F3 – Berechenbarkeit und Komplexität

Prüfungsunterlagen zum Vorlesungszyklus:
„Formale Grundlagen der Informatik“
gehalten von Prof. Dr. Matthias Jantzen
Wintersemester 2000/2001

Prof. Dr. Matthias Jantzen©

erstellt: Sommersemester 2001
korrigierte Version gedruckt: 22. Oktober 2001

Inhaltsverzeichnis

0 Vorwort **1**

1 Probleme und Algorithmen **3**

1.1 Einführende Beispiele 3

1.2 Historische Bemerkungen 7

1.3 Probleme und Algorithmen 8

1.4 Problemtypen 9

1.5 Eigenschaften von Algorithmen 11

1.6 Beschreibungsformen von Algorithmen 14

1.7 Die O-Notation 17

1.8 Effizienz von Algorithmen 22

2 Turingmaschinen: Entscheidbarkeit und Berechenbarkeit **26**

2.1 Deterministische Turing-Maschinen 27

2.2 Beispiele für Turingmaschinen 30

2.3 Konfigurationen der Turing-Maschine 35

2.4 Berechenbarkeit 37

2.5 Varianten der deterministischen Turing-Maschine 38

2.6 Nichtdeterministische Turing-Maschinen 42

2.7 Entscheidbarkeit und Aufzählbarkeit 44

2.8 Universelle Turing-Maschine 49

2.9 Das Halteproblem 51

2.10 Weitere Beispiele von Algorithmen 54

2.11 Reduktion des Halteproblems 55

2.12 Das Kachelproblem 57
2.13 Weitere unentscheidbare Probleme 59
2.14 Weitere universelle Automatenmodelle 61

3 Andere Präzisierungen der Berechenbarkeit 65

3.1 Primitiv-rekursive Funktionen 65
3.2 μ-rekursive Funktionen 69

4 Registermaschinen 72

4.1 Das RAM-Modell 72
4.2 Komplexitätsmaße für RAMs 79
4.3 Simulation von bzw. mit Turing-Maschine 82

5 Chomsky Typ-0 und Chomsky Typ-1 Grammatiken 84

5.1 Sprachfamilien und Abschlusseigenschaften 84
5.2 Typ-0 oder Phrasenstruktur-Grammatik 85
5.3 Typ-1 oder Kontextsensitive Grammatik 89
5.4 Kontextsensitivität und Entscheidbarkeit 92
5.5 Linear beschränkte Automaten 93

6 Algorithmen-Techniken 98

6.1 Rekursiv oder iterativ? 98
6.2 Divide and Conquer 100
6.3 Greedy-Algorithmen 105
6.4 Dynamische Programmierung 112
6.5 Backtracking 122
6.6 Branch-and-Bound Verfahren 123
6.7 Heuristiken und Approximationsverfahren 127

7 Analysemethoden **132**

7.1 Rekurrenzgleichungen 134

7.2 Vollständige Induktion 137

7.3 Reduzierung der Rekurrenzgleichungen auf Summen 138

7.4 Reduktion auf algebraische Gleichungen 141

7.5 Integralabschätzung 145

7.6 Schnelle Multiplikation 148

8 Strukturelle Komplexitätstheorie **151**

8.1 Darstellung von Problemen für TM-Eingabe 151

8.2 Zeit- und Platzbedarf 153

8.3 Bandkompression und Beschleunigung 156

8.4 Komplexitätsklassen 158

8.5 NP-Vollständige Probleme 160

Literatur **175**

0 Vorwort

Diese Vorlesung ist Teil 3 der Grundstudiumsveranstaltung **"Formale Grundlagen der Informatik"** und baut auf die vorangegangen Vorlesungen

F1: Logik und **F2**: Automaten und Formale Sprachen

auf. Der Besuch von **F1** (Logik) ist für das Verständnis von **F3** nicht unbedingt erforderlich, während jedoch auf die Inhalte von **F2** nicht vollständig verzichtet werden kann. In den unten angesprochenen Büchern sind diese aber im Wesentlichen auch enthalten

Dieser Text ist eine Überarbeitung des Skriptes zur gleichen Veranstaltung vom Wintersemester 2000/2001.

Der vorliegende Text ist als Arbeitsmaterial zu verstehen. Ein gründliches Studium erfordert von den Studierenden das Heranziehen von zusätzlicher Literatur, denn es konnten hier manche Sachverhalte nur angedeutet werden. Die Literaturliste ist relativ umfangreich, was allerdings nicht bedeutet, dass alle Texte zum Verständnis des Stoffes nötig sind: Neben historischen Quellen sind zum gleichen Thema auch alternative Darstellungen in den angegebenen Büchern zu finden. Hier kann je nach Geschmack oder Verfügbarkeit ausgewählt werden. Manche der angegebenen Werke sind auch noch sehr gut im Hauptstudium zu verwenden! Das gilt besonders für Studierende der Informatik im Hauptfach.

Für diejenigen die parallel zu diesem Skript einen weiteren Zugang suchen, vielleicht auch nur zur Vertiefung der Themen, sollen hier einige Kommentare helfen:
In der Neuauflage, [HMU01], von Hopcroft und Ullmans Standardwerk, sind Turing-Maschinen und Komplexität beschrieben, es fehlen aber die in früheren Ausgaben enthaltenen Typ-1 und Typ-0 Grammatiken. Wer dies auch noch braucht, ist mit einem der älteren Hopcroft/Ullman Bücher [HUl79, HUl86] für die theoretischen Grundbegriffe gut genug ausgestattet. Der eher algorithmenorientierte Stoff wird am besten überdeckt durch die Bücher von Brassard und Bratley, [BBr96] oder Sedgewick, [Sed92], zusammen mit einem der umfangreichen Bücher von Savage, [Sav98] oder Gruska, [Gru97], in denen Register- und Turing-Maschinen als formale Modelle von Algorithmen zusammen mit den Definitionen der Komplexitätstheorie enthalten sind. Zu den mathematischen Methoden eignet sich sehr schön das Buch von Graham, Knuth und Patashnik, [GKP89].

Die im **F3** wichtigen Notationen der diskreten Mathematik findet man allerdings auch ausreichend in [Aig93] oder [And01], wobei letzteres viele in der Informatik auftauchende Anwendungen und Themen behandelt. Das Buch von Biggs, [Big99], ist für diesen Vorlesungsstoff allenfalls knapp gehaltene Grundlage, aber letztlich nicht ganz ausreichend. Probleme und Methoden bei denen Matrizen Verwendung finden sind vollständiger – und auch für Anwendungen brauchbar – in [Möl97] dargestellt. Wer nur schnell etwas nachschlagen möchte, sei schließlich auf den Duden Informatik, [Dud93], hingewiesen!

