinen definieren. Sei $M(w)$ der Berechnungsbaum einer nichtdeterministische Turingmaschine bei Eingabe von w. Die Berechnung entlang eines festen Pfades α ist nichts anderes als eine deterministische Berechnung: Eine Folge von Konfigurationen. Für jeden Pfad α definieren wir die Zeit-und Raumfunktionen wie in Definition 20 für deterministische Turingmaschinen und bezeichnen sie mit $Time_M(w, \alpha)$ und $Space_M(w, \alpha)$.

Definition 23 (Nichtdeterministische Komplexitätsmaße)**.** *Sei M eine nichtdeterministische Turingmaschine mit Sprache $L(M) \subseteq \Sigma^*$, sei $w \in \Sigma^*$ und α ein Pfad in $M(w)$. Wir definieren die Zeitfunktion und die Platzfunktion $NTime_M, NSpace_M : \Sigma^* \to \mathbb{N}$ durch*

$$NTime_M(w) = \begin{cases} min\{Time_M(w,\alpha) \mid M \text{ akzeptiert } w \text{ auf } \alpha\ \} & \text{falls } w \in L(M) \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

$$NSpace_M(w) = \begin{cases} min\{Space_M(w,\alpha) \mid M \text{ akzeptiert } w \text{ auf } \alpha\ \} & \text{falls } w \in L(M) \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Definition 24 (Nichtdeterministische Zeit-und Raumklassen)**.** *Seien $s, t : \mathbb{N} \to \mathbb{N}$ total berechenbar. Wir sagen, M akzeptiert eine Menge A in der Zeit t, falls*

- *$NTime_M(w) \leq t(|w|)$ für jedes $w \in A$ gilt, und*
- *w nicht von M akzeptiert wird, falls $w \notin A$.*

Wir sagen, M akzeptiert eine Menge A im Raum s, falls

- *$NSpace_M(w) \leq s(|w|)$ für jedes $w \in A$ gilt, und*
- *w nicht von M akzeptiert wird, falls $w \notin A$.*

Wir definieren die nichtdeterministischen Komplexitätsklassen mit den Ressourcenfunktionen s und t wie folgt :

$$NTIME(t) = \{A \mid A = L(M) \text{ für eine NTM } M, \text{ die } A \text{ in der Zeit } t(n) \text{ akzeptiert}\}$$

$$NSPACE(s) = \{A \mid A = L(M) \text{ für eine NTM } M, \text{die } A \text{im Raum } s(n) \text{ akzeptiert}\}$$

Wie schon oben erwähnt, wollen wir die Unterscheidung in Komplexitätsklassen nicht zu fein machen.

So wollen wir z.B. $DTIME(n^2)$ und $DTIME(n^2 + 1)$ nicht als zwei echt verschiedene Klassen auffassen.[5] Stattdessen ist es sinnvoll, Familien $\mathcal{F}$ von „Ressourcenfunktionen“ zu betrachten und z.B. eine entsprechende deterministische Zeitklasse $DTIME(\mathcal{F}) = \bigcup_{f \in \mathcal{F}} DTIME(f)$ zu definieren. Eine solche Familie enthält alle Ressourcenfunktionen $f : \mathbb{N} \to \mathbb{N}$ mit einer ähnlichen Wachstumsrate. Beispiele für solche Familien sind:

[5] tatsächlich zeigt der Beschleunigungssatz, dass sie nicht verschieden sind.

- Lin enthält alle lineare Funktionen,
- Pol enthält alle Polynome,
- 2^{Lin} enthält alle Exponentialfunktionen, bei denen der Exponent linear in n ist,
- 2^{Pol} enthält alle Exponentialfunktionen, bei denen der Exponent polynomiell in n ist.

Die Komplexitätsmaße Zeit und Raum, wie auch die resultierenden Komplexitätsklassen sind offenbar invariant unter endlicher Variation, da eine endliche Anzahl von Ausnahmen immer mittels einer lookup Tabelle behandelt werden kann, die dem Programm der Turingmaschine als zusätzliche Information gegeben wird.

6.3 Klein-O und Groß-O Schreibweise

Die exakte Laufzeit oder der exakte Platzverbrauch ist häufig ein komplizierter Ausdruck. Für Abschätzungen reicht die so genannte asymptotische Analyse, bei der die Laufzeit des Algorithmus für große Eingaben abgeschätzt wird.

Definition 25. *Seien $f, g : \mathbb{N} \to \mathbb{R}^+$ zwei Funktionen. Man schreibt $f \in O(g)$, falls positive ganze Zahlen c und n_0 existieren, so dass $\forall n \geq n_0\ f(n) \leq c \cdot g(n)$. Wenn $f \in O(g)$ sagt man, dass g eine obere Schranke für f ist oder genauer: g heißt asymptotische obere Schranke für f um zu betonen, dass konstante Faktoren vernachlässigt werden.*

Intuitiv meint $f \in O(g)$, dass f kleiner oder gleich g ist (bis auf konstanten Faktor). In der Praxis haben die meisten Funktionen einen offensichtlich größten Term h, dann ist $f \in O(g)$, wobei g gleich h ohne den Koeffizienten ist.

Beispiel 19. *Sei $f_1(n) = 5n^3 + 2n^2 + 22n + 6$, dann ist $f_1 \in O(n^3)$[6]. Wir überprüfen die formale Richtigkeit: Für $n \geq 10$ gilt $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ also $n_0 = 10$, $c = 6$. Außerdem gilt auch $f_1 \in O(n^4)$, da auch n^4 eine asymptotische obere Schranke für f_1 ist. f_1 liegt aber nicht in $O(n^2)$, da wir keine geeigneten Konstanten n_0 oder c finden können.*

[6] n^3 steht an dieser Stelle nicht als Funktionswert, sondern als Name für die Funktion $\lambda n.n^3$. Man hätte auch $\_^3$ verwenden können, aber diese Art der Funktionsbeschreibung wird schnell unübersichtlich, insbesondere wenn mehrere Argumente im Spiel sind.

Groß-O arbeitet in spezieller Weise mit Logarithmen zusammen. Bei Logarithmen gibt man normalerweise die Basis an, so ist $x = \log_2 n$ gleich bedeutend mit $2^x = n$. Das Ändern der Basis mit Hilfe der Gleichung $\log_b n = \log_2 n / \log_2 b$ bewirkt eine Änderung des Wertes des Logarithmus nur um einen konstanten Faktor. Wenn man also $f \in O(\log)$ schreibt, muss man keine Basis angeben.

Sei $f_2(n) = 3n \log_2 n + 5n \log_2 \log_2 n + 2$. Dann reicht $f_2 \in O(n \log n)$[7], da $\log n$ größer als $\log \log n$ ist.

Manchmal ist es bequem, $O-Terme$ in Ausdrücken zusammenzufassen, wie zum Beispiel $f \in O(n^2) + O(n)$. Gemeint ist damit, dass $f(n) \leq h(n) + g(n)$ für gewisse h und g mit $h \in O(n^2)$ und $g \in O(n)$. Dabei steht jedes O für eine unterdrückte Konstante. Da $O(n^2)$ den Term in $O(n)$ dominiert, ist dieser Ausdruck äquivalent zu $f \in O(n^2)$.

In manchen Analysen tritt der Ausdruck $f(n) = 2^{O(\log n)}$ auf. Indem man die Gleichung $n = 2^{\log_2 n}$ benutzt, sieht man, dass $n^c = 2^{c \log_2 n}$, d.h. $2^{O(\log n)}$ ist eine obere Schranke für n^c für ein c. Der Ausdruck $n^{O(1)}$ besagt dasselbe, da $O(1)$ einen Wert darstellt, der nie größer als eine Konstante ist.

Ausdrücke der Form n^c mit $c > 0$ heißen polynomielle Schranken. Schranken der Form 2^{n^δ} heißen exponentielle Schranken falls $\delta > 0$ eine reelle Zahl ist.

Ergänzend zur Groß-O-Notation gibt es die Klein-O-Notation. Groß-O besagt, dass eine Funktion asymptotisch nicht größer ist als eine andere. Um zu beschreiben, dass sie asymptotisch kleiner ist, verwendet man:

Definition 26. *Seien $f, g : \mathbb{N} \to \mathbb{R}^+$. Man sagt $f \in o(g)$ falls*

$$lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Mit anderen Worten $f \in o(g)$ heißt, dass für jede reelle Zahl $c > 0$ eine Zahl n_0 existiert, sodass $f(n) < c \cdot g(n)$ für alle $n \geq n_0$.

D.h. der Unterschied zwischen o und O ist wie der zwischen $<$ und $\leq$.

Beispiel 20. *Die folgenden Beispiele lassen sich leicht überprüfen:*

1. *Sei $f_1(n) := \sqrt{n}$, dann gilt $f_1 \in o(n)$.*
2. *Sei $f_2(n) := n$, dann gilt $f_2 \in o(n \log \log n)$.*
3. *Sei $f_3(n) := n \log \log n$ dann gilt $f_3 \in o(n \log n)$.*

[7] wieder eigentlich $\lambda n.n \log n$

4. *Sei* $f_4(n) := n \log n$ *dann gilt:* $f_4 \in o(n^2)$.

5. *Sei* $f_5(n) := n^2$, *dann gilt* $f_5 \in o(n^3)$.

Bemerkung: f ist niemals in $o(f)$.

6.4 Algorithmenanalyse

Gegeben sei der folgende Entscheider für die Sprache $A = \{0^k 1^k \mid k \geq 0\}$.
$M_1 =$

„Auf Eingabe w:

1. Gehe über das Band und *reject*, wenn rechts von einer 1 eine 0 steht.
2. Wiederhole den folgenden Schritt solange sowohl Einsen als auch Nullen auf dem Band sind:
3. Gehe über das Band und streiche dabei immer je eine Null und Eins aus.
4. Falls noch Einsen da sind, wenn es keine Nullen mehr gibt, oder falls noch Nullen da sind, wenn es keine Einsen mehr gibt, *reject*; falls beide zugleich abgestrichen sind, *accept*.

“

Wir analysieren den Algorithmus von M_1 um zu bestimmen, wie viel Zeit er braucht. Die Zahl der Schritte, die der Algorithmus bei einer bestimmten Eingabe braucht, kann von verschiedenen Parametern abhängen. Falls die Eingabe z.B. ein Graph ist, kann die Anzahl der Schritte von der Zahl der Knoten, der Kanten, dem maximalen Grad oder einer Kombination davon abhängen.

In Schritt 1 braucht man n Schritte, um die Gültigkeit der Eingabe zu prüfen. Wieder zum Anfang zurückzugehen braucht auch n Schritte, also $2n$ oder $O(n)$ Schritte. Beachten Sie, dass die Neupositionierung des Kopfes nicht berücksichtigt wurde, da konstante Faktoren keine Rolle spielen. In Zustand 2 und 3 werden jeweils eine Null und Eins abgestrichen, was jedesmal $O(n)$ Schritte benötigt. Da das ganze höchstens $n/2$ mal gemacht wird, haben wir insgesamt in 2 und 3 $(n/2)O(n) = O(n^2)$ Schritte. Im Zustand 4 wird ein Test gemacht, der höchstens $O(n)$ braucht. Damit haben wir $O(n) + O(n^2) + O(n)$ bzw. $O(n^2)$

Wir haben oben vorgeführt, dass A in $DTIME(n^2)$ liegt. Gibt es eine schnellere Maschine? Oder mit anderen Worten: Liegt A in $DTIME(t(n))$ für $t \in o(n^2)$?

Man kann die Laufzeit verbessern, wenn man jeweils zwei Nullen und Einsen in einem Schritt ausstreicht. Aber das halbiert die Anzahl der Arbeitsschritte nur und hat keine Auswirkungen auf das asymptotische Verhalten. Mit der folgenden Maschine kann man A in $O(n \log n)$ Schritten entscheiden.

$M_2 =$

„Auf Eingabe w:

1. Gehe über die Eingabe und *reject*, wenn eine 1 vor einer 0 steht.
2. Wiederhole das folgende solange Nullen und Einsen auf dem Band sind:
3. Teste die Eingabe (über das Band gehen), ob sie gerade oder ungerade Zahl von Nullen und Einsen hat, *reject*, wenn ungerade.
4. Gehe über das Band und streiche jede zweite verbliebene Null aus und jede zweite Eins, jeweils vorn mit der ersten beginnend.
5. Wenn keine Nullen und Einsen mehr da sind, *accept*; anderenfalls *reject*.

“

Offenbar entscheidet M_2 die Menge A. Bei jedem Schritt 4 wird die Anzahl der verbleibenden Nullen halbiert. Wenn wir mit 13 Nullen starten haben wir nach dem ersten Mal sechs, dann drei, dann eine und schließlich Keine Null. Dasselbe gilt für die Einsen. Wir untersuchen nun die gerade/ungerade-Verhältnisse der Anzahl der Nullen und Einsen in jedem Stadium von 3. Falls wir mit 13 Nullen und Einsen begonnen haben, haben wir zuerst eine ungerade Zahl von Nullen und ungerade Zahl von Einsen, dann eine gerade (die 6), eine ungerade (3) und ungerade (1). Wenn keine Nullen vorhanden sind, wird 3. nicht nochmal ausgeführt, wegen Bedingung 2. Wenn wir bei der Folge der„Geradeheiten-Ungeradeheiten“ das Geradesein mit 0 und das Ungeradesein mit 1 darstellen und in umgekehrter Folge aufschreiben, bekommen wir 1101, was gerade die Binärdarstellung von 13 – der Anfangsanzahl – ist. (Das ist immer so!) Wenn in Zustand 3 geprüft wird, ob die Gesamtzahl der verbleibenden Nullen und Einsen gerade ist, überprüft man in Wirklichkeit die Übereinstimmung der Anzahl der Nullen mit der Anzahl der Einsen. Und wenn diese jeweils gleich sind, stimmen auch die Binärdarstellungen, also die Zahlen selbst überein.

Bei Analyse von M_2 stellen wir zuerst fest, dass jedes Stadium $O(n)$ Schritte benötigt. Wir müssen nun noch bestimmen, wie oft jedes ausgeführt wird. Bei Schritt 1 und 5 einmal, also insgesamt $O(n)$; Schritt 4 muss höchstens halb so oft ausgeführt werden, wie es Nullen und Einsen gibt . Es gibt also höchstens $1 + \log_2 n$ Iterationen. Damit ist die Gesamtzeit von 2, 3 und 4 $(1 + \log_2 n)(O(n))$ oder $O(n \log n)$. Also ist die Zeit von M_2 beschränkt durch $O(n) + O(n \log n) = O(n \log n)$. Wir haben bereits gezigt, dass $A \in DTIME(n^2)$. Jetzt haben wir eine bessere Schranke, nämlich $A \in DTIME(n \log n)$. Auf Einbandturingmaschinen lässt sich das Ergebnis nicht weiter verbessern.