Wie schon in anderen Skripten des Autors werden auch hier neue Begriffe bei deren Definition unterstrichen und die Englischen Bezeichnungen vieler mathematischer Notationen in *italic* notiert. Definitionen und Beweise werden mit einem kleinen Quadrat „ $\square$ “ abgeschlossen, Theoreme (Sätze) und Lemmata (Hilfssätze) dagegen mit einer kleinen Raute „ $\diamond$ “. Korollare, das sind Folgerungen aus Theoremen oder Lemmata (auch: Lemmas) werden dagegen nicht mit einem eigenen ”Endezeichen” versehen. Die in der Literatur häufiger auftretende Bezeichnung ”Proposition” für einen nicht hier, sondern anderswo bewiesenen Sachverhalt werden wir nicht verwenden, obschon nicht für alle Resultate eigene Beweise formuliert werden.

1 Probleme und Algorithmen

1.1 Einführende Beispiele

Der Algorithmenbegriff ist einer der Grundbegriffe der Informatik. Alles, was wir mit einem Computer machen oder machen können, setzt das Vorhandensein von Algorithmen voraus. Auf die Frage: " *Was ist ein "Algorithmus" und wozu dient ein solcher?* " zunächst nur eine knappe und wenig zufriedenstellende Antwort: Algorithmen braucht man zur Lösung von Problemen. Diese Antwort kann man so natürlich nicht stehen lassen, denn sie setzt eine Erklärung der Begriffe "Lösung" und "Problem" voraus.

Fünf Beispiele:

1. **Ägyptische (russische) Multiplikation von Zahlen:**
   Wir alle kennen das Verfahren aus dem Schulunterricht. Wenn die Zahlen groß sind, dann muß die Multiplikation im kleinen Einmaleins flott von der Hand gehen. Wenn man nur Addition, Multiplikation mit 2 und ganzzahlige Division[1] mit 2 beherrscht, dann bietet sich das schon den Ägyptischen Priestern bekannte Verfahren an:

   Die zu multiplizierenden Zahlen werden an den Beginn zweier Spalten in die erste Zeile geschrieben. Deren Reihenfolge ist dabei beliebig. Nun wird die Zahl in der ersten Spalte ganzzahlig halbiert und die der zweiten Spalte verdoppelt. Beide Ergebnisse werden in den gleichen Spalten der nächste Zeile geschrieben. Dieses Verfahren wird solange mit den Zahlen der jeweils neuen Zeile durchgeführt, bis in der ersten Spalte die 1 steht. Da die "schönen" geraden Zahlen nur dem Pharao zustehen, werden alle Zeilen gestrichen, bei denen in der ersten Spalte eine gerade Zahl steht. Anschließend werden alle Zahlen der zweiten Spalte addiert und ergeben als Summe gerade das gesuchte Produkt der beiden Zahlen der ersten Zeile. Zum Verständnis ein Zahlenbeispiel: $230 \cdot 123 = 28290$

[1] Die ganzzahlige Division $\div$ bedeutet, dass stets alle Nachkommastellen weggelassen werden. Also ist $5 \div 2 = 2$ und $7 \div -2 = -3$.

230	123	(zu streichen)
115	246	
57	492	
28	984	(zu streichen)
14	1968	(zu streichen)
7	3936	
3	7872	
1	15744	
	28290	

Diese Methode ist tatsächlich immer korrekt, wie man z.B. mit vollständiger Induktion beweisen kann, (vergl. [Dit96, BBr96]).

2. **größter gemeinsamer Teiler:**
   Zu zwei beliebigen gegebenen natürlichen Zahlen m und n ist der größte gemeinsame Teiler (kurz: ggT, *gcd* **g**reatest **c**ommon **d**ivior) gesucht (d.h. eine natürliche Zahl t, die sowohl m als auch n teilt), so dass für jeden anderen gemeinsamen Teiler d von m und n gilt: $d|t$ (lies: d teilt t). Für den ggT von m und n schreiben wir hier kurz: $< m, n >$ (In der Mathematik ist auch die Notation (m, n) gebräuchlich, was wir hier aber vermeiden wollen, da die „Tupel-Notation" sonst überbeansprucht werden würde!).

3. **Primzahltest:**

   Gegeben: Eine natürliche Zahlen $n \in I\!N$.
   Frage: Ist n eine Primzahl?

4. **Primfaktorzerlegung:**

   Die natürliche Zahl $n \geq 10000$ ist in Primfaktoren zu zerlegen.

   Darin steckt das Problem, alle Primteiler von n zu bestimmen.

5. **Bestimmung von Primzahlen:**

   Man bestimme alle Primzahlen zwischen 1 und 1000000.

Um Aufgaben dieser Art zu lösen, braucht man für jede Fragestellung ein allgemeines Verfahren mit dem die Lösung gefunden wird. Man nennt solch ein allgemeines Verfahren auch **Algorithmus**. Da hiermit der Begriff des Algorithmus nicht sonderlich präzise spezifiziert wird sehen wir in der Literatur nach. Die Definitionen werden von verschiedenen Buchautoren ganz unterschiedlich formuliert:

- *Kleine Enzyklopädie MATHEMATIK* [KEM68]:

  > Ein **Algorithmus** liegt genau dann vor, wenn gegebene Größen, auch **Eingabegrößen, Eingabeinformationen** oder **Aufgaben** genannt, auf Grund eines Systems von **Regeln**, Umformungsregeln, in andere Größen, auch **Ausgabegrößen, Ausgabeinformationen** oder **Lösungen** genannt, umgeformt oder umgearbeitet werden.

- *Bauer,Goos, Informatik I* [BGo91](S. 47-55):

  > Ein **Algorithmus** ist eine präzise, d.h. in einer festgelegten Sprache abgefaßte, endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte.

- *B.A.Trachtenbrot, Algorithmen und Rechenautomaten* [Tra77]:

  > Unter einem **Algorithmus** versteht man eine genaue Vorschrift, nach der ein gewisses System von Operationen in einer bestimmten Reihenfolge auszuführen ist und nach der man alle Aufgaben eines gegebenen Typs lösen kann.

Ähnliche Definitionen finden sich in den Büchern [Loe76] (S. 10 ff.) sowie [Mau69] (S. 14).

Aus diesen Definitionen folgt, dass es terminierende und nicht-terminierende Algorithmen geben kann. Mit die wichtigste Eigenschaft aber ist, dass die einzelnen anwendbaren Schritte und Aktionen in jeder Situation zweifelsfrei und eindeutig bestimmt sind.

An unseren Beispielen lässt sich erkennen, dass es Probleme gibt, die sich leicht lösen lassen, während andere Probleme zu ihrer Lösung einen größeren Aufwand erfordern. Schließlich gibt es auch noch "unlösbare" Probleme, das sind solche, für die es keinen Algorithmus gibt, der stets eine Lösung liefert. Eine Behauptung, die man ohne einen Beweis natürlich nicht so im Raum stehen lassen darf!

Es ist nun ein natürliches Anliegen der Informatik zu untersuchen, welche Aufgaben überhaupt berechenbar sind und welche nicht. Lösbare Probleme sind hinsichtlich des zu ihrer Lösung benötigten Aufwandes an Ressourcen zu untersuchen, z.B. in Hinblick auf benötigte Rechenzeit und erforderlichen Speicherplatz. Diese Fragestellung ist wichtig, weil gerade in technischen Bereichen mit hohem Risiko - z.B. für den sicheren Betrieb von Kernkraftwerken oder bei der Steuerung von Flugzeugen - auf veränderte Situationen in kürzester Zeit reagiert werden muss. Längere Rechnungen sind hier nicht akzeptabel. Algorithmische Prozesse müssen in Sekundenschnelle ausgeführt werden. Es kann nur als naiver Wunsch bezeichnet werden, dass dies immer möglich sein solle.

Die Tatsache, dass die Computer immer schneller werden, könnte zu der Meinung verleiten, es sei nur eine Frage der Zeit, bis sich alle schwierigen und rechenintensiven Probleme schnell lösen ließen. Aber praktische Erfahrungen und theoretische Erkenntnisse bestätigen das Gegenteil:

1. Unlösbare Probleme werden auch bei Verwendung immer schnellerer Rechner künftig unlösbar bleiben.

2. Unter den lösbaren Problemen wird es auch in Zukunft schwer-lösbare geben. Auch schnelle Computer können daran nichts ändern.

Der zur Bestimmung einer Lösung eines Problems erforderliche Aufwand an Rechenzeit und Speicherkapazität wird als die **Komplexität** des Problems bezeichnet. Die **Komplexitätstheorie**, wird in **Konkrete Komplexitätstheorie** und **Strukturelle Komplexitätstheorie** eingeteilt. Die Konkrete Komplexitätstheorie befasst sich mit der Analyse von konkreten Algorithmen, während die Strukturelle Komplexitätstheorie die Definition, Einordnung und den Vergleich von ganzen Problemklassen untersucht. Von beiden Anteilen wird in dieser Veranstaltung ein nicht zu großer, aber dennoch der wichtigste Teil aufgenommen werden.

1.2 Historische Bemerkungen

Algorithmen gibt es, seitdem Menschen rechnen. Rechenverfahren aus Indien und Ägypten sind überliefert. Zu den ältesten Rechenverfahren gehört der sog. Euklidische Algorithmus (Euklid,um 300 v.u.Z.) zur Bestimmung des größten gemeinsamen Teilers zweier natürlichen Zahlen. Unser heutiges dezimales Stellenwertsystem stammt aus Indien, es wurde später von den Arabern übernommen.

Das Wort "Algorithmus" ist vermutlich abgeleitet von dem Namen Muhammad Ibn Musa Al-Hwârizmî, ein persischer Gelehrter, der im 9. Jahrhundert in China (heute Usbekistan) lebte, viele Bücher über Mathematik und Astronomie schrieb und der etwa um 820 n. Chr. in dem Buch *Kitab al jabr w'almuqabala* („Regeln der Wiedereinsetzung und Reduktion") über die Behandlung algebraischer Gleichungen das indische Zahlensystem und das Rechnen in diesem System beschrieb. Das Original ist leider nicht erhalten, jedoch eine lateinische Übersetzung aus dem 12. Jahrhundert mit dem Titel "Algorithmi de numero Indorum".

Über die Handelswege kam das mathematische Wissen aus dem Orient dann nach Europa. In Spanien war es dann Raymundus Lullus um 1300, der sich in seiner **Ars magna** um ein Verfahren bemühte, "*alle Wahrheiten überhaupt aufzufinden*". Verdienstvoll war seine Idee, ein allgemeines Verfahren zu entwickeln. Lullus beeinflußte die Entwicklung der Mathematik in der nachfolgenden Epoche, vor allem das algorithmische Denken.

Besonders Gottfried Wilhelm Leibniz, (1646-1716), griff die Ideen von Lullus auf und versuchte möglichst allgemeine Verfahren zu entwickeln. Er entwickelte die Ideen von Lullus weiter und verfeinerte den Begriff der Ars magna, indem er unterschied zwischen einer **Ars inveniendi**, (Entscheidungsverfahren), und einer **ars iudicandi**, (Erzeugungsverfahren, Axiomatisierungsverfahren). Darüberhinaus meinte Leibniz, dass man ein allgemeines Verfahren auf einer Maschine realisieren können müsse. Von Leibniz weiß man, dass er einer der ersten Konstrukteure einer Rechenmaschine war.

Das 19. Jahrhundert war in der Mathematik geprägt von dem allseitigen Bemühen um exakte Grundlegung der mathematischen Theorien. Glaubte man zu Beginn unseres Jahrhunderts noch, dass jedes korrekt formulierte mathematische Problem prinzipiell lösbar sei, wenn man nur genügend Anstrengung und Mühe darauf verwendet (David Hilbert, [Hil00]), so wissen wir heute, dass es Probleme gibt, die algorithmisch unlösbar sind (z.B. das **Halteproblem**). Aber selbst wenn die Existenz von Lösungen gesichert ist, bleibt die Schwierigkeit, aus der Menge möglicher Lösungen eine bestimmte auszuwählen.

1.3 Probleme und Algorithmen

Das Wort "Problem" kann aus dem griechischen Wort *problema* abgeleitet werden, das soviel wie "schwierig zu lösende Aufgabe" bedeutet. Ein "Problem" ist eine allgemeine Aufgabenstellung, für die in konkreten Fällen eine Lösung gesucht wird. Ein algorithmisches Problem lässt sich also auffassen als eine Schar von konkreten mathematischen Aufgaben einheitlichen Typs. Zu seiner Beschreibung gehört die Charakterisierung der Aufgabenschar und eine Angabe über die Beschaffenheit der Lösung.

In der Formulierung eines Problems kommen im allgemeinen gewisse Parameter vor. Im Beispiel des ggT sind dies m und n aus dem Parameterbereich $\mathbb{N}$. Werden alle in einer Problemformulierung vorkommenden Parameter spezifiziert, d.h. durch konkrete Werte aus einem geeigneten Parameterbereich ersetzt, so erhält man eine konkrete Aufgabenstellung, die wir eine Instanz des Problems nennen. Setzen wir in unserem Beispiel $m = 125$ und $n = 35$, so bedeutet $< 125, 35 >$ die konkrete Aufgabe, den größten gemeinsamen Teiler der beiden Zahlen 125 und 35, also 5, zu bestimmen.

Ein Problem - wir bezeichnen es einfach mit Π - besteht also aus einer (im allgemeinen unendlichen) Anzahl von Instanzen. Die Menge der Instanzen des Problems Π sei $\mathcal{I}(\Pi)$. Zu jeder Instanz $I \in \mathcal{I}(\Pi)$ gehört ein Lösungsraum $\mathcal{L}(I)$, der endlich, unendlich oder auch leer sein kann. Wir setzen $\mathcal{L}(\Pi) = \bigcup\limits_{I \in \mathcal{I}(\Pi)} \mathcal{L}(I)$ und nennen $\mathcal{L}(\Pi)$ den Lösungsraum von Π.