Bemerkung: In Wirklichkeit ist jede Sprache, die in $o(n \log n)$ von einer Einbandturingmaschine entschieden werden kann, regulär (ohne Beweis).

Man kann A auch in $O(n)$, also in linearer Zeit entscheiden, d.h. $A \in DTIME(n)$, wenn man eine Zweiband-Turingmaschine verwendet.
$M_3 =$

„Auf Eingabe w:

1. Gehe über das Band und *reject*, wenn 0 nach einer 1 kommt.
2. Gehe über die Nullen auf Band I bis zur ersten 1, kopiere dabei die Nullen auf Band II.
3. Gehe über die Einsen auf Band I, streiche beim Lesen jeder Eins eine Null auf Band II. Wenn alle Nullen ausgestrichen sind bevor die Einsen zu Ende sind, *reject*.
4. Wenn alle Nullen ausgestrichen sind, accept, wenn noch welche übrig sind, *reject*.

“

Jedes der vier Stadien braucht $O(n)$ Schritte, damit ist die totale Rechenzeit linear.

Fassen wir zusammen: M_1 braucht $O(n^2)$, M_2 braucht $O(n \log n)$ Schritte und wir haben behauptet, dass es keine Einbandturingmaschine gibt, die schneller ist. M_3 braucht nur lineare Zeit, d.h. die Zeitkomplexität von A hängt vom Berechenbarkeitsmodell ab. Damit haben wir einen wichtigen Unterschied zwischen Berechenbarkeitstheorie und Komplexitätstheorie gesehen. In der Berechenbarkeitstheorie haben wir die Church-Turing These, die besagt, dass alle denkbaren Berechenbarkeitsmodelle äquivalent sind. d.h. dieselbe Klasse von Sprachen entscheiden. In der Komplexitätstheorie bestimmt die Wahl des Modells die Zeitkomplexität der Sprachen. Wir inter-

essieren uns in der Komplexitätstheorie dafür, wie viel Zeit die Lösung eines Problems benötigt. Welches Modell nehmen wir also? Zum Glück unterscheiden sich die Zeiten nicht allzu sehr für die typischen deterministischen Modelle.

7 Zusammenhänge zwischen den Komplexitätsklassen

7.1 Komplexitätsbeziehungen zwischen den Berechnungsmodellen

Wie beeinflusst die Wahl des Modells die Zeitkomplextität einer Sprache? Wir betrachten drei Modelle: Einbandturingmaschine, Mehrbandturingmaschine, nichtdeterministische Turingmaschine.

Satz 36. *Sei $t(n)$ eine Funktion mit $t(n) \geq n$. Dann gibt es zu jeder $t(n)$-Zeit Mehrbandturingmaschine eine äquivalente $O(t^2(n))$-Zeit Einbandturingmaschine.*

Beweisidee: Wir hatten gezeigt, wie man eine Mehrbandturingmaschine in eine Einbandturingmaschine umwandelt, die sie simuliert. Jetzt analysieren wir diese Simulation um zu bestimmen, wie viel zusätzliche Zeit gebraucht wird. Wir zeigen, dass jeder Schritt der MBTM höchstens $O(t(n))$ Schritte der EBTM braucht, damit ist die totale Zeit $O(t^2(n))$.

Beweis: Sei M die k-Band Turingmaschine, die in $t(n)$-Zeit läuft. Wir konstruieren S mit $O(t^2(n))$-Zeit. S arbeitet, indem sie M simuliert. S verwendet das Band, um den Bandinhalt aller k Bänder zu repräsentieren. Die Bänder werden hintereinander weg gespeichert, mit der markierten Position des Kopfes im entsprechenden Kästchen. Am Anfang bringt S das Band in eine Form, die alle Bänder von M darstellt und dann wird die Arbeitsweise von M simuliert. Um einen Schritt von M zu simulieren, wird das ganze Band gelesen, um alle Symbole unter den Bandköpfen von M zu lesen. Danach läuft S über das Band, um Bandinhalt und Kopfposition zu aktualisieren. Falls einer von M's Köpfen sich nach rechts über den „Rand“ (noch nicht gelesener Teil) bewegt, dann muss S Platz schaffen, indem es den gesamten Bandinhalt nach rechts bewegt (um eins).

Für jeden Schritt von M läuft S zweimal über den aktiven Teil des Bandes; einmal um die nötigen Informationen zu bekommen, das zweite Mal um den Schritt auszuführen. Dabei bestimmt die Länge des aktiven S-Bandes die Zeit, die gebraucht wird: Um sie abzuschätzen, bilden wir die Summe der

aktiven Teile der k Bänder von M. Jedes kann höchstens $t(n)$ Zellen haben, da in n Schritten nur $t(n)$ Zellen verwendet werden können (wenn sich der Kopf nur nach rechts bewegt, sonst noch weniger). D.h. das Lesen des aktiven Bandteiles von S braucht höchstens $O(t(n))$. Um jeden von M's Schritten auszuführen, muss S zweimal alles lesen und höchstens k-mal nach rechts schieben. Jede der Prozeduren braucht $O(t(n))$ Schritte also braucht S um einen von M's Schritten zu simulieren $O(t(n))$.

Wir können nun zusammenrechnen: Am Anfang braucht S - um das Band ins richtige Format zu bringen - $O(n)$ Schritte. Danach simuliert S jeden der $t(n)$ Schritte von M in $O(t(n))$ Schritten, braucht also insgesamt $O(t^2(n))$ Schritte. Damit braucht S $O(n) + O(t^2(n))$. Wir haben angenommen, dass $t(n) \geq n$ (was vernünftig ist, da M ja sonst nicht einmal die Eingabe lesen könnte). D.h. die Laufzeit von S ist $O(t^2(n))$ □

Die Beziehung zwischen nichtdeterministischen und deterministischen Einbandturingmaschinen lässt sich wie folgt darstellen.

Satz 37. *Sei $t(n)$ eine Funktion mit $t(n) \geq n$. Zu jeder $t(n)$-Zeit N-Einband-Turingmaschine (nichtdeterministische) gibt es eine äquivalente $2^{O(t(n))}$-deterministische Einbandturingmaschine.*

Beweis: Sei N eine nichtdeterministische Turingmaschine mit Laufzeit $t(n)$. Wir konstruieren eine deterministische Turingmaschine D, die N simuliert, indem sie den Berechnungsbaum absucht.

Für eine Eingabe der Länge n hat der kürzeste akzeptierende Pfad von N höchstens die Länge $t(n)$. Jeder Knoten kann höchstens b Nachfolgerknoten haben, wenn b das Maximum der Verzweigungsmöglichkeiten ist, die durch Übergangsfunktion von N gegeben ist. Die Gesamtzahl der Blätter in allen Ebenen des Teilbaums bis zu einem akzeptierenden Zustand ist höchstens $b^t t(n)$. Die Simulation arbeitet den Baum in der Breite zuerst ab (also erst alle Knoten der Tiefe d, danach die der Tiefe $d+1$ usw.). Die Gesamtzahl der Knoten eines Baums (und auch Teilbaums) ist weniger als doppelt so groß als die Zahl der Blätter, damit können wir uns auf $O(b^{t(n)})$ beschränken. Um von der Wurzel bis zu einem Knoten eines Teilbaums zu gelangen, braucht man höchstens $O(t(n))$ Schritte. Damit lässt sich die Laufzeit von D abschätzen durch $O(t(n) \cdot b^{t(n)}) = 2^{O(t(n))}$. Die im Originalbeweis gegebene Maschine D hat drei Bänder, das Verwandeln in eine Einbandmaschine bewirkt höchstens eine Quadrierung der Laufzeit. Also haben wir $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$. □

Ein Zusammenhang zwischen Raum-und Zeitklassen wird im folgenden Satz konstruiert:

Satz 38. *1. $DTIME(t) \subseteq DSPACE(t)$.*

2. Wenn $s \geq \log$, dann gilt $DSPACE(s) \subseteq DTIME(2^{Lin(s)})$.

Beweis: Die erste Aussage ist unmittelbar einsehbar, wenn man bedenkt, dass eine Turingmaschine in n Schritten höchstens n Bandzellen erreichen kann.

Um die zweite Aussage zu beweisen, nehmen wir an, wir hätten eine Turingmaschine, die in der Zeit $t(n)$ und mit Platzbeschränkung $s(n)$ arbeitet. Wir nehmen an, M hat q Zustände, k Bänder und ein Eingabeband. Das Bandalphabet hat l Symbole. Für jede Eingabe w der Länge n ist die Zeitschranke $t(n)$ nach oben durch die Anzahl der verschiedenen Konfigurationen von $M(w)$ beschränkt. Denn wenn es eine Konfiguration gäbe, die zweimal in der Berechnungshistorie von M auf w auftaucht, dann würde M, die ja deterministisch arbeitet in eine Endlosschleife geraten und nicht halten, was ein Widerspruch ist. Wieviele Konfigurationen kann M auf w durchlaufen? Es gibt q mögliche Zustände, n mögliche Kopfpositionen auf dem Eingabeband, und $(s(n))^k$ mögliche Kopfpositionen auf den Arbeitsbändern und außerdem $l^{k \cdot s(n)}$ mögliche Bandinhalte. Damit ist

$$t(n) \leq q \cdot n \cdot (s(n))^k \cdot l^{k \cdot (n)}$$

Mit Hilfe von geeigneten Konstanten a, b, c und unter Annahme $s \geq \log$ können wir dies weiter abschätzen durch:

$$t(n) \leq q \cdot n \cdot (s(n))^k \cdot l^{k \cdot s(n)} \leq q \cdot 2^{\log n} \cdot 2^{a \cdot (s(n))} \leq q \cdot 2^{b \cdot s(n)} \leq 2^{c \cdot s(n)}$$

Damit haben wir gezeigt, dass t in $DTIME(2^{Lin(s)})$. □

Tabelle 15 zeigt einige der wichtigsten deterministischen und nichtdeterministischen Komplexitätsklassen. Wie bereits erwähnt, ist die logarithmische Ressourcenfunktion nur für Raumklassen, nicht aber für Zeitklassen sinnvoll, da in logarithmischer Zeit ja nicht einmal die Eingabe gelesen werden könnte. Es lässt sich zeigen, dass $PSPACE = NPSPACE$ gilt. Alle anderen Paare von Komplexitätsklassen sind jedoch entweder beweisbar verschieden, oder es ist nicht bekannt, ob sie gleich sind oder nicht.

Es ist kein Zufall, dass die Polynom- und Exponentialfunktionen als die wichtigsten Ressourcenfunktionen angesehen werden. Für praktische Zwecke ist es üblich, Algorithmen aus der Klasse P als „machbar“ oder „praktikabel“ (englisch: feasible) zu betrachten, wohingegen man Algorithmen mit exponentieller unterer Zeitschranke als „widerspenstig“ (englisch: intractable) einstuft.

Dogma: *Polynomialzeit erfasst den intuitiven Begriff der Effizienz und Exponentialzeit erfasst den intuitiven Begriff der Ineffizienz.*

Abbildung 15: Einige typische Komplexitätsklassen

Raumklassen	Zeitklassen
$L = DSPACE(\log)$	$REALTIME = DTIME(id)$
$NL = NSPACE(\log)$	$LINTIME = DTIME(Lin)$
$LINSPACE = DSPACE(Lin)$	$P = DTIME(Pol)$
$NLINSPACE = NSPACE(Lin)$	$NP = NTIME(Pol)$
$PSACE = DSPACE(Pol)$	$E = DTIME(2^{Lin})$
$NPSPACE = NPSPACE(Pol)$	$NE = NTIME(2^{Lin})$
$EXPSPACE = DSPACE(2^{Pol})$	$EXP = DTIME(2^{Pol})$
$NEXPSPACE = NSPACE(2^{Pol})$	$NEXP = NTIME(2^{Pol})$

Abbildung 16: Vergleich einiger Polynomial- und Exponentialfunktionen

$t(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
n	.00001 Sek.	.00002 Sek.	.00003 Sek.	.00004 Sek.	.00005 Sek.	.00006 Sek.
n^2	.0001 Sek.	.0004 Sek.	.0009 Sek.	.0016 Sek.	.0025 Sek.	.0036 Sek.
n^3	.001 Sek.	.008 Sek.	.027 Sek.	.064 Sek.	.125 Sek.	.256 Sek.
n^5	.1 Sek.	3.2 Sek.	24.3 Sek.	1.7 Min.	5.2 Min.	13.0 Min.
2^n	.001 Sek.	1.0 Sek.	17.9 Min.	12.7 Tage	35.7 Jahre.	366 Jhdte.
3^n	.059 Sek.	58 Min.	6.5 Jahre	3855 Jhdte	$2 \cdot 10^8$ Jhdte.	$1.3 \cdot 10^{13}$ Jhdte.

Das Dogma wird durch die Beobachtung gestützt, dass sich Polynomial- und Exponetialfunktionen signifikant in ihrer asymptotischen Wachstumsrate unterscheiden.

In Tabelle 16 sind einige Beispiele angegeben, wobei angenommen wird, dass ein Computer in einer Sekunde eine Million Instruktionen ausführen kann. Man stellt fest, dass bis zu einer Eingabelänge von $n = 60$ die durch die Polynome beschränkten Algorithmen eine erträgliche Ausführungszeit haben. Dagegen braucht schon ein durch 3^n beschränkter Algorithmus unter Umständen bereits Jahre um ein Problem mit Eingabelänge $n = 30$ zu lösen.

Die nächste Tabelle 17 illustriert, dass selbst bei einer rasanten Entwicklung der Technik und angenommener Vervielfachung der Ausführungszeiten für Exponentialzeitalgorithmen kaum eine Verbesserung zu erwarten ist. Die Tabellen stammen aus [2]. Natürlich ist ein Dogma eine Glaubenssache und als solches muss muss man es kritisch hinterfragen. Offensichtlich ist ein Algorithmus mit einer Laufzeit von $n^{10^{77}}$ – was formal ein Polynom ist, dessen Grad ungefähr gleich der geschätzten Anzahl der Atome im sichtbaren Universum ist ineffizient und nicht praktikabel, nicht einmal für eine Eingabegröße von $n = 2$. Andererseits darf eine exponentielle Schranke von $2^{0.0001 \cdot n}$ für die meisten praktisch relevanten Eingabegrößen durchaus als machbar

Abbildung 17: Vergleich einiger Polynomial- und Exponentialfunktionen

$t_i(n)$	Computer heute	100-mal schneller	1000mal schneller
$t_1(n) = n$	N_1	$100 \cdot N_1$	$1000 \cdot N_1$
$t_2(n) = n^2$	N_2	$10 \cdot N_2$	$31.6 \cdot N_2$
$t_3(n) = n^3$	N_3	$4.64 \cdot N_3$	$10 \cdot N_3$
$t_4(n) = n^5$	N_4	$2.5 \cdot N_4$	$3.98 \cdot N_4$
$t_5(n) = 2^n$	N_5	$N_5 + 6.64$	$N_5 + 9.97$
$t_6(n) = 3^n$	N_6	$N_6 + 4.19$	$N_6 + 6.29$

gelten, bevor das exponentielle Wachstum „zuschlägt“. Jedoch ist für die große Mehrheit natürlicher Probleme, die einen Algorithmus in Polynomialzeit haben, die Zeitschranke ein Polynom niedrigen Grades, wie $O(n^2)$ oder $O(n^3)$. Allerdings gibt es auch Probleme, bei denen man eine untere Schranke mit einem hohen Grad angeben kann. In vielen Fällen ist es aber so, dass, wenn man erst einmal einen polynomiellen Algorithmus gefunden hat, sich auch noch bessere Algorithmen finden lassen.