Die Begriffe "Problem", "Instanz" und "Lösung" können formal wie folgt definiert werden: Gegeben sei irgendeine Menge $\mathcal{I}$, die **Menge der Instanzen**, und eine Menge $\mathcal{L}$, die **Menge der Lösungen** (oder ein **Lösungsraum**). Ein **Problem** ist dann eine Relation

$$\Pi \subseteq \mathcal{I} \times \mathcal{L}. \tag{1}$$

Wenn $I \in \mathcal{I}$ und $l \in \mathcal{L}$ und $(I, l) \in \Pi$, so sagt man "l ist eine **Lösung** von I".

Wichtig ist: Aus der Formulierung des Problems muss eindeutig hervorgehen, welche Lösungen gesucht sind.

Beim Problem des ggT zweier natürlicher Zahlen m und n können für die Parameter beliebige Elemente aus der Menge der natürlichen Zahlen eingesetzt werden, also wäre hier $\mathcal{I} = I\!N \times I\!N$. Als Lösung ist für diesen Fall ein wohlbestimmtes Element aus dem Lösungsraum $\mathcal{L} = I\!N$ gesucht, nämlich der ggT $< m, n >$ der beiden natürlichen Zahlen. Das Problem der ggT-Berechnung wäre somit als Menge $\Pi = \{(m, n, < m, n >) \mid n, m \in I\!N \times I\!N\}$ beschreibbar. Letztlich suchen wir ein einheitliches Lösungsverfahren für alle konkreten Aufgaben dieser Art. Ein solches einheitliches Lösungsverfahren ist der **Euklidische Algorithmus**.

Im vierten Beispiel sind sämtliche Primteiler $p_1, p_2, \ldots, p_k$ der Zahl n und deren Vielfachheit $\alpha_1, \alpha_2, \ldots, \alpha_n$ gesucht, so dass

$$n = {p_1}^{\alpha_1} {p_2}^{\alpha_2} \cdots {p_k}^{\alpha_k}$$

gilt.

1.4 Problemtypen

Je nach Aufgabenstellung lassen sich verschiedene Grundtypen von Problemen unterscheiden:

1. **Entscheidungsprobleme:**

   Ein Problem Π ist ein Entscheidungsproblem, wenn der Lösungsraum $\mathcal{L}(\Pi)$ aus genau zwei Elementen besteht, also $\mathcal{L}(\Pi) = \{0, 1\}$, wobei jedes Instanz $I \in \mathcal{I}(\Pi)$ genau eine Lösung hat. Wenn die Lösung $l(I) = 1$ ist, nennen wir I eine **JA-Instanz**, wenn $l(I) = 0$ ist, heißt I eine **NEIN-Instanz** von Π. Bei einem Entscheidungsproblem besteht die Aufgabe darin, von einer gegebenen Instanz I zu entscheiden, ob $l(I) = 1$ oder $l(I) = 0$ gilt. Entscheidungsprobleme werden häufig in Form einer Frage formuliert, etwa so:

   ENTSCHEIDUNGSPROBLEM Π:

   Gegeben: Eine Instanz $I \in \mathcal{I}(\Pi)$, ein auf Π definiertes Attribut (Prädikat) p (d.i. eine Eigenschaft, die den Instanzen $I \in \mathcal{I}(\Pi)$ zukommen kann oder auch nicht).

Frage: Erfüllt I das Attribut p?

Ein typisches Entscheidungsproblem ist das vorher benannte Problem:

PRIMZAHL:

Gegeben: Eine natürliche Zahl $n \in I\!N$.
Frage: Ist n eine Primzahl?

Alternative Beschreibungsform von Entscheidungsproblemen:

Eine andere Art der Beschreibung von Entscheidungsproblemen besteht darin, eine Menge $\mathcal{I}$ von Instanzen anzugeben und dazu eine Teilmenge $\mathcal{I}_Y$ zu beschreiben, die die Menge der "JA-Instanzen" von $\mathcal{I}$ darstellt. Ein Entscheidungsproblem Π ist somit ein Paar $(\mathcal{I}, \mathcal{I}_Y)$ mit $\mathcal{I}_Y \subseteq \mathcal{I}$. Und die Aufgabe besteht darin, für eine beliebige Instanz $I \in \mathcal{I}$ zu entscheiden, ob $I \in \mathcal{I}_Y$.

Beim obigen Primzahlproblem wäre also $\mathcal{I} = I\!N$ und $\mathcal{I}_Y = \{n \mid n \in I\!N, n \text{ prim}\}$, so dass das Entscheidungsproblem als $(I\!N, \mathcal{I}_Y)$ beschrieben wäre. In diesem Sinne kann ein Entscheidungsproblem ganz einfach als formale Sprache durch die Menge $\mathcal{I}_Y$ der JA-Instanzen beschrieben werden. Als Entscheidungsalgorithmus könnte dann ein irgendwie geartetes Verfahren angesehen werden, das genau die Wörter der Menge $(I\!N, \mathcal{I}_Y)$ erkennt, als auch eine Maschine oder ein deterministischer Automat der die Sprache $(I\!N, \mathcal{I}_Y)$ akzeptiert.

2. **Weitere Problemtypen:**

   Wenn die Lösungsmengen der Instanzen $I \in \mathcal{I}(\Pi)$ mehrere Elemente enthalten, kann entweder

   (a) nach irgendeiner Lösung,

   (b) nach einer bestimmten ("optimalen"),

   (c) nach allen Lösungen gesucht werden.

   Im ersten Fall spricht man von einem **Suchproblem**, im zweiten von einem **Optimierungsproblem**, im dritten von einem **Abzählungsproblem**.

Ein Beispiel für ein Optimierungsproblem ist das sog. Rundreiseproblem, das auch als Travelling Salesman Problem bekannt ist:

RUNDREISE:

Gegeben: Ein vollständiger ungerichteter Graph $G_n = (V, E)$ mit $n := |V|$ Knotenpunkten, eine quadratische symmetrische Matrix $D \in \mathbb{N}^n$ mit Elementen $(d_{i,k}) \in \mathbb{N}$ für $i, k \in \{1, 2, \ldots, n\}$, wobei d_{ik} die "Länge" der Kante $\{i, k\} \in E$ bezeichnet.

Gesucht: Ein Kreis C durch sämtliche n Knotenpunkte von G_n (d.h. eine "Rundreise") mit minimaler Länge.

Die Länge $l(C)$ von C ist dabei die Summe der Längen d_{ik} der in C enthaltenen Kanten.

Die verschiedenen Problemtypen sind nicht unabhängig voneinander. Beispielsweise kann man dem RUNDREISEPROBLEM das folgende Entscheidungsproblem zuordnen:

TRAVELLING SALESMAN:

Gegeben: Ein beliebiger ungerichteter Graph $G_n = (V, E)$ mit $n := |V|$ Knotenpunkten, eine quadratische symmetrische Matrix $D \in \mathbb{N}^n$ mit Elementen $(d_{i,k}) \in \mathbb{N}$ für $i, k \in \{1, 2, \ldots, n\}$ und eine natürliche Zahl $L \in \mathbb{N}$.

Frage: Enthält G_n einen Kreis C der jeden der n Knotenpunkten mindestens einmal besucht und dessen Länge $l(C)$ nicht größer als L ist?

Aus einer Lösung des Problems TRAVELLING SALESMAN folgt nun nicht unmittelbar eine Lösung von RUNDREISE. Wenn man aber weiß, dass das Entscheidungsproblem TRAVELLING SALESMAN algorithmisch schwer lösbar ist, so ist daraus zu schließen, dass RUNDREISE keinesfalls leichter, also mindestens ebenso schwer zu lösen ist. Wir kommen darauf später wieder in Kapitel 6, Abschnitt 6.4 und Kapitel 8, Abschnitt 8 zurück.

1.5 Eigenschaften von Algorithmen

Seit vielen Jahrhunderten wurden in der Mathematik Algorithmen unterschiedlichster Art erdacht. Alle diese Algorithmen haben gewisse gemeinsame Eigenschaften. Diese

Eigenschaften formulieren wir als Forderungen, die ein Algorithmus erfüllen muss. Ein Algorithmus muss:

- schrittweise arbeiten (**Diskretheit**),
- nach jedem Schritt muss eindeutig feststehen, was im nächsten Schritt zu geschehen hat (**Determiniertheit**).
- die einzelnen Schritte müssen hinreichend elementar sein (**Elementarität**).
- Der Algorithmus muss nach endlichen vielen Schritten zu einer Lösung führen (**Konklusivität**) [2].
- er muss auf eine hinreichend große Klasse von Instanzen anwendbar sein (**Generalität**).
- Ein Algorithmus muss sich durch einen endlichen Text beschreiben lassen (**endliche Beschreibbarkeit**).

Offensichtlich ist diese Erklärung des Begriffs "Algorithmus" keine mathematisch exakte Definition, denn die definierenden Begriffe sind ebenfalls nur umgangssprachlich formuliert. Man spricht deshalb auch vom "intuitiven" Algorithmenbegriff, im Unterschied zu den in der ersten Hälfte unseres Jahrhunderts erfolgten Präzisierungen des Algorithmenbegriffs, auf die wir erst in den nächsten Kapiteln eingehen werden. Der intuitive Algorithmenbegriff reichte über Jahrhunderte hinweg für die Mathematiker völlig aus, denn sie erfanden und entwickelten Algorithmen für die von ihnen untersuchten Probleme.

In der zweiten Hälfte des vergangenen Jahrhunderts und zu Beginn des 20. Jahrhunderts war in den mathematischen Wissenschaften eine Bewegung zu erkennen, die sich um die exakte Grundlegung der Mathematik bemühte. Besonders David Hilbert (1862-1943) erwarb sich große Verdienste um die Grundlegung der mathematischen Wissenschaften. In seinem Buch "Grundlagen der Geometrie" legte er eine exakte axiomatische Grundlegung der Euklidischen Geometrie vor, die bis dahin im Wesentlichen noch nach den Elementen des Euklid gelehrt wurde. David Hilbert hielt auf dem 2. Internationalen

[2] Die Konklusivität für Algorithmen wird nicht immer gefordert. Vergleiche [BGo91, HUl86, KEM68] oder [Tra77].

Mathematiker-Kongreß 1900 in Paris einen berühmt gewordenen Vortrag mit dem Titel "Mathematische Probleme" und gab damit den Anstoß für weitere Forschungen auf dem Gebiet der Grundlegung der mathematischen Wissenschaften. Hilbert beschrieb in seinem Vortrag 23 ungelöste Probleme, die er selbst für besonders wichtig hielt und deren Lösung er der mathematischen Öffentlichkeit dringend empfahl. Das **zehnte Hilbertsche Problem** lautet:

10. Entscheidung der Lösbarkeit einer diophantischen Gleichung

Eine diophantische Gleichung mit irgendwelchen unbekannten und mit ganzen rationalen Zahlenkoeffizienten sei vorgelegt: *Man soll ein Verfahren angeben, nach welchem sich mittels einer endlichen Anzahl von Operationen entscheiden lässt, ob die Gleichung in ganzen Zahlen lösbar ist.*

Dieses Problem bestand also darin, einen Algorithmus zu finden, der von einer diophantischen[3] Gleichung zu entscheiden gestattet, ob sie eine ganzzahlige Lösung hat oder nicht.

Für Hilbert stand es außer jedem Zweifel, dass ein solcher Algorithmus existiert, er sei nur bislang nicht entdeckt worden.

Nach intensiven Bemühungen, einen solchen Algorithus zu finden, wurde schließlich die Hoffnung, einen zu finden, aufgegeben. Statt dessen kam die Vermutung auf, dass es möglicherweise einen solchen Algorithmus gar nicht geben kann.

Nun stand man also vor dem Problem, die Unmöglichkeit der algorithmischen Lösbarkeit von Problemen exakt zu beweisen. Dazu reichte aber der oben genannte intuitive Algorithmenbegriff nicht aus. Er war nicht geeignet für den Nachweis, dass ein Problem durch keinen Algorithmus gelöst werden kann. Eine Präzisierung des Algorithmenbegriffs wurde notwendig.

Inzwischen gibt es mehrere verschiedene präzisierte Definitionen des Begriffs "Algorithmus". Doch dazu mehr im 3. Kapitel. Wir begnügen uns zunächst mit dem intuitiven Algorithmenbegriff und beschäftigen uns mit dem Entwurf, der Analyse und Effizienz von Algorithmen.

[3] nach DIOPHANTOS. Eine diophantische Gleichung ist eine Polynomgleichung über $\mathbb{Z}$, d.h. alle Koeffizienten sind ganzzahlig, und es wird nach ganzzahligen Lösungen gesucht.

1.6 Beschreibungsformen von Algorithmen

Ein Problem oder eine Aufgabe zu formulieren ist relativ einfach, eine Lösung für eine Aufgabenstellung zu finden ist schon viel schwieriger. Und ein allgemeines Lösungsverfahren, also einen Algorithmus, zu finden, ist schon fast eine Kunst. Jedenfalls gibt es keine Methode, die beschreibt, wie man zu einem gegebenen Problem einen zugehörigen Algorithmus findet. Wir werden in Kapitel 6 unterschiedliche Entwurfstechniken ansprechen (vornehmlich an Beispielen) und einige wichtige Methoden zu deren Analyse in Kapitel 7 genauer studieren. **Beispiel 1:** (Summe der ersten n natürlichen Zahlen)

gegeben: $n \in I\!N$,
gesucht: $s = \sum_{i=1}^{n} i$.