7.2 Beschleunigungs- und Kompressionssätze

In diesem Abschnitt wollen wir feststellen, wie stark eine Ressource vergrößert werden muss, damit echt mehr berechnet werden kann. Betrachten wir z.B. die Komplexitätsklasse $DTIME(t_1)$ für eine Ressourcenfunktion t_1, dann können wir fragen, wie viel stärker eine Funktion t_2 wachsen muss, damit die Ungleichheit $DTIME(t_1) \neq DTIME(t_2)$ gilt. Die beiden Sätze 40 und 41 besagen, dass ein linearer Zuwachs der gegebenen Ressourcenfunktion *nicht* genügt. Bevor wir die Sätze beweisen, wollen wir festhalten, dass es möglich ist, beliebig komplexe Probleme zu konstruieren.

Satz 39. *Für jede total berechenbare Funktion $t : \mathbb{N} \to \mathbb{N}$ gibt es ein Problem A_t, so dass $A_t \notin DTIME(t)$.*

Beweis: (indirekt, mit Diagonalisierung) Sei $M_0, M_1 \ldots$ eine Aufzählung aller DTM. Wir definieren

$$A_t := \{0^i \mid M_i \text{ akzeptiert } 0^i \text{ nicht in } t(i) \text{ Schritten}\}$$

und nehmen an, A_t wäre in $DTIME(t)$. Dann gibt es ein j mit $L(M_j) = A_t$ und $time_{M_j}(n) \leq t(n)$ für alle n. Es gilt:

$$\begin{aligned} 0^j \in A_t &\Leftrightarrow M_j \text{ akzeptiert } 0^j \text{ nicht in } t(j) \text{ Schritten} \\ &\Leftrightarrow 0^j \notin L(M_j) = A_t \end{aligned}$$

womit wir einen Widerspruch haben. Also gilt $A_t \notin DTIME(t)$. □

Nun wenden wir uns den Raumkompressions- und Beschleunigungssätzen zu.

Satz 40. *Für jede total berechenbare Funktion* $s : \mathbb{N} \to \mathbb{N}$ *gilt:*

$$DSPACE(s) = DSPACE(Lin(s)).$$

Beweis: Es genügt zu zeigen, dass $DSPACE(2 \cdot s) \subseteq DSPACE(s)$. Sei M eine DTM, die bei Eingaben w der Länge n im Raum $2s(n)$ arbeitet. Der Einfachheit halbern fordern wir, dass M nur auf Bandzellen mit gerader Nummer[8] eine Linksbewegung macht. Sei Γ das Bandalphabet. Wir konstruieren eine DTM N, die φ_M simuliert, aber im Raum $s(n)$ arbeitet. Die Idee ist, dass N einige Schritte abwartet, M beobachtet und dann mehrere Schritte von M in einem Schritt ausführt.

N bekommt dadurch mehr Zustände und braucht ein größeres Bandalphabet, nämlich $\Gamma \times \Gamma$. Zur Umsetzung wird das Band von M in Blöcke von je zwei benachbarten Bandzellen unterteilt, d.h. Zellen auf dem Band von N haben die Nummern $(2i-1, 2i)$ und ein Eintrag (a, b) auf dem Band von M wird als ein Eintrag von N aufgefasst. N simuliert das Verhalten von M auf Eingabe w, indem es dasselbe tut wie M, nur dass es den Kopf erst bewegt wenn M die Blockgrenze überschreitet. Auf diese Weise berechnet N dieselbe Funktion wie M, aber braucht nur Raum $s(n)$. □

Der lineare Beschleunigungssatz für die Zeit macht eine ganz ähnlich Aussage, allerdings kann eine Beschleunigung nur für solche Ressourcenfunktionen erreicht werden, die bis auf endlich viele Ausnahmen echt stärker wachsen, als die Identitätsfunktion.

Satz 41. *Für jede total berechenbare Funktion* $t : \mathbb{N} \to \mathbb{N}$ *mit* $id \in o(t)$ *gilt:*

$$DTIME(t) = DTIME(Lin(t)).$$

Beweis: Sei $A \in DTIME(t)$ und M eine DTM mit $L(M) = A$, wobei M bei Eingaben der Länge n in $t(n)$ Schritten arbeitet. Wir konstruieren eine DTM N mit $\varphi_M = \varphi_N$, die m-mal schneller als M ist für eine Konstante $m > 1$. Wie schon im Beweis zur Raumkompression verzögert N die Arbeit und macht dann m Schritte von M in einem Schritt. Wie oben verwenden wir dazu ein Bandalphabet, das mehrere Zeichen von M in einem Zeichen kodiert, und wieder hat N mehr Zustände als M. Wir konstruieren N in zwei Phasen: In der ersten wird die Eingabe in die komprimierte Kodierung überführt und in der zweiten die Arbeit von M simuliert. Der Einfachheit

[8] wir stellen uns die Bandzellen numeriert vor

halber benutzen wir als Maschinenmodell eine Zweibandturingmaschine, die auf einem Band nur liest und auf dem anderen arbeitet.

Phase 1: Sei $m > 1$. N kodiert das Eingabewort $w_1, \ldots w_n$ mit $w_i \in \Sigma$, indem sie es in Blöcke der Länge m aufteilt, wobei der i-te Block durch das Wort $\beta_i = w_{1+(i-1)m} w_{2+(i-1)m} \ldots w_{im}$ repräsentiert wird. Damit ist $w = \beta_1 \beta_2 \ldots \beta_{k+1}$ mit $k = \lfloor n/m \rfloor$. [9] N schreibt folgende (redundante) Kodierung auf ihr Band:

$$(\square^m, \beta_1, \beta_2)(\beta_1, \beta_2, \beta_3) \ldots (\beta_{k-2}, \beta_{k-1}, \beta_k)(\beta_{k-1}, \beta_k, \square^m)$$

Jedes Tripel wird als ein Symbol von N aufgefasst. Die erste Phase erfordert $n + k = (1 + 1/m)n$ Schritte.

Phase 2: Angenommen der aktuelle Bandinhalt von M ist ein Wort a der Länge l. Wie oben bekommt N das kodierte Wort:

$$(\square^m, \alpha_1, \alpha_2)(\alpha_1, \alpha_2, \alpha_3) \ldots (\alpha_{z-2}, \alpha_{z-1}, \alpha_z)(\alpha_{z-1}, \alpha_z, \square^m)$$

wobei z der ganzzahlige Anteil von l/m ist und das letzte Stückchen α_{z+1} wieder separat durch die Überführungsfunktion bearbeitet wird. Jetzt simuliert N die m Schritte von M auf a wie folgt. Liest M eine im Block α_j enthaltene Bandzelle, so steht der Kopf von N auf dem Symbol $(\alpha_{j-1}, \alpha_j, \alpha_{j+1})$ [10] Nach m Schritten steht der Kopf von M auf einer Bandzelle, die zu α_{j-1}, α_j oder α_{j+1}) gehört, also kann N alles in einem Schritt ausführen und geht auf das Symbol, das dem Block entspricht, in dem sich der Kopf von M nach den m Schritten befindet, und zwar:

$$\begin{array}{c}(\alpha_{j-2}, \alpha_{j-1}, \alpha_j), \text{ falls } M \text{ eine Bandzelle im Block } \alpha_{j-1} \text{ liest;} \\ (\alpha_{j-1}, \alpha_j, \alpha_{j+1}), \text{ falls } M \text{ eine Bandzelle im Block } \alpha_j \text{ liest} \\ (\alpha_j, \alpha_{j+1}, \alpha_{j+2}), \text{ falls } M \text{ eine Bandzelle im Block } \alpha_{j+1} \text{ liest}\end{array}$$

Akzeptiert oder verwirft M die Eingabe w innerhalb dieser m Schritte, so tut dies auch N. Also ist $L(M) = L(N)$. Phase 2 erfordert höchstens $\lceil t(n)/m \rceil$ Schritte, d.h. N ist in der Simulationsphase m-Mal schneller als M. Um den Gesamtaufwand zu schätzen benutzen wir nun die Annahme, dass $id \in o(t)$. Somit gilt
[11]

$$\forall c > 0 \text{ gilt } n <_{ae} c \cdot t(n)$$

[9] β_{k+1} ist genau dann leer, wenn m Teiler von n ist. Da ein nichtleeres β_{k+1} in der Steuerung von N behandelt werden kann, können wir annehmen, dass β_{k+1} leer ist.

[10] wenn $\square^n$ Teil des Symbols ist, wird analog behandelt.

[11] das *ae* unter dem Ungleichheitszeichen steht für almost everywere und Ungleichung steht hier als Ersatz für $\exists n_0 \forall n > n_0 . n < c \cdot t(n)$

Addiert man den Zeitaufwand beider Phasen, so benötigt N auf w nicht mehr als

$$\left(1+\frac{1}{m}\right)n+\left\lceil\frac{t(n)}{m}\right\rceil <_{ae} \left(1+\frac{1}{m}\right)\frac{1}{m(1+\frac{1}{m})}t(n)+\left\lceil\frac{t(n)}{m}\right\rceil \leq \left\lceil\frac{2t(n)}{m}+1\right\rceil$$

wobei die erste Ungleichung mit der spezifischen Konstante

$$c=\frac{1}{m(1+\frac{1}{m})}=\frac{1}{m+1}$$

folgt. Damit haben wir gezeigt, dass eine beliebige lineare Beschleunigung möglich ist.

□

Beispiel 21. *Sei $t(n) = d \cdot n$ für eine Konstante $d > 1$ die Laufzeitbeschränkung von M und N habe die Laufzeit*

$$T(n)=\left(1+\frac{1}{m}\right)n+\frac{t(n)}{m}=\left(1+\frac{1}{m}\right)n+\frac{d\cdot n}{m}=\left(1+\frac{d+1}{m}\right)n$$

wobei wir der Einfachheit halber annehmen, dass m sowohl n als auch $t(n)$ teilt. Da $d > 1$ gilt, impliziert die Wahl von $m > (d+1)/(d-1)$ die Ungleichung

$$T(n) < d \cdot n = t(n)$$

und somit eine echte Beschleunigung. Da $t(n) = d \cdot n$ mit $d > 1$ nicht echt schneller als die Identitätsfunktion wächst, zeigt das Beispiel, dass die Voraussetzung $id \in o(t)$ etwas stärker ist, als notwendig. Mit $d = 1$ würde der Beweis jedoch nicht funktionieren.

Tatsächlich hat Rosenberg 1967 folgendes Resultat bewiesen:

Satz 42 (Rosenberg)**.** $REALTIME \neq LINTIME$.

7.3 Hierarchiesätze

In diesem Abschnitt kehren wir zur eingangs gestellten Frage zurück: „Wie stark muss eine Ressource vergrößert werden, damit echt mehr berechnet werden kann?“

Aus den Sätzen 40 und 41 wissen wir, dass ein linearer Zuwachs der Ressourcenfunktion nicht genügt, um echt größere Klassen zu erhalten. Wir haben gesehen, dass wenn $s_2 \in O(s_1)$, dann wächst s_2 nicht stark genug, um s_1 an Berechnungskraft zu übertreffen. Wir nehmen also an, dass

$$s_1 \prec_{io} s_2 \Leftrightarrow_{def} \forall c > 0 \text{ gilt } s_1(n) <_{io} c \cdot s_2(n)$$

wobei $<_{io}$ bedeuten soll, dass an unendlich vielen Stellen (infinitely often) die linke Seite kleiner als die rechte ist. Unter dieser Voraussetzung bekommt man den unten beschriebenen Raumhierarchiesatz, der komplementär zu unserem schon bewiesenen Satz 40 ist. Damit die Beweise der Hierarchiesätze funktionieren, müssen die Ressourcenfunktionen eine technische Eigenschaft besitzen: Sie müssen *raumkonstruierbar* bzw. *zeitkonstruierbar* sein. Alle üblichen Ressourcenfunktionen wie Logarithmus, Polynome und Exponentialfunktionen sind raumkonstruierbar, und bis auf die Logarithmusfunktion auch zeitkonstruierbar.

Definition 27. *Seien f, s und t total berechenbare Funktionen.*

- *Eine Funktion s heißt* raumkonstruierbar*, falls es eine DTM M gibt, so dass für alle n gilt: M benötigt bei einer beliebigen Eingabe der Länge n nicht mehr als $s(n)$ Bandzellen, um das Wort $\sharp 1^{s(n)-2}\$$ auf das Band zu schreiben und anzuhalten, wobei $\sharp$ und $\$$ spezielle Symbole zur Markierung des linken und rechten Randes sind. Man sagt, M* hat den Raum $s(n)$ ausgelegt.
- *Eine Funktion f heißt* konstruierbar in der Zeit t*, falls es eine DTM M gibt, so dass für jedes n gilt: M arbeitet bei einer beliebigen Eingabe der Länge n genau $t(n)$ Takte und schreibt dabei das Wort $\sharp 1^{f(n)-2}\$$ auf das Band. Man sagt, t* ist zeitkonstruierbar *, falls t in der Zeit t konstruierbar ist.*

Satz 43 (Raumhierarchiesatz)**.** *Falls $s_1 \prec_{io} c \cdot s_2$ gilt und falls s_2 raumkonstruierbar ist, dann gilt:*

$$DSPACE(s_2) \not\subseteq DSPACE(s_1)$$

Beweis: Der Satz wird nur für den Fall $s_1 \geq log$ bewiesen.[12] Wir konstruieren eine Menge A in der Differenz $DSPACE(s_1) \setminus DSPACE(s_2)$ durch Diagonalisierung. Sei $M_0, M_1, M_2, \ldots$ eine Aufzählung aller Einbandturingmaschinen. Wir definieren eine DTM N mit einem Eingabeband und drei Arbeitsbändern wie folgt:

„Auf Eingabe $w \in \{0,1\}^*$ der Länge n

1. N legt den Raum $s_2(n)$ auf allen Arbeitsbändern aus.
2. Sei $w = 1^i y$, wobei $0 \leq i \leq n$ und $y \in \{\epsilon\} \cup 0\{0,1\}^*$, d.h w beginnt mit einem(womöglich leeren) Präfix von Einsen, gefolgt entweder vom leeren Wort oder von einer 0 und einem (womöglich leeren) Wort. N interpretiert i als Maschinennummer und schreibt das geeignet kodierte Programm von M_i auf das erste Arbeitsband. Falls das Programm größer als $s_2(n)$ ist, *reject*. Sonst simuliert N das Verhalten von M_i auf w auf dem zweiten Band.
3. Das dritte Band enthält einen Binärzähler, der anfangs den Wert 0 enthält und in jedem Schritt der Simulation von M_i um eins erhöht wird. Ist die Simulation von M_i beendet bevor der Zähler überläuft, *accept*, wenn M_i akzeptiert, sonst *reject*.

[12] Mit einem Resultat von Sipser[**?**] kann man die eigentlich unnötige Einschränkung loswerden.

“

Technische Erläuterungen:

- Der Zähler garantiert, dass N immer anhält
- Es gibt eine Konstante c, so dass die Simulation von M_i auf dem zweiten Band mit Platzbeschränkung $c \cdot space_{M_i}$ gelingt. Der Grund dafür ist, dass N in der Lage sein muss, jede DTM M_i zu simulieren. Wenn M_i für ein i insgesamt z_i Zustände und l_i Symbole in ihrem Bandalphabet hat, dann kann N diese Symbole und Zustände binär als Wörter der Länge $\lceil \log z_i \rceil$ bzw. $\lceil \log l_i \rceil$ codieren. Diese Kodierung verursacht zusätzlich konstante Raumkosten für die simulierende Maschine N, wobei die Konstante c_i nur von M_i abhängt.

Definiere $A := L(N)$. Offenbar ist A in $DSPACE(s_2)$. Wir nehmen an, $A \in SPACE(s_1)$. Dann existiert ein i mit $A = L(M_i)$ und $space_{M_i}(n) \leq s_1(n)$. Nach Voraussetzung gilt $s_1 \prec_{io} s_2(n)$, d.h.

$$\forall c > 0 \text{ gilt } s_2(n) >_{io} c \cdot s_1(n)$$

Also gibt es eine reelle Konstante $c > 0$ und unendlich viele Zahlen $n_1, n_2, \ldots \in \mathbb{N}$ mit $s_2(n_k) > c \cdot s_1(n_k)$. Unter diesen wähle man ein n_j so, dass folgende drei Bedingungen erfüllt sind:

(i) Das Programm von M_i kann im Raum $s_2(n_j)$ berechnet und auf das zweite Arbeitsband geschrieben werden.

(ii) Die Simulation von M_i auf $1^i 0^{n_j - i}$ gelingt im Raum $s_2(n_j)$.

(iii) $time_{M_i}(n_j) \leq 2^{s_2(n_j)}$

Bedingung (i) kann für ein hinreichend großes n_j erfüllt werden, da die Größe des Programms von M_i nicht von der Eingabe der Maschine abhängt.

Bedingung (ii) kann erfüllt werden, denn die Simulation von M_i auf Eingabe $1^i 0^{n_j - i}$ gelingt im Raum

$$c_i \cdot space_{M_i}(n_j) \leq c_i \cdot s_1(n_j) < s_2(n_j)$$

wobei c_i die Konstante ist, die durch die Kodierung von M_i entsteht.

Bedingung (iii) kann für ein hinreichend großes n_j erfüllt werden, da aus Satz 38 für$s_1 \geq log$ folgt:

$$\begin{aligned} time_{M_i}(n_j) &\leq 2^{d \cdot space_{M_i}(n_j)} \text{ für geeignete Konstante } d \\ &\leq 2^{d \cdot s_1(n_j)} \\ &< 2^{s_2(n_j)}, \end{aligned}$$

wobei die letzte Ungleichung aus der Voraussetzung des Satzes folgt. Also gelingt die Simulation von M_i auf $1^i 0^{n_j - i}$, bevor der Binärzähler der Länge $s_2(n_j)$ überläuft.

Aus allen drei Bedingungen und der Konstruktion von N folgt für das Wort $w = 1^i 0^{n_j - i}$

$$w \in A \Leftrightarrow N \text{ akzeptiert } w \Leftrightarrow M_i \text{ lehnt ab.}$$

Also ist $A \neq L(M_i)$ im Widerspruch zur Annahme. Es folgt: $A \notin DSPACE(s_1)$
□
Eigentlich liefert der Beweis ein stärkeres Resultat:

Folgerung:

$$DSPACE(s_2) \nsubseteq \bigcup_{s_1 <_{io} s_2} DSPACE(s_1)$$

Die Vereinigung wird über alle Funktionen genommen, die an unendlich vielen Stellen asymptotisch echt kleiner als s_2 sind. Und unmittelbar daraus sieht man, dass für Ressorcenfunktionen mit echt größerem Wachstum die zugehörigen Raumklassen echt ineinander enthalten sind.

Folgerung: Falls $s_1 \leq s_2, s_1 \prec_{io} s_2$ und s_2 ist raumkonstruierbar, dann gilt:

$$DSPACE(s_1) \subset DSPACE(s_2)$$

Wir definieren $POLYLOGSPACE = \bigcup_{k \geq 1} DSPACE((\log n)^k)$, dann folgt aus dem Korollar eine echte Hierarchie der Raumklassen wie folgt:

L $\subset$ **POLYLOGSPACE** $\subset$ **LINSPACE** $\subset$ **PSPACE** $\subset$ **EXPSPACE**.

Ein analoges Resultat gibt es auch für die Zeitkomplexität.

Satz 44 (Zeithierarchiesatz)**.** *Sei $t_2 \geq id, t_1 \prec_{io} t_2$ und sei t_2 in der Zeit $t_2 \log t_2$ konstruierbar, so gilt*

$$DTIME(t_2 \log t_2) \nsubseteq DTIME(t_1).$$

Beweisidee: Der Beweis beruht wieder auf Diagonalisierung. Ausgehend von einer Aufzählung der DTM muss die diagonalisierende DTM N, die in $t_2 \log t_2$ arbeitet, jede Maschine M_i „schlagen“ d.h. wenn M_i in t_1 arbeitet, dann haben M und N verschiedene Sprachen. Für eine genaue Konstruktion von N braucht man einige technische Details, die an dieser Stelle aus Platzgründen nicht ausgeführt werden sollen.

Wie schon der Satz über Raumkomplexität hat auch dieser über Zeitkomplexität einige interessante Folgerungen:

Folgerung:

1. Falls $t_1 \leq t_2 \log t_2$ und $t_2 \geq id$ und $t_1 \prec_{io} t_2$ und t_2 in der Zeit $t_2 \log t_2$ konstruierbar ist, dann gilt $DTIME(t_1) \subset DTIME(t_2 \log t_2)$.

2. Für jede Konstante $k > 0$ gilt $DTIME(n^k) \subset DTIME(n^{k+1})$ und $DTIME(2^{k \cdot n}) \subset DTIME(2^{(k+1) \cdot n})$.

3. $P \subset E \subset EXP$.

8 Die Zeitkomplexitätsklassen P und NP

8.1 Die Klasse P

In Tabelle 15 haben wir $P = DTIME(Pol)$ als Klasse der Probleme kennengelernt, die sich in polynomieller Zeit lösen lassen.

Mit den Sätzen 36 und 37 haben wir auf einen bedeutenden Unterschied aufmerksam gemacht. Auf der einen Seite haben wir höchstens einen polynomiellen Zeitzuwachs beim Übergang von einer deterministischen Einband- zu einer deterministischen Mehrbandturingmaschine. Auf der anderen Seite ist der Unterschied der Laufzeit beim Übergang von einer nichtdeterministischen zu einer deterministischen Turingmaschine exponentiell.

Aus dem Zeithierarchiesatz geht hervor, dass die Klasse P echt verschieden ist von $E = DTIME(2^{Lin})$, da jedes Polynom asymptotisch echt kleiner ist als jede Exponentialfunktion. In unserem Dogma auf Seite 95 haben wir festgelegt, dass wir die Klasse der Polynomialzeit-beschränkten Algorithmen als die praktikablen betrachten wollen, d.h. als diejenigen, die in der Praxis benutzt werden können. In folgenden Abschnitt werden wir stellvertretend für die Algorithmen dieser Klasse drei Beispiele von P-Algorithmen betrachten.

Wir werden wegen der Übersichtlichkeit wieder high-level-Beschreibungen verwenden. Algorithmen werden mit numerierten Teilen (Stadien) beschrieben. Ein Stadium eines Algorithmus' ist ähnlich wie der Schritt einer Turingmaschine, außer dass seine Implementierung in einer Turingmaschine im Allgemeinen mehrere Schritte benötigt. Um festzustellen, dass ein Algorithmus in P liegt, müssen wir erstens eine polynomielle obere Schranke für die Anzahl der nötigen Stadien angeben und zweitens die einzelnen Stadien des Algorithmus daraufhin überprüfen, ob sie in Polynomialzeit ausgeführt werden können.

Bei der Beschreibung des Algorithmus wählen wir die Stadien so, dass die Analyse einfach ist. Sind beide Teile der Analyse erfolgreich abgeschlossen, dann wissen wir, dass der Algorithmus in polynomieller Zeit läuft, da jeder der polynomiell vielen Schritte in polynomieller Zeit läuft und Komposition von Polynomen wieder Polynome ergibt.

Ein Punkt bedarf noch genauerer Aufmerksamkeit. Es geht um die Kodierung von Eingaben. Wir werden weiterhin die Klammer-Notation <> verwenden ohne eine spezielle Kodierungsmethode damit zu bezeichnen. Unter einer „vernünftigen“ Kodierungsmethode wollen wir ab jetzt eine solche Darstellung von Objekten verstehen, die in polynomieller Zeit die Umrechnung in eine interne natürliche-Zahl-Darstellung erlaubt. Die bekannten Kodierungen von Graphen, Automaten usw. sind unter diesen Umständen vernünftig.

Die Kodierung von natürlichen Zahlen als unäre Darstellung $111\ldots11$ hingegen ist nicht vernünftig, da sie exponentiell größer als wirklich vernünftige Eingaben wie die binäre Darstellung ist. Viele Probleme, die wir als Beispiele sehen werden, enthalten Kodierungen von Graphen. Eine vernünftige Kodierung eines Graphen ist eine Liste der Knoten und Kanten. Eine andere ist die durch eine Nachbarschaftsmatrix (adjacency matrix), wobei der Eintrag an der Stelle (i, j) eine Eins ist, wenn es eine Kante zwischen den Knoten i und j gibt und eine Null, falls nicht. Bei der Analyse von Graphenalgorithmen gibt man die Laufzeit in Abhängigkeit von der Anzahl der Knoten an, anstelle der Länge der Darstellung, die bei vernünftigen Darstellungen polynomiell in der Anzahl der Knoten ist.

Das erste Problem, das wir als Beispiel für einen polynomiellen Algorithmus untersuchen wollen, betrifft gerichtete Graphen. Seien s und t zwei Knoten in einem gerichteten Graphen. Das Pfadproblem ist die Frage, ob in einem gerichteten Graphen ein gerichteter Pfad von s nach t existiert.

$$PATH = \{< G, s, t > \mid G \text{ ist ein gerichteter Graph, in dem es einen gerichteten Pfad von } s \text{ nach } t \text{ gibt.}\}$$

Satz 45. $PATH \in P$.

Beweisidee: Wir geben einen Polynomialzeitalgorithmus an, der $PATH$ entscheidet. Bevor wir das tun, überzeugen wir uns davon, dass der naive Algorithmus nicht funktioniert (nicht schnell genug ist). Eine brute-force Methode würde alle potentiellen Pfade in G daraufhin untersuchen, ob sie Pfade von s nach t sind. Ein potentieller Pfad ist ein Pfad von höchstens der Länge m (wenn m die Anzahl der Knoten ist). Falls ein Pfad von s nach t existiert, dann gibt es auch einen, der höchstens Länge m hat, da Wiederholungen von Knoten nicht notwendig sind. Die Zahl der potentiellen Pfade[13] ist kleiner als m^m, was exponentiell in der Anzahl der Knoten von G ist.

Um einen Polynomialzeitalgorithmus zu bekommen, muss man etwas anderes als brute-force-Suche finden. Man verwendet stattdessen Breitensuche. Dazu markiert man zunächst alle Knoten, die von s aus in einem Schritt, danach die in zwei und drei Schritten usw. bis in m Schritten erreichbar sind. Die Laufzeit dieses Verfahrens durch ein Polynom abzuschätzen, ist einfach.

[13] Die Anzahl der Pfade der Länge m ist $m!$ Anzahl der Pfade der Länge $m-1$ ist $(m-1)!$ mit Hilfe der Stirling-Formel $n! \backsimeq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ kommt man auf die Abschätzung m^m.

Beweis: Ein polynomieller Algorithmus M für $PATH$ arbeitet wie folgt: $M =$

„Auf Eingabe $< G, s, t >$, wobei G ein gerichteter Graph und s und t Knoten sind:

1. Markiere s.
2. Wiederhole Schritt 3 bis kein neuer Knoten mehr markiert wird.
3. Durchsuche alle Kanten von G: wenn eine Kante (a, b) dabei ist, bei der a markiert ist und b unmarkiert, dann markiere b.
4. Falls t markiert ist, *accept*, sonst *weise ab*.

“

Wir analysieren jetzt die Laufzeit des Algorithmus, um zu zeigen, dass es dafür eine polynomielle Schranke gibt.

Offenbar werden die Stadien 1 und 4 genau einmal ausgeführt. Stadium 3 wird höchstens m Mal ausgeführt, da, bis auf das letzte Mal, jedes Mal höchstens ein zusätzlicher Knoten markiert wird. Damit ist die Gesamtzahl der möglichen Schritte $1 + 1 + m$, was ein Polynom in der Länge der Knotenanzahl von G ist.

Zustände 1 und 4 können einfach in Polynomialzeit implementiert werden, 3 benötigt eine Prüfung der Eingabe und Tests, ob bestimmt Knoten markiert sind, was ebenfalls in polynomieller Zeit gemacht werden kann. □

Als nächstes betrachten wir ein arithmetisches Problem.

$$RELPRIME = \{< x, y > | x \text{ und } y \text{ sind teilerfremd}\}$$

Satz 46. $RELPRIME \in P$.

Beweisidee: Da in Dual- oder anderer Basis die Größe der Zahl exponentiell mit der Länge der Eingabe wächst, ist der Brute-Force-Algorithmus, der der Reihe nach alle möglichen Teiler für beide Zahlen ausprobiert, exponentiell. Stattdessen verwendet man den Euklidischen Algorithmus. Falls ggT $(x, y) = 1$, dann ist x relativ prim zu y. Zuerst betrachten wir den Algorithmus am Beispiel.