Wir geben zunächst eine verbale Beschreibung eines Algorithmus, danach eine Beschreibung in einer programmiersprachen-ähnlichen Form.

1.1 Algorithmus

(a) verbale Beschreibung: Beginne mit Summe := 0. Addiere sukzessive zu Summe die Zahlen 1 bis n. Am Ende enthält Summe das gesuchte Resultat.

(b) programm-ähnliche Notation:

function *Summe*(n)
  {berechnet die Summe der natürlichen Zahlen von 1 bis n}
  sum $\leftarrow$ 0
  for $i \leftarrow 1$ **to** n **do** sum $\leftarrow$ sum + i
  return sum

1.2 Theorem

Für beliebiges $n \in I\!N$ gilt:

$$1 + 2 + 3 + \ldots + n = \frac{(n+1)\cdot n}{2} = \binom{n+1}{2}.$$

◇

Dies ist der kombinatorische Satz, den Gauss 1786 als 9-jähriger Schulbub bewies, und den wir sofort in ein einfaches Programm zur Lösung einsetzen werden:

1.3 Algorithmus

Die Summe der ersten n natürlichen Zahlen ist $\binom{n+1}{2}$.

Nachdem ein Algorithmus für ein Problem entworfen wurde, muss er aufgeschrieben werden und validiert werden, d.h. seine Korrektheit muss nachgewiesen werden. Wenn es nicht ganz offensichtlich ist, gehört zu jedem neu entworfenen Algorithmus die zu beweisende Aussage, dass der Algorithmus zu jeder zulässigen Eingabe (*input*) die richtige Ausgabe (*output*) liefert. Darüberhinaus muss geprüft werden, ob der Algorithmus die obengenannten Forderungen erfüllt.

Die Korrektheit des 2. Algorithmus ergibt sich einfach durch Beweis des Satzes durch vollständige Induktion über n, oder einfacher nach der direkten Methode von Gauss:

$$2s = 2 \cdot \sum_{i=1}^{n} i = \begin{array}{c} 1 + \quad 2 \; + \cdots + \; (n-1) + n \; + \\ n + (n-1) + \cdots + \; 2 \quad + \; 1 \end{array} = (n+1) \cdot n.$$

Beispiel 2:
Wir geben zwei Algorithmen an, die testen, ob eine beliebige Zeichenkette $w \in \{L, R\}^*$ ein Dyck-Wort bildet, d.h. einem vollständig geklammerten Ausdruck mit mit den Symbolen 'L' und 'R' als öffnende bzw. schließende Klammer entspricht. Die formale Sprache aller Dyck-Wörter ist $D_1 := \{w \in \{L, R\}^* \mid |w|_L = |w|_R \wedge (w = uv) \to (|u|_L \geq |v|_R)\}$[4].

1.4 Algorithmus A

Schritt 0: Eingabe sei $w \in \{L, R\}^*$.
Schritt 1: Markiere das am weitesten links stehende unmarkierte Symbol R in w.
Schritt 2: Markiere das nächstgelegene unmarkierte Symbol L links davon.
Schritt 3: Wiederhole Schritte 1 und 2 in dieser Reihenfolge solange, bis:

- **(a)** alle Symbole von w markiert sind (Erfolg und $w \in D_1$)
- **(b)** oder Schritte 1 und 2 waren *nicht beide* ausführbar, obwohl $w \neq \lambda$ ist! (Misserfolg und $w \notin D_1$).

Warum ist der folgende Algorithmus 1.5 dem Algorithmus 1.4 vorzuziehen?

[4] Für $w \in \Sigma^*, a \in \Sigma$ bedeutet $|w|_a$ die Anzahl des Vorkommens des Symbols a im Wort w

1.5 Algorithmus B

Schritt 0: Eingabe sei $w \in \{L, R\}^*$ und $k \in \mathbb{N}$.
Schritt 1: Setze $k := 0$ und beginne w von links nach rechts Symbol für Symbol zu lesen. Dabei bilde $k := k+1$ für jedes auftretende L, und $k := k-1$ für jedes auftretende R. Es gilt $w \in D_1$ genau dann, wenn k dabei niemals negativ werden musste und zum Schluß den Wert 0 besitzt!

c. Flussdiagramm: Algorithmen lassen sich auch durch sog. **Flussdiagramme** beschreiben. Wir geben hier nur ein Beispiel an, es ist selbsterklärend. Es beschreibt einen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen.