Beispiel 22. *Wir wollen den größten gemeinsamen Teiler von* 144 *und* 60 *bestimmen.* $x : 144 \mod 60 = 24$
$x : 60 \mod 24 = 12$
$x : 24 \mod 12 = 0$
$G.g.T(144, 60) = 12$

Dabei ist $x \mod y$ die Funktion, die den Rest von x bei Division durch y berechnet. Der Euklidische Algorithmus funktioniert wie folgt:
$E =$

„Auf Eingabe $< x, y >$, wobei x und y natürliche Zahlen in Binärdarstellung sind:

1. Wiederhole bis $y = 0$:
2. Berechne $x := x \mod y$.
3. Vertausche x und y.
4. Gebe x aus.

“

Beweis: Der Algorithmus R löst $RELPRIME$ mit E als Unterprogramm
$R =$:

„Auf Eingabe $< x, y >$, wobei x und y natürliche Zahlen in Binärdarstellung sind:

1. Starte E auf $< x, y >$.
2. Falls das Ergebnis 1 ist, *accept*, sonst *reject*.

“

Offenbar läuft R in Polynomialzeit mit dem richtigen Ergebnis, wenn das bei E der Fall ist. Die Korrektheit von E ist bekannt, also brauchen wir uns nur um die Laufzeit zu kümmern. Zuerst überlegen wir uns, dass jede Ausführung von Schritt 2 den aktuellen Wert von x mindestens durch 2 teilt.

Nach Ausführung von Schritt 2 ist $x < y$, wegen Definition von $\mod$. Nach Schritt 3 gilt dann also $x > y$, weil die beiden vertauscht wurden. Wenn Schritt 2 immer hintereinander ausgeführt wird, ist $x > y$. Wenn $x/2 \geq y$, dann gilt $x \mod y < y \leq x/2$ und x wird mindestens halbiert.

Wenn $x/2 < y$, dann $x \mod y = x - y < x/2$ und x wird mindestens halbiert.

Da die Werte von x und y in Schritt 3 jeweils vertauscht werden, werden beide immer wieder halbiert. D.h. die maximale Anzahl der Schleifen ist das Minimum von $log_2 x$ und $log_2 y$. Die Logarithmen sind proportional zur Länge der Darstellung und damit ist die Anzahl der Schritte $O(n)$. Jeder Schritt von E braucht nur polynomielle Zeit, damit ist die Gesamtlaufzeit auch polynomiell. □

Als letztes Beispiel eines polynomiellen Algorithmus' betrachten wir einen Entscheidungsalgorithmus für kontextfreie Sprachen.

Satz 47. *Jede kontextfreie Sprache liegt in P.*

Beweisidee: In Satz 9 wurde Entscheidbarkeit von A_{CFG} bewiesen. Der naive Algorithmus, der für ein Wort w der Länge n (das $2n-1$ Ableitungsschritte hat) alle möglichen Ableitungen durchsucht, ist exponentiell in n. Statt mit diesem, löst man das Problem mit dynamischer Programmierung. Man trägt nach und nach in einer $n \times n$ Tabelle alle Variablen ein, die zur Erzeugung der Teilwörter beitragen (Cocke-Younger-Kasami-Verfahren). An der Stelle (i, j) stehen die Variablen, die das Wort $w_i \dots w_j$ erzeugen. Man fängt mit den Teilwörtern der Länge 1 an und erzeugt nach und nach den Rest. Für Wörter der Länge $k+1$ werden alle möglichen Zerlegungen in zwei (nichtleere) Teilwörter gemacht und alle Regeln der Form $A \to BC$ daraufhin untersucht, ob B im Eintrag für das erste Teilwort und C im Eintrag für das zweite Teilwort steht.

Beweis: Sei G eine kontextfreie Grammatik in Chomsky Normalform, die die Sprache L erzeugt.
$D =$

„Auf Eingabe $w = w_1 \dots w_n$:

1. Falls $w = \epsilon$ und $S \to \epsilon \in G$ *accept.*
2. Für jedes $i = 1$ bis n:
3. Für jede Variable A,
4. Teste, ob $A \to b \in G$ mit $b = w_i$.
5. Wenn ja, füge A zu $table(i, i)$ hinzu.
6. Für $l = 2$ bis n:
7. Für $i = 1$ bis $l - 1$:

8. Sei $j = i + l - 1$

9. Für $k = i$ bis j–1

10. Für jede Regel $A \to BC$:

11. Wenn $B \in table(i, k)$ und $C \in table(k + 1, j)$ dann füge A zu $table(i, j)$ hinzu.

12. Falls $S \in table(1, n)$, *accept*; wenn nicht *reject*.

“

Jeder Schritt läuft in Polynomialzeit, dabei werden Schritt 4 und 5 höchstens nv–mal ausgeführt, wobei v die Zahl der Nichtterminale in G ist, also eine Konstante und unabhängig von n. Also werden diese Schritte $O(n)$-mal ausgeführt. Schritt 6 wird höchstens n-mal ausgeführt. Jedes Mal wenn 6 ausgeführt wird, läuft 7 höchstens n-mal. Bei jeder Ausführung von 7 werden 8 und 9 höchstens n-mal ausgeführt. Jedes Mal wenn 9 läuft, läuft 10 r-mal, wobei r die Zahl der Regeln in G ist (also eine Konstante, die nicht von der Größe der Eingabe abhängt). Damit wird Schritt 11, die innere Schleife des Algorithmus in $O(n^3)$ ausgeführt, also hat D eine Laufzeit von $O(n^3)$. □

8.2 Die Klasse NP

Wie wir im vergangenen Abschnitt gesehen haben, lässt sich für viele Algorithmen Vollsuche vermeiden und man kann polynomielle Lösungen bekommen. Bisher ist jedoch für eine ganze Reihe von interessanten Problemen die Suche nach Ersatz für Vollsuche nicht erfolgreich gewesen, und man weiß nicht, ob es Polynomialzeitalgorithmen gibt.

Warum konnte man bisher keine solchen Lösungen finden? Die Antwort auf diese Frage ist nicht bekannt. Vielleicht gibt es für diese Probleme Polynomialzeitalgorithmen, die auf bisher unbekannten Prinzipien beruhen. Möglicherweise können sie aber einfach nicht in polynomieller Zeit entschieden werden. Eine wichtige Entdeckung bei der Betrachtung dieser Frage ist, dass die Komplexität vieler Probleme miteinander verbunden ist. Um dieses Phänomen zu verstehen betrachten wir ein Beispiel.

Beispiel 23. *Ein Hamiltonscher Pfad in einem gerichteten Graphen G ist ein gerichteter Pfad, der genau einmal durch jeden Knoten geht. Wir betrachten das Problem, ob ein gegebener Graph einen solchen Hamiltonschen Pfad*

zwischen zwei bestimmten Punkten besitzt. Sei

$$HAMPATH = \{< G, s, t > \mid G \textit{ ist gerichteter Graph mit einem Hamiltonschen Pfad zwischen } s \textit{ und } t\}$$

Man kann auf einfache Weise einen exponentiellen Algorithmus für $HAMPATH$ finden, indem man den Algorithmus aus Satz 45 modifiziert. Man muss nur eine Prüfung, ob der gefundene Pfad hamiltonsch ist, hinzufügen. Mit diesem Verfahren hat man aber exponentiellen Zeitaufwand. Bisher ist nicht bekannt, ob sich $HAMPATH$ auch in polynomieller Zeit lösen lässt.

Interessanterweise hat $HAMPATH$ eine Eigenschaft, die sich Polynomielle Verifizierbarkeit nennt. Diese Eigenschaft ist wichtig, um die Komplexität von Algorithmen zu verstehen. Auch wenn wir keinen schnellen (d.h. Polynomialzeit-)Algorithmus haben, um zu entscheiden, ob ein Graph einen Hamiltonschen Pfad hat, könnten wir, wenn wir von irgenwoher einen Pfad gegeben hätten, in polynomieller Zeit bestimmen, ob er eine Lösung ist. Mit anderen Worten: Das Verifizieren einer Lösung ist viel einfacher als zu bestimmen, ob es eine Lösung gibt.

Ein anderes polynomiell verifizierbares Problem ist Zerlegbarkeit. Wir erinnern uns, das eine natürliche Zahl *zusammengesetzt* ist, wenn sie das Produkt von zwei natürlichen Zahlen größer als 1 ist, d.h. wenn sie keine Primzahl ist. Sei

$$COMPOSITES = \{x \mid x = pq \text{ für ganze Zahlen } p, q > 1\}$$

(Seit 2004 ist bekannt, dass $PRIMES$ in P liegt.) Es gibt auch Probleme, die nicht polynomiell verifizierbar sind, z.B. $\overline{HAMPATH}$. Auch wenn jemand behauptet, dass ein bestimmter Graph keinen Hamilton-Pfad enthält, ist kein Algorithmus bekannt, mit dem man in polynomieller Zeit feststellen kann, ob die Behauptung richtig ist.

Definition 28. *Ein* Verifizierer *für eine Sprache A ist ein Algorithmus V, wobei $A = \{w \mid V$ akzeptiert $< w, c >$ für eine Zeichenkette $c\}$.*

Die Laufzeit eines Verifizierers wird immer in Abhängigkeit von der Länge von w angegeben, d.h. ein Polynomialzeit-Verifizierer läuft in Polynomialzeit in Abhängigkeit von der Länge von w.

Eine Sprache heißt polynomiell verifizierbar, wenn es einen Polynomialzeit-Verifizierer für sie gibt.

Ein Verifizierer benutzt (neben den Eingaben für das Problem) zusätzliche Information (in der Definition repräsentiert durch c). Diese Information heißt Zertifikat oder Beweis (Zeuge) für Enthaltensein in A.

Bemerkung: Für polynomielle Verifizierer hat das Zertifikat polynomielle Länge (in der Länge von w), da ein Verifizierer in polynomieller Zeit auch nicht mehr lesen kann. Für $HAMPATH$ ist ein Hamiltonscher Pfad von s nach t ein Zertifikat. Für $COMPOSITES$ ist einer der Teiler das Zertifikat.

Wir definieren die Klasse der in Polynomialzeit verifizierbaren Sprachen als NP. In Satz 48 werden wir sehen, dass die so definierte Klasse NP mit der bereits definierten Klasse $NTIME(Pol)$ übereinstimmt.

Definition 29. *NP ist die Klasse der Sprachen, die polynomielle Verifizierer besitzen.*

Wir geben eine nichtdeterministische Turingmaschine für $HAMPATH$ an:
$N_1 =$

„Auf Eingabe $< G, s, t >$, wobei G ein gerichteter Graph mit Knoten s und t ist:

1. Schreibe eine Liste von m Zahlen $p_1 \dots p_m$, wobei m die Zahl der Knoten in G ist (jede Zahl wird nichtdeterministisch zwischen 1 und m gewählt.
2. Überprüfe, ob es Wiederholungen gibt, wenn ja *reject*.
3. Prüfe, ob $s = p_1$ und $t = p_m$; falls eine der Bedingungen nicht gilt, lehne ab.
4. Für jedes i zwischen 1 und $m - 1$ prüfe, ob (p_i, p_{i+1}) eine Kante von G ist, falls ein Paar keine Kante von G ist, lehne ab; sonst *accept*.

“

Nach Definition 23 ist die Laufzeit $NTime_M(w)$ einer nichtdeterministischen Turingmaschine M auf einem Wort w, die Anzahl der Schritte des kürzesten akzeptierenden Pfades.

N_1 läuft offenbar in nichtdeterministischer Polynomialzeit. Schritt 1 ist polynomiell. In 2 + 3 wird nur ein einfacher Test ausgeführt, also haben wir zusammen polynomielle Zeit. Schritt 4 ist ebenfalls polynomiell.

Satz 48. *Eine Sprache liegt in genau dann NP , wenn sie durch eine nichtdeterministische Turingmaschine in Polynomialzeit entschieden werden kann.*

Beweisidee: Wir zeigen, wie man einen Polynomialzeit-Verifizierer in eine nichtdeterministische Turingmaschine umwandelt und umgekehrt. Die nichtdeterministische Turingmaschine simuliert den Verifizierer durch Raten

des Zeugen. Der Verifizierer simuliert die nichtdeterministische Turingmaschine, indem es den akzeptierenden Pfad als Zeugen verwendet.

Beweis:

Sei $A \in NP$ und V der Verifizierer für A. Wir nehmen an, V ist eine Turingmaschine und läuft bei Eingabelänge n in n^k für eine Konstante k. Wir konstruieren eine nichtdeterministische Turingmaschine N, die A entscheidet.
$N =$

„Auf Eingabe w der Länge n:

1. Wähle nichtdeterministisch ein Wort c der Länge n^k.
2. Starte V mit der Eingabe $< w, c >$.
3. Falls V akzeptiert, *accept*, sonst *reject*.

“

Um die andere Richtung zu beweisen, nehmen wir an, wir hätten eine nichtdeterministische Turingmaschine, die A in polynomieller Zeit entscheidet, und konstruieren einen polynomiellen Verifizierer V.

$V =$

„Auf Eingabe $< w, c >$, wobei w und c Wörter sind:

1. Simuliere N auf der Eingabe w und behandle dabei jedes Symbol von c als Beschreibung einer nichtdeterministischen Entscheidung in jedem Schritt wie in Satz 37
2. Falls dieser Zweig von N akzeptiert, *accept*; sonst *reject*.

“

□

Folgerung: $NP = NTIME(Pol)$

Die Klasse NP hängt nicht von der Wahl des Berechenbarkeitsmodells ab. Bei der Beschreibung und Analyse von NP-Algorithmen verwenden wir dieselben Konventionen wie bei polynomiellen Algorithmen. Jeder Schritt eines solchen Algorithmus' muss sich auf einem nichtdeterministischen Modell polynomiell implementieren lassen.

8.2.1 Beispiele für NP-Probleme

Definition 30. *Sei G ein ungerichteter Graph, dann ist eine* Clique *ein Teilgraph, bei dem je zwei Knoten durch eine Kante verbunden sind. Eine k-Clique ist eine Clique mit k Knoten.*

Das Cliquen-Problem besteht darin festzustellen, ob ein Graph eine Clique enthält.

$$CLIQUE = \{< G, k > | G \text{ ist ein ungerichteter Graph, der eine } k\text{-Clique enthält } \}$$

Satz 49. *$CLIQUE$ liegt in NP.*

Beweisidee: die Clique selbst ist das Zertifikat bzw. der Zeuge.
Beweis:

„Auf Eingabe $<< G, k >, c >$, wobei k eine Zahl und c ein String ist:

1. Prüfe, ob c eine Menge von k Knoten in G ist.
2. Prüfe, ob alle Verbindungskanten von c in G enthalten sind.
3. *Akzeptiere*, wenn 1 und 2 erfolgreich sind; sonst *reject*.

“

□

Der alternative Beweis durch Angabe einer nichtdeterministischen Turingmaschine N, die $CLIQUE$ entscheidet:

„Auf Eingabe $< G, k >$, wobei G ein Graph und $k \in \mathbb{N}$

1. Wähle nichtdeterministisch eine Teilmenge c mit k Knoten aus der Knotenmenge von G.
2. Überprüfe, ob alle Kanten, die Knoten von c verbinden, in G enthalten sind.
3. Wenn ja, *accept*, wenn nein, *lehne ab.*

“

Als nächstes betrachten wir ein Problem der Integer-Arithmetik. Wir haben eine Menge von Zahlen sowie einen Wert gegeben und fragen, ob es möglich ist, dass sich eine Teilmenge der Zahlen zu genau diesem Wert aufaddieren lässt. Dabei werden sowohl die Menge als auch die Teilmenge als Multimenge aufgefasst, d.h. die Elemente dürfen mehrfach auftreten.

$$SUBSET - SUM = \{< S, t > | S = \{x_1 \dots x_k\} \text{ und für eine Menge} \quad (4)$$
$$\{y_1, \dots, y_c\} \subseteq \{x_1, \dots, x_k\} \text{ gilt } \Sigma_i y_i = t\} \quad (5)$$

Satz 50. *$SUBSET - SUM$ liegt in NP.*

Beispiel 24. *$: < \{4, 11, 16, 21, 27\}, 25 > \in SUBSET - SUM$ wegen $4+21 = 25$.*

Beweis: Wir geben einen Verifizierer V für $SUBSET - SUM$ an.
$V =$

„Auf Eingabe $<< S, t >, c >$, wobei S eine Liste von Zahlen und t eine Zahl ist :

1. Prüfe, ob c eine Menge von Zahlen ist, die sich zu t summieren, wenn nicht
2. Prüfe, ob S alle in c vorkommenden Zahlen enthält, wenn nicht *reject*; sonst *accept*.

“

Alternativer Beweis: Wir geben eine nichtdeterministische Turingmaschine N an, die $SUBSET - SUM$ entscheidet. N

„Auf Eingabe $< S, t >$:

1. Wähle nichtdeterministisch eine Teilmenge c aus S aus.
2. Prüfe, ob die Zahlen von c sich zu t aufaddieren, wenn nicht *reject*, sonst *accept*.

“

□

Bemerkung: Man sieht nicht ohne weiteres, ob die Komplemente dieser Mengen, d.h. $\overline{CLIQUE}$ und $\overline{SUBSET - SUM}$ in NP liegen. Festzustellen,

dass irgendetwas nicht existiert (vorhanden ist), scheint schwieriger zu sein als festzustellen, dass etwas vorhanden ist. Wir führen eine weitere Komplexitätsklasse ein.

Definition 31. $co-NP$ *ist die folgende Klasse von Problemen:*

$$co-NP = \{L | \overline{L} \in NP\}$$

8.2.2 Die P-ungleich- NP-Frage

- P = Klasse der Sprachen, bei denen Zugehörigkeit schnell entschieden werden kann.
- NP = Klasse der Sprachen, bei der Zugehörigkeit schnell überprüft werden kann.

Wir haben die Beispiele $HAMPATH$ und $CLIQUE$ gesehen, die in NP liegen, von denen wir aber nicht wissen, ob sie in P liegen. Anscheinend ist NP größer als P. Aber die beiden Klassen könnten auch gleich sein. Wir können es nicht beweisen. Bisher ist kein Problem bekannt, das in NP liegt, aber mit Sicherheit nicht in P. „$P = NP$?“ ist die wichtigste ungelöste Frage der theoretischen Informatik. Die meisten Forscher glauben $P \neq NP$, da andernfalls jedes polynomiell verifizierbare Problem auch polynomiell entscheidbar wäre und damit wäre Beweisen (Beweissuche) genauso schwer wie Beweisverifikation. Man hat sowohl versucht polynomielle Algorithmen für bestimmte NP-Probleme zu finden, als auch zu zeigen, dass die Klassen ungleich sind, d.h. dass es für bestimmte Probleme keinen polynomiellen Algorithmus geben kann, der Vollsuche ersetzt. Die beste (allgemein anwendbare) Methode NP-Probleme deterministisch zu lösen braucht exponentielle Zeit. Man kann $NP \subseteq EXPTIME = \bigcup_k DTIME(2^{n^k})$ zeigen, aber es ist nicht bekannt, ob es eine kleinere Klasse gibt, in der NP enthalten ist.

9 NP-Vollständigkeit

In den frühen 70er Jahren veröffentlichten Stephen Cook und Leonid Levin Arbeiten, die später zu einem Fortschritt in der P-ungleich-NP-Frage geführt haben. Die beiden entdeckten bestimmte Probleme, deren individuelle Komplexität mit der der ganzen Klasse in Beziehung steht. Falls polynomielle Algorithmen für eins dieser Probleme existieren, dann wären alle NP-Probleme in P. Die Probleme dieser Klasse heißen NP-vollständig. NP-Vollständigkeit ist sowohl vom praktischen als auch vom theoretischen Aspekt her wichtig. Als Theoretiker konzentriert man sich auf NP-Probleme um $P = NP$? Zu

lösen. Wenn man von einem NP-Problem sagen kann, dass es mehr als polynomielle Zeit braucht, dann von einem NP-vollständigen. Umgekehrt muss man einen P-Algorithmus finden, wenn man zeigen will, dass $P = NP$. Auf der praktischen Seite kann man sich die Mühe, einen effizienten Algorithmus zu finden, sparen, wenn man ein NP-vollständiges Problem vor sich hat. Auch wenn wir nicht zeigen können, dass es für ein bestimmtes Problem keinen effizienten Algorithmus geben kann, ist die Tatsache, dass es NP-vollständig ist ein starkes Indiz dafür, dass er gar nicht existiert (wenn man $P \neq NP$ annimmt).

Beispiel 25. *Als erstes Beispiel eines NP-vollständigen Problemes betrachten wir das* Erfüllbarkeits- bzw. SAT-Problem. *Wir erinnern uns, dass Variablen, die die Werte $TRUE$ and $FALSE$ annehmen können, als Boolesche Variablen bezeichnet werden. Wie üblich stellen wir $TRUE$ und $FALSE$ durch 1 und 0 dar. Die Booleschen Operatoren AND, OR und NOT (mit den üblichen Wertetabellen) werden durch $\wedge, \vee$ und $\neg$ dargestellt. Außerdem benutzen wir einen Oberstrich als verkürzte Schreibweise für das $\neg$-Symbol, d.h $\overline{x}$ steht für $\neg x$.*

Eine Boolesche Formel[14] *ist ein Ausdruck, der aus Boolschen Variablen und Operatoren besteht, wie z.B.*

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

Eine Formel heißt erfüllbar, wenn es eine Belegung der Variablen mit 0 und 1 gibt, so dass die Formel den Wert 1 bekommt. Die oben angegebene Formel φ ist zum Beispiel erfüllbar, da $x = 0, y = 1, z = 0$ eine Belegung ist, die φ wahr macht. Das Erfüllbarkeitsproblem SAT ist, festzustellen, ob eine gegebene Formel erfüllbar ist oder nicht.

Satz 51. *(Cook-Levin) $SAT \in P$ genau dann, wenn $P = NP$.*

Bevor wir den Beweis des Theorems beginnen, beschäftigen wir uns mit polynomieller Reduzierbarkeit, die uns als Beweismethode dienen soll.

9.1 Polynomialzeit – Reduzierbarkeit

In Kapitel 4 haben wir den Begriff der funktionalen Reduzierbarkeit eingeführt. Ein Problem A lässt sich auf ein Problem B reduzieren, wenn eine Lösung von B verwendet werden kann, um A zu lösen. Wenn wir aus der effizienten Lösbarkeit von B auf die effiziente Lösbarkeit von A schließen wollen, müssen wir außerdem den Zeitaufwand, der für die „Übersetzung“, der Eingaben von A in die Eingaben von B gebraucht wird, berücksichtigen.

[14] Eine korrekte Definition für Boolesche Formeln findet man z.B. in [1]

Definition 32. *Für ein fest gewähltes Alphabet Σ seien A und B Mengen von Wörtern über Σ. Wir bezeichnen mit FP die Menge der polynomialzeit-berechenbaren totalen Funktionen $f : \Sigma^* \to \Sigma^*$. Die Menge A heißt in Polynomialzeit funktional reduzierbar (bzw. polynomiell reduzierbar) auf B, geschrieben als $A \leq_P B$, wenn es eine Funktion $f \in FP$ gibt, so dass für alle $w \in \Sigma^*$ gilt $w \in A$ genau dann, wenn $f(w) \in B$. Die Funkton f heißt polynomielle Reduktion von A auf B.*

Bisher haben wir die funktionale Reduzierbarkeit von A auf B verwendet, um die Zugehörigkeit $w \in A$ auf die Zugehörigkeit $f(w) \in B$ zurückzuführen. Dass f effizient berechenbar (=Polynomialzeit-berechenbar) sein soll, sorgt dafür, dass die Lösung von B auf effiziente Weise in eine Lösung von A verwandelt werden kann.

Satz 52. *Sei $A \leq_P B$ und $B \in P$. Dann gilt $A \in P$.*

Beweis:
Sei M eine Turingmaschine, die in polynomieller Zeit die Zugehörigkeit zu B entscheidet, und sei f eine polynomielle Reduktion von A auf B. Ein polynomieller Entscheider N für A sieht so aus: $N :=$

„Auf Eingabe w:

1. Berechne $f(w)$.
2. Starte M auf $f(w)$, gebe aus, was M ausgibt.

“

N läuft in polynomieller Zeit, da jeder der beiden Schritte polynomiell ist. □

Als Beispiel für eine polynomielle Reduktion betrachten wir die Reduktion von $3SAT$ auf $CLIQUE$. Dabei ist $3SAT$ eine spezielle Variante von SAT. Die Booleschen Formeln, deren Erfüllbarkeit festgestellt werden soll, haben eine bestimmte Form. Ein Literal ist eine negierte oder nichtnegierte Boolesche Variable, wie x oder $\overline{x}$. Eine Klausel ist die Oder-Verknüpfung von Literalen wie $x_1 \vee x_2 \vee x_3$. Eine Boolesche Formel ist in konjunktiver Normalform, abgekürzt CNF, wenn sie aus der konjunktiven Verknüpfung von Klauseln besteht. Falls jede Klausel genau drei Literale enthält, sagen wir die Formel ist in 3-CNF.

$$3SAT = \{\varphi \mid \varphi \text{ ist eine erfüllbare Boolesche Formel in 3-CNF}\}$$

Satz 53. *$3SAT$ ist polynomiell reduzierbar auf $CLIQUE$.*

Beweisidee: Die Reduktionsfunktion ordnet den Formeln geeignete Graphen zu, und zwar so, dass in den so konstruierten Graphen Cliquen (mit fester bestimmter Größe) erfüllenden Belegungen entsprechen. Strukturen innerhalb der Graphen korrespondieren mit Abhängigkeiten von Variablen und Klauseln.

Beweis: Sei φ eine Formel mit k Klauseln wie:

$$\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

wobei die a_i, b_j, c_l Literale sind, die nicht alle voneineinander verschieden sein müssen.

Die Reduktion f erzeugt einen String $< G, k >$, wobei G ein ungerichteter Graph ist, der wie folgt definiert wird: Für jede Klausel entsteht eine Dreiergruppe von Knoten, die mit den Namen der Literale beschriftet werden. Bis auf zwei Typen von Paaren verbinden die Kanten alle Knoten miteinander. Es gibt keine Kante zwischen den Knoten einer Klausel. Es gibt keine Kante zwischen einem Literal und seiner Negation. Warum existieren Cliquen genau dann, wenn es erfüllende Belegungen für die Formeln gibt? In jedem Tripel wird höchstens ein Element zur Clique gehören, nämlich eines, das mit wahr belegt ist. Hat man aus jedem Tripel ein Element, dann sind alle Klauseln erfüllt und wegen des Kanten„verbotes“ $x - \overline{x}$ gibt es keine Widersprüche.

Erfüllende Belegung: In jeder Klausel wird ein Literal mit 1 belegt; k-Clique vorhanden: Belegung ablesen. □

9.2 Definition von NP-Vollständigkeit

Definition 33. *Eine Sprache B heißt NP-vollständig, wenn sie zwei Bedingungen erfüllt:*

(i) B ist in NP

(ii) Jedes Problem A in NP ist polynomiell reduzierbar auf B.

Satz 54. *Falls B NP-vollständig ist und $B \in P$, dann gilt $P = NP$.*

Beweis: unmittelbar aus der Definition.

Satz 55. *Falls B NP-vollständig ist und $B \leq_P C$ für ein $C \in NP$, dann ist C auch NP-vollständig.*

Beweis: Wir wissen bereits, dass C in NP liegt, es reicht also zu zeigen, dass jedes A aus NP in polynomieller Zeit auf C reduzierbar ist. Wegen $B \leq_P C$ existiert ein $f \in FP$ mit $x \in B$ genau dann, wenn $f(x) \in C$. B ist NP-vollständig, d.h. für alle $A \in NP$ existiert eine $g_A \in FP$ mit $g_A(a) \in B$ gdw. $a \in A$. Komposition von polynomiellen Funktionen ist wieder polynomiell. Damit haben wir für jedes $A \in NP$ eine total berechenbare polynomialzeit-beschränkte Funktion $f \circ g_A$ mit $f \circ g_A(a) \in C$ gdw. $g_A(a) \in B$ gdw. $a \in A$. Das heißt $A \leq_P C$. □

9.3 Der Satz von Levin-Cook

Wenn man ein NP-vollständiges Problem gefunden hat, ist es einfach, mit Hilfe von polynomieller Reduktion andere daraus abzuleiten. Das erste jedoch ist schwieriger. Wir beweisen:

Satz 56 (LEVIN-COOK)**.** *SAT ist NP-vollständig*

Beweisidee:

1. Wir zeigen, dass SAT in NP liegt.
2. Wir zeigen, dass jede Sprache in NP in polynomieller Zeit auf SAT reduzierbar ist.

Während der erste Teil des Beweises einfach ist, ist der zweite Teil die eigentliche Herausforderung. Es bedeutet, dass wir für jede Sprache A in NP eine polynomielle Reduktion f_A angeben müssen, so dass jedem String w eine Boolesche Formel $\varphi = f(w)$ zugeordnet wird, die die Arbeit der nichtdeterministischen Polynomialzeitturingmaschine M_A auf w simuliert. Falls M_A die Eingabe w akzeptiert, soll $\varphi = f(w)$ eine erfüllende Belegung haben und umgekehrt. Die tatsächliche Konstruktion der Reduktion ist konzeptuell nicht weiter schwierig, allerdings müssen viele Details bedacht werden.