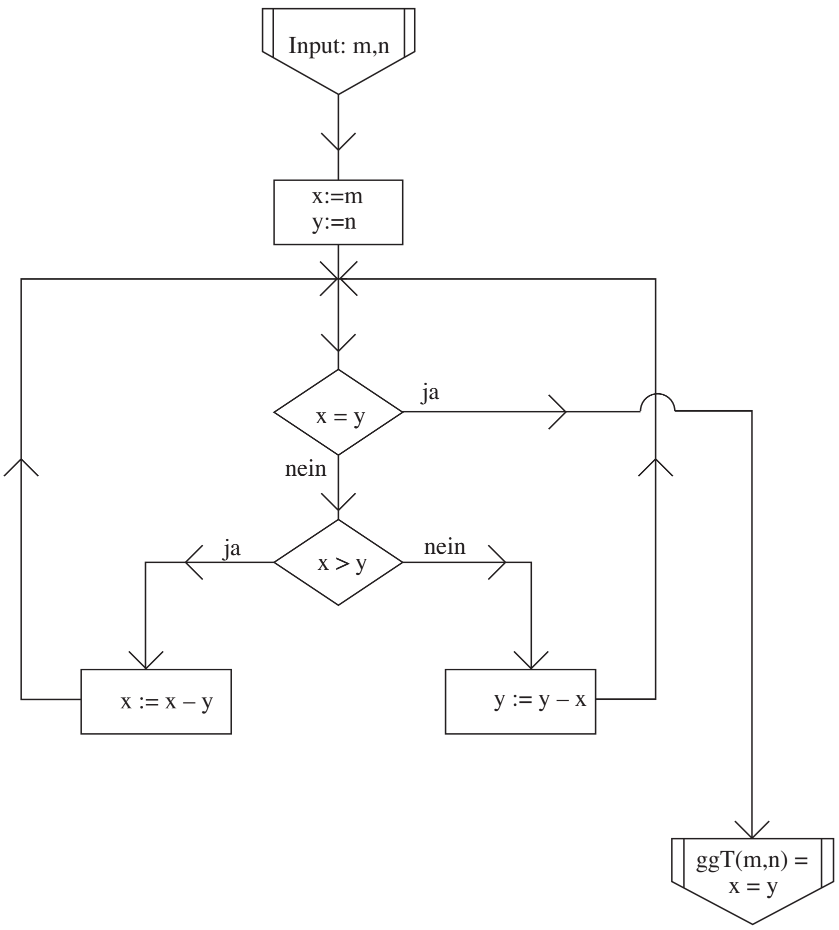


Abbildung 1: Flussdiagramm für den ggT

1.7 Die O-Notation

In der Algorithmik und Komplexitätstheorie benutzt man häufig eine 1892 von dem deutschen Mathematiker P.H. Bachmann (1837 – 1920) eingeführte und durch die Arbeiten von Edmund Landau (1877 – 1938) verbreitete Symbolik, die O-Notation (sprich: "[groß] *Oh*-Notation"). Die darauf aufbauenden Ω- und Θ-Notationen wurden später von D.E. Knuth popularisiert. Mit diesen Begriffen lässt sich der Verlauf von Funktionen qualitativ miteinander vergleichen. Sei $f : \mathbb{N} \to \mathbb{R}$ eine beliebige arithmetische, reellwertige Funktion (so bezeichnen wir Funktionen, die auf der Menge $\mathbb{N}$ definiert sind). Um Funktionen mit nicht ganzzahligen Funktionswerten durch die dann nötigen floor oder ceiling Klammern nicht unübersichtlich zu notieren, steht im Folgenden $f(x)$ stets anstelle von $\lceil f(x) \rceil$, und $f(\,)$ bezeichnet dann stets eine beliebige arithmetische, reellwertige Funktion. Oft werden auch Logarithmen vorkommen, so dass wir folgendes verabreden wollen:

1.6 Definition

Wir schreiben $\log(n)$ anstelle des oft üblichen $\log\ n$, außer in der Variante $(\log\ n)$ statt $(\log(n))$, und meinen damit:

$$\log(n) := \begin{cases} 1 & \text{, falls } n \leq 1 \\ \lceil \log_2(n) \rceil & \text{, sonst.} \end{cases}$$

$\square$

Dadurch berücksichtigen wir, dass nur ganzzahlige Werte von Speicherplatz oder Anzahlen von Schritten in Algorithmen auftreten können, und die Werte $0 = \log_2(1)$ oder $\log_2(x)$ undefiniert für $x \leq 0$, in diesem Zusammenhang nicht adäquat zum Weiterrechnen sind. Welche Basis bei den Logarithmen benutzt wird ist relativ belanglos, gilt doch bekanntlich: $\log_b(x) = \frac{\log_a(x)}{\log_a(b)} = c \cdot \log_a(x)$ für die Konstante $c := \frac{1}{\log_a(b)}$. Wir werden zudem sehen, dass konstante Faktoren für das asymptotische Verhalten von Funktionen keine Rolle spielen werden.

Da Funktionen generell auch partiell sein können, muss man bei der Anwendung der folgenden Definitionen darauf aufpassen, dass die verglichenen Funktionen nur endlich oft *undefiniert* sind.

1.7 Definition

Eine Funktion $f : \mathbb{N} \to \mathbb{R}$ wächst mit der Ordnung $g(n)$ bzw. g, notiert durch $f(n) \in O(g(n))$, falls $g : \mathbb{N} \to \mathbb{R}$ eine Funktion ist und eine Konstante $c \in \mathbb{R}^+ := \mathbb{R} \setminus \{0\}$ existiert, so dass $|f(n)| \le c \cdot |g(n)|$ für alle, bis auf endlich viele $n \in \mathbb{N}$ gilt.

Etwas knapper notiert liest sich das so:

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R} \mid (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \ge n_0)[\,|f(n)| \le c\,|g(n)|\,]\}.$$

□

Mit $O(f(n))$ bezeichnen wir also eine Menge von Funktionen. Man sagt von einer Funktion $f(n) \in O(g(n))$, sie sei **von der Ordnung** $O(g(n))$. Die O-Notation ist eine asymptotische Notation. Sie spiegelt das Verhalten einer Funktion nur für große n korrekt wider. $f(n) \in O(g(n))$ bedeutet:

- $g(n)$ und $f(n)$ haben für große n ein ähnliches Wachstumsverhalten.
- Es ist $\frac{|f(n)|}{|g(n)|} \le c$ für alle $n \ge n_0$ mit $g(n) \ne 0$.
- Es gilt: $O(a_k n^k + a_{k-1}n^{k-1} + \ldots + a_1 n + a_0) = O(n^k)$, falls $a_k \ne 0$.
- $O(f(n) + g(n)) = O(max(|f(n)|, |g(n)|)$.

Als weitere Beispiele seien f und g definiert durch: $f(n) := 12n^4 - 11n^3 + 1993$ und $g(n) := 7n^3 - n$. Dann ist $g \in O(f)$ aber nicht $f \in O(g)$. Auch gilt $f \in O(n^4)$ oder $f \in O(7028060n^4 - 1948)$. Andere Schreibweisen für $f \in O(g)$ sind in der Literatur auch $f = O(g)$ oder sehr selten $f \simeq O(g)$.

In der Regel betrachten wir Funktionen, die nicht wie $f(n) := n \cdot \sin(n)$ periodisch schwanken, sondern solche, die monoton wachsen. Ursache für die Betrachtung monoton wachsender Funktionen ist die Tatsache, dass es hier um den Ressourcenverbrauch von Algorithmen geht. Die Zahl n bezeichnet ja die Größe der Eingabe für einen Algorithmus oder eine Turing-Maschine (vergleiche Kapitel 2), und es ist zumindest anschaulich klar, dass ein Algorithmus, der ein größeres Problem lösen soll, dafür auch mehr Ressourcen braucht. In allen Fällen aber ist die Aussage wichtig, dass f nicht schneller wächst als g.

Wir definieren neben $O(f(n))$ noch weitere asymptotische Funktionenklassen, $o(g(n))$ ("klein *oh*"), $\Omega(g(n))$ ("groß Omega"), $\omega(g(n))$ ("klein Omega") und $\Theta(g(n))$ ("Theta", was in diesem Zusammenhang nur als Großbuchstabe auftritt), und sagen dann, wie vorher schon, dass eine Funktion $f(n)$ **von der Ordnung** $\mathcal{O}(g(n))$ ist, wenn sie für $\mathcal{O} \in \{O, o, \Omega, \omega, \Theta\}$ zur Menge $\mathcal{O}(g(n))$ gehört.

1.8 Definition

$o(g(n)) = \{f : I\!N \to I\!R \mid (\forall c \in I\!R^+)(\exists n_0 \in I\!N)(\forall n \geq n_0)[\,|f(n)| \leq c\,|g(n)|\,]\}$.

□

Die Tatsache, dass $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$ äquivalent zu der Beziehung $f(n) \in o(g(n))$ ist, lässt sich oftmals zum schnellen Abschätzen verwenden.

1.9 Definition

$\Omega(g(n)) = \{f : I\!N \to I\!R \mid (\exists c \in I\!R^+)(\exists n_0 \in I\!N)(\forall n \geq n_0)[\,c\,|g(n)| \leq |f(n)|\,]\}$.