Eine Boolesche Formel kann die Booleschen Operationen $\wedge, \vee$ und $\neg$ enthalten. Wenn man sich überlegt, dass diese Operationen die Grundlagen für die Schaltkreise bilden, aus denen Computer aufgebaut sind, dann ist es nicht weiter überraschend, dass eine Boolesche Formel eine Turingmaschine simulieren kann.

Beweis:

1. SAT liegt in NP: Eine nichtdeterministische Turingmaschine kann in polynomieller Zeit eine Belegung für φ raten und akzeptieren, falls diese erfüllend ist.
2. Sei $A \in NP$ und N_A eine polynomielle nichtdeterministische Turingmaschine für A mit Laufzeit $O(n^k)$ für eine Konstante k. Der folgende Begriff soll helfen, die Reduktion zu beschreiben: Ein Tableau für N_A auf w ist eine $n^k \times n^k$-Tabelle, deren Zeilen die Konfigurationen eines Berechnungszweiges von N_A auf w sind. Wegen Bequemlichkeit im späteren Teil wollen wir annehmen, dass jede Konfiguration mit einem Trennzeichen $\sharp$ beginnt und endet. Damit besteht die erste und die letze Spalte eines Tableaus nur aus Trennzeichen. Die erste Zeile enthält die Startkonfiguration und jede folgende Zeile eine Nachfolgekonfiguration der darüber liegenden Zeile entsprechend der Überführungsfunktion

von N_A. Ein Tableau ist akzeptierend, wenn eine Zeile eine akzeptierende Konfiguration ist. Damit ist die Frage, ob N_A das Wort w akzeptiert, die Frage, ob sie ein akzeptierendes Tableau hat.

Wir wollen nun die Reduktionsfunktion f beschreiben, die aus der Eingabe w eine Formel φ konstruiert. Zunächst beschreiben wir die Variablen von φ: Seien Q und Γ die Menge der Zustände und das Bandalphabet von N_A. Sei $C = Q \cup \Gamma \cup \{\sharp\}$. Für jedes $i, j \in \{1 \dots n^k\}$ und $c \in C$ definieren wir eine Variable . Wir bezeichnen mit $cell(i, j)$ den Eintrag in Spalte j, in Zeile i und die Variable $x_{i,j,c}$ bekommt den Wert 1, wenn $cell(i, j) = c$. Nun müssen wir die Formel φ so festlegen, dass sie einem akzeptierenden Tableau entspricht.

$$\varphi := \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{move} \wedge \varphi_{accept}$$

Die Formel besteht also aus einer Konjunktion von vier Teilformeln, die wir im folgenden beschreiben:

Um eine eindeutige Belegung von φ zu bekommen, müssen wir sicherstellen, dass jede Belegung jeder Zelle genau einen Eintrag gibt:

$$\varphi_{cell} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{c \in C} x_{i,j,c} \right) \wedge \left(\bigwedge_{c,c' \in C, c \neq c'} (\overline{x_{i,j,c}} \vee \overline{x_{i,j,c'}}) \right) \right]$$

Die Symbole $\bigwedge$ und $\bigvee$ stehen für iterierte $\wedge$ und $\vee$. Der erste Teil der Formel besagt, dass in jeder Stelle des Tableaus mindestens ein Wert c stehen muss, der zweite Teil besagt, dass es niemals zwei verschiedene Einträge c und c' in einer Zelle geben kann, bzw. nur für genau ein c ist bei festem i und j eine Variable $x_{i,j,c}$ mit 1 belegt.

Die Teilformeln $\varphi_{start}, \varphi_{move}$ und φ_{accept} sichern, dass die Tabelle tatsächlich ein akzeptierendes Tableau ist. Die erste Zeile ist eine Startkonfiguration:

$$\begin{aligned} \varphi_{start} = & x_{1,1,\sharp} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \dots \wedge x_{1,n+2,w_n} \wedge \\ & \wedge x_{1,n+3,\sqcup} \dots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\sharp} \end{aligned}$$

In einer Zeile steht eine akzeptierende Konfiguration:

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

Mit der Formel φ_{move} drücken wir aus, dass zwei übereinanderstehende Zeilen wirklich aufeinanderfolgende Konfigurationen darstellen. Dazu betrachten wir kleine Ausschnitte der Tabelle, die jeweils 3 Zellen

breit und 2 Zellen hoch sind. Wir sagen von einem solchen Fenster, dass es legal[15] ist, wenn es durch Übereinanderschreiben eines Paares aus Konfiguration und deren Nachfolgekonfiguration entsprechend der Überführungsfunktion von N_A entstehen kann.

Wir betrachten ein Beispiel: Seien $a, b, c \in \Gamma$ und $q_1, q_2 \in Q$ und es gelte $\delta_{N_A}(q_1, a) = \{(q_1, b, R)\}$ und $\delta_{N_A}(q_1, b) = \{(q_2, c, L), (q_2, a, R)\}$. Es ergeben sich z.B. folgende gültige Fenster:

a	q_1	b
q_2	a	c

a	q_1	b
a	a	q_2

a	a	q_1
a	a	b

♯	a	b
♯	a	b

a	b	a
a	b	q_2

b	b	b
c	b	b

Die folgenden Fenster sind nicht legal für unsere Maschine N_A:

a	b	a
a	a	a

a	q_1	b
q_1	a	a

b	q_1	b
q_2	b	q_2

Fakt: *Wenn die oberste Zeile der Tabelle eine Startkonfiguration ist und jedes 2×3-Fenster legal ist, dann ist jede Zeile der Tabelle eine Nachfolgekonfiguration der darüberliegenden.*

Wir beweisen diese Behauptung indem wir zwei benachbarte Konfigurationen betrachten, die obere und untere Konfiguration heißen sollen. In der oberen Konfiguration ist jede Zelle, die nicht unmittelbar neben einem Zustandssymbol steht und die kein ♯ enthält, in der Mitte eines 2×3-Fensters, das keine Zustandsymbole enthält. Daher muss dieses Symbol auch unverändert in der Mitte der unteren Zeile stehen.

Ein Fenster, das ein Zustandssymbol in der Mitte der oberen Zeile enthält, garantiert, dass die entsprechenden drei Positionen in Übereinstimmung mit der Übergangsfunktion erneuert werden. Daher ist, wenn die obere Konfiguration legal ist, auch die untere legal und die untere ergibt sich durch die Übergangsfunktion als Nachfolgekonfiguration aus der oberen.

Wir machen nun mit der Konstruktion von φ_{move} weiter. Die Formel soll so eingerichtet werden, dass sie ausdrückt, dass alle 2×3-Fenster legal sind.

$$\varphi_{move} = \bigwedge_{1<i\leq n^k, 1<j<n^k} (\text{das } (i,j)\text{-Fenster ist legal})$$

Der Text „das (i, j)-Fenster ist legal“ steht dabei für folgende Formel:

$$\bigvee_{a_1,\ldots,a_6 \text{ist legal}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

[15] Eine korrekte Definition davon, wann ein Fenster legal ist, ist sehr mühsam, daher wird sie hier weggelassen. Wer die Details ausarbeiten will, sehe sich noch einmal den Beweis des PKP an, in dem wir ganz ähnlich vorgegangen waren.

Abschließend analysieren wir die Komplexität der Reduktion, um zu zeigen, dass sie in Polynomialzeit berechenbar ist. Als erstes betrachten wir die Größe von φ. Da jedes Tableau eine $n^k \times n^k$-Tabelle ist, enthält sie n^{2k} Zellen. Jede Zelle hat $|C|$[16] Variablen. Da $|C|$ nur von N abhängt, aber nicht von der Eingabelänge, liegt die Zahl der Variablen in $O(n^{2k})$. Da φ_{cell} für jede Zelle nur eine feste beschränkte Teilformel enthält, ist ihre Größe in $O(n^{2k})$. Da φ_{start} für jede Zeile in der obersten Reihe eine Teilfomel enthält ist ihre Größe $O(n^k)$. Die Formeln φ_{move} und φ_{accept} enthalten für jede Zelle eine fest beschränkte Teilformel, also sind sie ebenfalls in $O(n^{2k})$. Damit ist die Gesamtgröße von φ in $O(n^{2k})$, d.h. ihre Größe wird beschränkt durch ein Polynom in n.

Um zu sehen, dass die Formel auch in polynomieller Zeit erzeugt werden kann, beachte man ihren iterativen Charakter. Jeder Teil der Formel besteht aus fast identischen Teilen, die nur in den Indizes voneinander abweichen. Man kann daher auch eine Reduktion angeben, die aus einer Eingabe w eine Formel φ produziert. □

Mit dem Beweis für den Satz von Cook-Levin haben wir für ein NP-Problem direkt nachgewiesen, dass es NP-vollständig ist. Für ein beliebiges NP-vollständiges Problem ist es nicht notwendig, den Beweis direkt zu führen. Statt dessen reicht es, ein bekanntermaßen NP-vollständiges Problem auf das zu untersuchende Problem polynomiell zu reduzieren. Für diesen Zweck kann man SAT direkt benutzen. Häufig ist aber eine Reduktion von $3SAT$ einfacher. Dazu müssen wir natürlich zuerst beweisen, dass $3SAT$ tatsächlich NP-vollständig ist. Eine sehr umfangreiche Quelle für Reduktionen von NP-vollständigen Probleme ist [2]. Dort werden viele Probleme aufgelistet und in Kürze die Beweisidee für die Reduktion angegeben. Außerdem findet man meist eine Quellenangabe für einen ausführlichen Beweis.

Folgerung: $3SAT$ ist NP-vollständig.

Beweis: Offensichtlich ist $3SAT$ in NP. Für eine gegebene Belegung(als Zertifikat) können wir in polynomieller Zeit überprüfen, ob eine gegebene Formel in $3CNF$ von dieser Belegung erfüllt wird oder nicht.

Wir müssen also nur noch beweisen, dass sich alle NP-Sprachen in polynomieller Zeit auf $3SAT$ reduzieren lassen. Ein Weg dazu wäre zu zeigen, dass sich SAT auf $3SAT$ polynomiell reduzieren lässt. Stattdessen modifizieren wir den Beweis von Satz 56, sodass direkt eine Formel in $3CNF$ entsteht. φ_{cell} ist eine Konjunktion von Teilformeln, von denen jede eine Disjunktion und eine Konjunktion von Disjunktionen enthält. Das heißt, sie ist bereits in konjunktiver Normalform. Fasst man jede der Variablen als Klausel der Länge 1 auf, dann ist φ_{start} eine Konjunktion von Klauseln. Die Formel φ_{accept} ist eine

[16] Größe der Vereinigung von Bandalphabet und Menge der Zustände von N

große Disjunktion, also eine einzelne Klausel. Die Formel φ_{move} ist die einzige, die nicht schon in konjunktiver Normalform ist. Sie ist eine Konjunktion von Teilformeln, die selbst wieder Disjunktionen von Konjunktionen sind. Nach den Distributivgesetzen kann man jede Disjunktion von Konjunktionen in eine Konjunktion von Disjunktionen transformieren. Das erhöht signifikant die Größe jeder Teilformel, kann aber die absolute Größe von φ_{move} nur um einen konstanten Faktor verändern, da die Größe jeder Teilformel nur von N_A abhängt. Als Ergebnis erhält man eine Formel in konjunktiver Normalform. Hat man eine Formel in konjunktiver Normalform, muss man sie noch umwandeln, so dass sie nur noch drei Literale pro Klausel hat. In jeder Klausel die bis jetzt nur ein oder zwei Literale hatte , vervielfältigen wir ein oder zwei Literale. In jeder Klausel mit mehr als drei Literalen teilen wir die Klausel in mehrere Klauseln auf und führen zusätzliche Variablen ein, um die Erfüllbarkeit bzw. Nicht-Erfüllbarkeit zu erhalten. Zum Beispiel würde man die Klausel $a_1 \vee a_2 \vee a_3 \vee a_4$, in der die a_i Literale sind, durch den Ausdruck $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$ ersetzen, der unabhängig davon, wie z belegt wird, genau dann falsch ist, wenn $a_1 = a_2 = a_3 = a_4 = 0$. Allgemein gilt, dass eine Klausel mit l Literalen durch eine Konjunktion von $l - 2$ Klauseln mit drei Literalen ersetzt werden kann. Damit haben wir gezeigt, dass die neue Formel in $3CNF$ genau dann erfüllbar ist, wenn die ursprüngliche Formel in SAT lag.

□

9.4 Weitere NP-vollständige Probleme

In diesem Abschnitt präsentieren wir eine Reihe von Aussagen über NP-vollständige Probleme. Die allgemeine Strategie wird sein, eine polynomielle Reduktion von $3SAT$ auf das betreffende Problem anzugeben. Um das zu tun, halten wir nach Strukturen in den entsprechenden Problemen Ausschau, die die Variablen und Klauseln in den Formeln darstellen können. So simuliert ein einzelner Knoten in der Reduktion, die wir in Satz 53 angegeben haben, eine Variable und jede Dreiergruppe von Knoten eine Klausel. Ein einzelner Knoten kann entweder in der Clique sein oder nicht – eine Variable kann mit wahr oder falsch belegt werden. Jede Klausel muss ein wahres Literal enthalten, jede Dreiergruppe muss einen Knoten aus der Clique enthalten. Die Folgerung stellt fest, dass $CLIQUE$ NP-vollständig ist.

Folgerung: $CLIQUE$ ist NP-vollständig.

9.4.1 Kantenüberdeckung–Vertex-Cover

Sei G ein ungerichteter Graph, dann ist eine Kantenüberdeckung von G eine Teilmenge von Knoten, wobei jede Kante von G einen dieser Knoten als Endpunkt hat. Das Kantenüberdeckungsproblem fragt nach der Größe der kleinsten Kantenüberdeckung[17].

$$VERTEX-COVER = \{< G, k > |\ G \text{ ist ein ungerichteter Graph mit einer Kantenüberdeckung von } k \text{ Knoten}\}$$

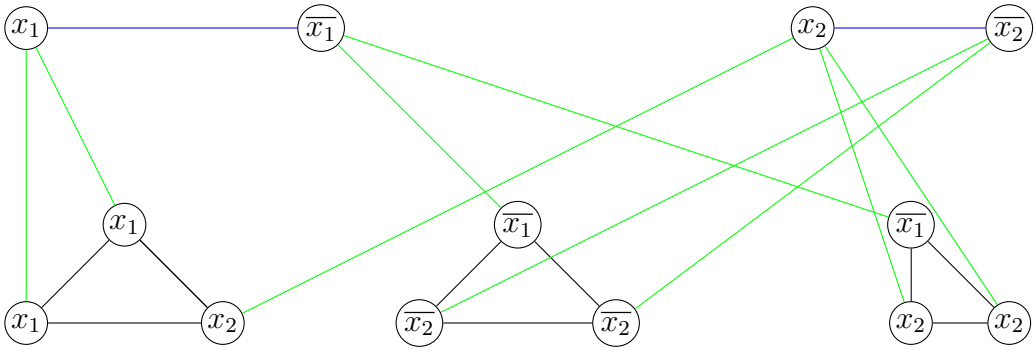
Satz 57. *$VERTEX-COVER$ ist NP-vollständig.*

Beweis: Wir geben eine polynomielle Reduktion von $3SAT$ auf $VERTEX-COVER$ an. Die Reduktion bildet eine Boolesche Formel φ auf einen Graphen G und einen Wert k ab. Jede Kante von G muss wenigstens einen Endpunkt in der Kantenüberdeckung haben. Damit ist die natürliche Struktur, dass eine Variable einer Kante entspricht. Das Setzen einer Variable auf *true* entspricht der Auswahl des linken Knotens für die Kantenüberdeckung, wohingegen $false$ der Auswahl des rechten Knotens entspricht. Wir benennen die beiden Knoten für die Variable x mit x und $\overline{x}$. Bei den Klauseln ist die Struktur etwas komplexer. Jede Klausel ist ein Tripel von drei Literalen. Die entsprechenden drei Knoten sind paarweise miteinander und mit den Variablenkanten so verbunden, das jeweils gleiche Namen eine Kante bekommen. Damit ist die Gesamtzahl der Knoten, die in G vorkommen $2m + 3l$, wenn φ m Variablen und l Klauseln hat. Wir definieren $k := m + 2l$.

Wenn z.B. $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$, dann produziert die Reduktion aus φ das Paar $< G, k >$ mit $k = 2 + 2 \cdot 3 = 8$ und G hat die folgende Form wie in Abbildung 18 Um zu zeigen, dass die Reduktion funktioniert, müssen wir zeigen, dass φ genau dann erfüllbar ist, wenn G eine Kantenüberdeckung mit k Knoten hat. Wir beginnen mit einer erfüllenden Belegung. Sei M die Menge der Knoten der Variablenkanten, die bei der Belegung wahr wird. In der Knotenmenge, die den Literalen entspricht, markieren wir je ein wahres Literal pro Klausel und fassen die anderen in einer Menge M' zusammen. $M \cup M'$ hat die Größe k. Alle Kanten werden davon überdeckt, da offensichtlich die Variablenkanten erreicht werden, in jeder Klausel die Verbindungskanten erreicht werden und die Kanten von den Variablen-Knoten zu den Literal-Knoten abgedeckt sind. Für die andere Richtung zeigen wir, dass, falls G eine Kantenüberdeckung mit k Knoten hat,

[17] Man kann sich die Aufgabe so vorstellen, dass der Graph einen Grundriss eines Museums darstellt, wobei die Kanten Gänge und die Knoten Kreuzungen der Gänge sind. Das Vertex-Cover Problem besteht darin, die minimale Anzahl an Aufsichtspersonen zu finden, die jeweils an einer Ecke stehend alle Gänge im Blick haben

Abbildung 18: Graph zur Formel $\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$



man eine erfüllende Belegung finden kann. Eine Kantenüberdeckung muss in jeder Variablenkante einen Knoten haben, und zwei Knoten aus jedem Tripel. □

9.4.2 Das Hamilton-Pfad-Problem

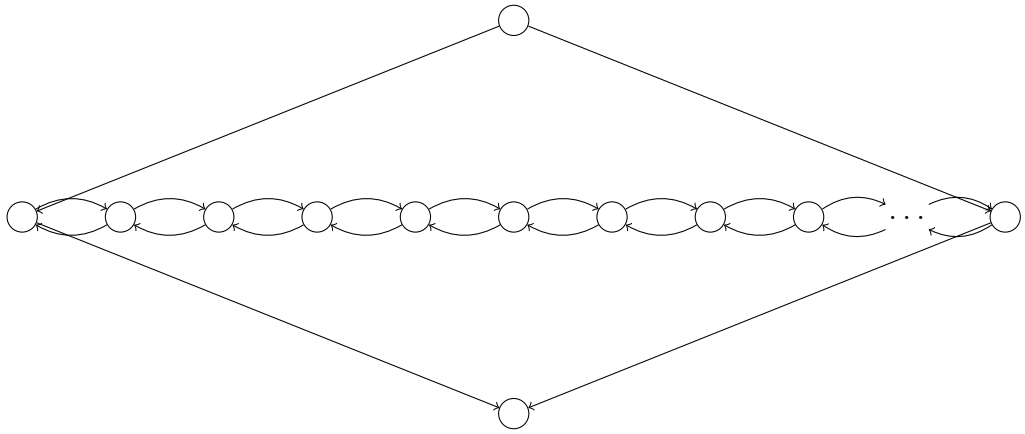
Satz 58. *$HAMPATH$ ist NP-vollständig.*

Beweisidee Um zu zeigen, dass $HAMPATH$ NP-vollständig ist, müssen wir wieder zwei Dinge beweisen: $HAMPATH \in NP$ und jede NP-Sprache A hat eine polynomielle Reduktion auf $HAMPATH$. Den ersten Teil haben wir schon in Abschnitt 8.2 gezeigt. Um den zweiten Teil zu beweisen, reduzieren wir ein bekanntermaßen NP-vollständiges Problem, nämlich $3SAT$, polynomiell auf $HAMPATH$. Dazu müssen wir angeben, wie wir Formeln in 3CNF so auf Graphen abbilden, dass genau dann, wenn die Formel erfüllbar ist, ein Hamilton-Pfad im Graphen existiert. Der Graph muss Strukturen haben, die das Verhalten von Variablen und Klauseln nachahmen. Die Variablenstruktur (im Graphen) ist ein Rhombus, der – entsprechend den beiden möglichen Belegungen – auf zwei verschiedene Arten durchlaufen werden kann. Die Klauselstruktur (im Graphen) ist ein Knoten. Der Pfad soll durch jeden Klauselknoten führen, wenn alle Klauseln wahr sind.

Beweis: Wir konstruieren eine Reduktion für $3SAT \leq_p HAMPATH$. Für jede Formel φ wird ein Graph konstruiert. Sei k die Anzahl der Klauseln, dann lässt sich eine Formel in $3CNF$ schreiben als:

$$\varphi = \bigwedge_{i=1}^{k} (a_i \vee b_i \vee c_i),$$

Abbildung 19: Teilgraph für x_i



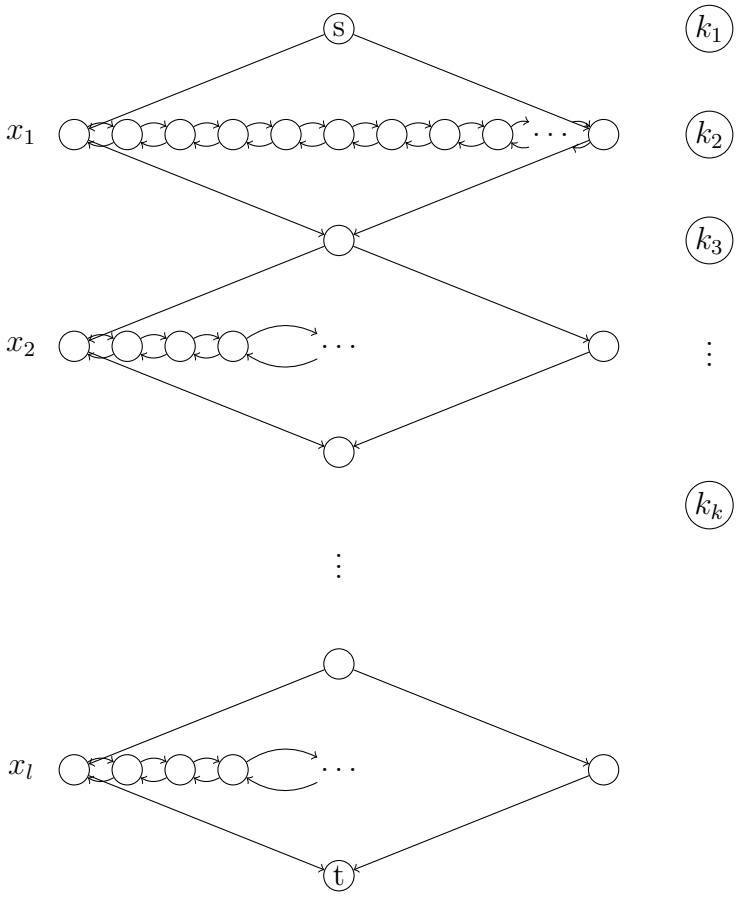
wobei die a_i, b_i, c_i Literale, also Variablen x_i oder deren Negation $\overline{x_i}$ sind. Für jede Variable konstruieren wir einen Rhombus siehe Abbildung19 mit einer Folge von $3k + 1$ Knoten in der Mitte, also einen Teilgraphen mit $3k + 5$ Knoten.

Außer den l Rhomben für die l Variablen konstruieren wir für jede Klausel k_j einen Knoten k_j erzeugt (ergibt weitere k Knoten). Der ganze Graph wird zusammengesetzt durch „Verkleben“ der oberen Knoten eines Variablenteilgraphen mit dem unteren Knoten eines anderen Variablenteilgraphen, siehe Abbildung 20.

Der oberste Knoten im x_1 Rhombus bekommt den Namen s, der unterste im x_l-Rhombus den Namen t. Bei den inneren $3k + 1$ Knoten in der Mitte von jedem Rhombus werden zwei aufeinanderfolgende Knoten zu Zweiergruppen zusammengefasst, für jede Klausel eine, und dazwischen wird jeweils ein Trennknoten belassen. Es entsteht also eine Folge der Form: Trennknoten-Zweiergruppe für k_1-Trennknoten-Zweiergruppe für k_2 ... Zweiergruppe für k_k-Trennknoten. Falls die Variable x_i in der Klausel k_j vorkommt, fügen wir zwei Kanten zum Graphen hinzu: eine vom linken Knoten des zu k_j gehörenden Paars zum Knoten k_j und eine von k_j zum rechten Knoten des zu k_j gehörenden Paares im Rhombus der zu x_i gehört. Wenn $\overline{x_i}$ in der Klausel k_j vorkommt, gibt es eine Kante vom rechten Knoten des Paars zu k_j und eine von k_j zum linken Knoten des Paares(im Rhombus x_i). Damit ist die Konstruktion von G beendet.

Es bleibt zu zeigen, dass die Konstruktion korrekt ist und in P gemacht werden kann. Wir nehmen an, dass φ erfüllbar ist und müssen zeigen dass dann ein Hamilton-Pfad von s nach t existiert. Unser Pfad beginnt in s, falls

Abbildung 20: Graph für Formel mit l Variablen und k Klauseln



x_1 in der erfüllenden Belegung *wahr* ist, geht der Pfad nach links unten in den Rhombus von x_1, entlang der Mittelknoten und auf der anderen Seite nach links unten zum ersten Verklebungspunkt zwischen x_1 und x_2. War x_i *falsch* in der erfüllenden Belegung, geht man von s nach rechts unten und von rechts nach links durch den Rhombus. Am Ende führt der Pfad zum „Verklebungspunkt“. Im zweiten Rhombus verfährt man ebenso, wenn x_2 wahr ist, von links nach rechts durch die Mittelknoten andernfalls von rechts nach links. Das Verfahren wird fortgesetzt, bis man bei t angekommen ist. Nun müssen noch die Klauselknoten einbezogen werden. Für jede Klausel k_j wählen wir ein wahres Literal a_j, b_j oder c_j aus. Ist das Literal eine Variable x_i, dann machen wir beim Durchgehen durch den zugehörigen Rhombus einen „Umweg“ über k_j. Da wir von links kommen, ist das möglich durch die Kanten auf dem zu k_j gehörenden Paar von und nach k_j im Rhombus x_i. Ist das Literal eine negierte Variable $\overline{x_i}$, dann würden wir von rechts durch den entsprechenden Rhombus gehen und haben die passenden Kanten um einen „Umweg“ über k_j zu gehen. Damit haben wir einen Hamilton-Pfad konstruiert.

Nehmen wir an, es gäbe eine Hamilton-Pfad durch den Graphen. Wenn er *normal* ist, d.h. wenn er von oben nach unten und quer durch die Rhomben geht unter Einbeziehung der Klauselknoten, dann können wir eine Belegung einfach ablesen. Die Variablen, die zu den Rhomben, die von links nach rechts durchlaufen werden, werden mit *wahr* belegt, die anderen mit $falsch$.

Was zu zeigen bleibt, ist, dass andere als normale Hamiltonpfade nicht existieren können. Die einzige Möglichkeit, dass der Pfad nicht normal ist, besteht darin, dass ein Klauselknoten von einem anderen Rhombus aus erreicht wird, als von dem zu dem der ausgehende Pfad hingeht. Nehmen wir an, wir hätten eine solche Situation. Wir nennen den Knoten im Rhombus x_i, von dem aus die Klausel k_j erreicht wird, a_1 und dessen rechte Nachbarn a_2 und a_3. Der Knoten von k_j in den Rhombus x_n heiße b_2 und sein linker Nachbar b_1. Dann müsste entweder a_2 oder a_3 ein Trennknoten sein. Wenn a_2 der Trennknoten wäre, dann wären a_1 und a_3 die einzigen Knoten, von denen aus a_2 erreichbar wäre. Von a_1 geht der Pfad aber zu k_j und a_3 wird als Ausgang gebraucht. Wenn a_3 der Trennknoten wäre, dann wären a_2 und a_1 im selben Klauselpaar und die einzigen Kanten nach a_2 wären die von a_1 und a_3 und k_j. In jedem Fall würde der Pfad dann a_3 nicht enthalten. Damit haben wir uns davon überzeugt, dass jeder Hamilton-Pfad normal sein muss.

Die Reduktion lässt sich in Polynomialzeit ausführen, da nur polynomiell viele Knoten und Kanten erzeugt werden müssen.

□

Literatur

[1] R. Cori and D. Lascar. *Mathematical Logic*. Oxford University Press, 2001.

[2] M.R. Garey and D.S. Johnson. *Computers and Intractability*. A series of books in the mathematical sciences. W.H. Freeman and Company, New York, 2003.

[3] J.E. Hopcroft, R. Motwani, and J.D Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Verlag, 2002.

[4] Jörg Rothe. *Komplexitätstheorie und Kryptologie*. Springer-Verlag, 2008.

[5] U. Schöning. *Theoretische Informatik-kurz gefasst*. Spektrum, 2001.

[6] M. Sipser. *Introduction of the Theory of Computation*. PWS Publishing Company, 1997.

[7] I. Wegener. *Theoretische Informatik*. B.G.Teubner Stuttgart, 1993.

[8] I. Wegener. *Complexity Theory*. Springer, 2005.