□

1.10 Theorem

Für zwei beliebige Funktionen $f, g : I\!N \to I\!R$ gilt: $g(n) \in O(f(n))$ genau dann, wenn $f(n) \in \Omega(g(n))$.

◇

Beweis: als Aufgabe.

1.11 Definition

Es sei $f(n) \in \omega(g(n))$ genau dann, wenn $g(n) \in o(f(n))$

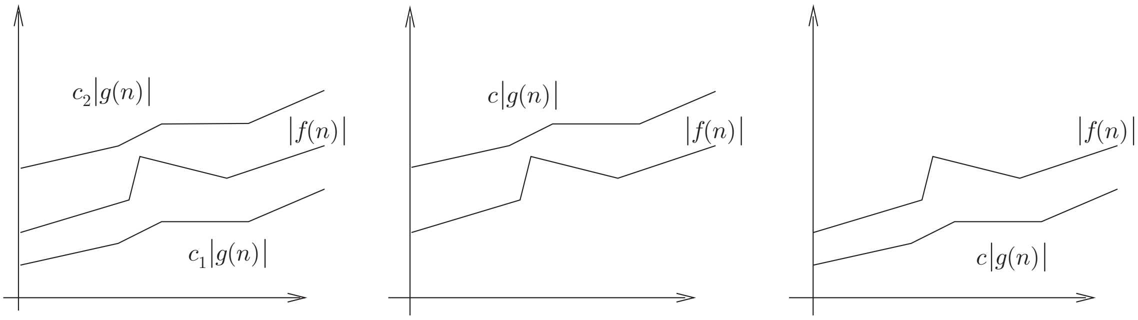
□

1.12 Definition

$\Theta(g(n)) = \{f : I\!N \to I\!R \mid (\exists c_1, c_2 \in I\!R^+)(\exists n_0 \in I\!N)(\forall n \geq n_0)[c_1\,|g(n)| \leq |f(n)| \leq c_2\,|g(n)|\,]\}$.

□

Statt $f(n) \in \Theta(g(n)), f(n) \in \Omega(g(n))$ wird in der Literatur oft auch $f(n) = \Theta(g(n))$ und $f(n) = \Omega(g(n))$ geschrieben. Wir wollen dies hier genauso wie bei der O-Notation vermeiden, weil diese (Gleichungs-)Schreibweise nicht symmetrisch ist wie die übliche Gleichungsrelation!



$f(n) \in \Theta(g(n))$ $\qquad$ $f(n) \in O(g(n))$ $\qquad$ $f(n) \in \Omega(g(n))$

Oft benutzt man die Θ-Notation, die Ω-Notation und die O-Notation mit verschiedenen Beschränkungen für die Variablen. Zum Beispiel bedeutet die Schreibweise

$$f(n) \in \Theta(g(n)) \qquad n \to \infty$$

dass es $c_1 > 0, c_2, n_0$ gibt, so dass $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)| \quad$ für $n \geq n_0$ gilt.

Beispiel: Um zu beweisen, dass

$$\frac{1}{2}n^2 - 3n \in \Theta(n^2) \qquad n \to \infty$$

muss man solche $c_1 > 0, c_2, n_0$ finden, dass

$$c_1 n^2 \leq |\frac{1}{2}n^2 - 3n| \leq c_2 n^2 \quad \text{für } n \geq n_0$$

oder (dividiert durch n^2):

$$c_1 \leq |\frac{1}{2} - \frac{3}{n}| \leq c_2 \quad \text{für } n \geq n_0$$

was wahr ist, wenn $c_1 \leq \frac{1}{14}, \quad c_2 \geq \frac{1}{2}, \quad n \geq 7$ gilt.

Beispiel: Um zu beweisen, dass $5n^3 \notin \Theta(n^2)$ gilt, setzen wir voraus, dass es c_1, n_0 gibt, so dass gilt:

$$5n^3 \leq c_1 n^2 \quad \text{für } n \geq n_0$$

Daraus folgt, dass n kleiner sein muss als $\frac{c_1}{5}$ für alle $n \geq n_0$, was unmöglich ist. Dieser Widerspruch zeigt also, dass $5n^3 \notin \Theta(n^2)$ gilt.

Man kann die Θ-, Ω- bzw. O-Notation auch für Funktionen mit reellen Variablen und auch für die Konvergenz gegen eine reelle Zahl benutzen. Zum Beispiel bedeutet

$$f(x) \in O(g(x)) \qquad x \to 0$$

dass es zwei Konstanten C, ϵ gibt, so dass gilt:

$$|f(x)| \leq C|g(x)| \qquad \text{für } |x| \leq \epsilon.$$

Es bedeutet

$$\frac{1}{2}n^3 + O(n^2) \subseteq O(n^3)$$

dass jede Funktion, die in der Menge $\frac{1}{2}n^3 + O(n^2)$ ist, auch in der Menge $O(n^3)$ ist.

Die folgende grundlegende Beziehungen zwischen Θ-, O- und Ω-Notation folgen direkt aus den Definitionen, vergleiche Theorem 10:

$$f(n) \in \Theta(g(n)) \quad \longleftrightarrow \quad f(n) \in O(g(n)) \text{ und } f(n) \in \Omega(g(n)).$$

Es gibt verschiedene Regeln, die man einfach beweisen kann und mit denen man O-Ausdrücke manipulieren kann.

$$
\begin{align}
n^m &\in O(n^{m'}) \quad \text{falls } m \leq m' \tag{2}\\
f(n) &\in O(f(n)) \tag{3}\\
cO(f(n)) &= O(f(n)) \tag{4}\\
O(O(f(n))) &= O(f(n)) \tag{5}\\
O(f(n)) + O(g(n)) &= O(|f(n)| + |g(n)|) \tag{6}\\
O(f(n))O(g(n)) &= O(f(n)g(n)) \tag{7}\\
O(f(n)g(n)) &= f(n)O(g(n)) \tag{8}
\end{align}
$$

Zum Beispiel folgt aus (2), (3), (4), (6), (8), falls $a_0 \neq 0$):

$$
\begin{aligned}
a_0 n^m + a_1 n^{m-1} + a_2 n^{m-2} + \cdots + a_{m-1} n + a_m &\in O(n^m) + O(n^m) + \cdots + O(n^m)\\
&= O(mn^m) = O(n^m)
\end{aligned}
$$

Weitere Einzelheiten über die Landau'sche O-Notation sind zu finden in [BBr96, GKP89, OWi90, Rei90] und vielen anderen Lehrbüchern zur Komplexität.

1.8 Effizienz von Algorithmen

Wenn wir mehrere Algorithmen für die Lösung ein und desselben Problems haben, versuchen wir verständlicherweise den ”besten” auszuwählen. Dazu müssen wir verschiedene Algorithmen miteinander vergleichen können. Natürlich möchten wir immer einen ”möglichst guten”, um nicht zu sagen: ”den besten” Algorithmus für ein Problem haben. Aber einen besten werden wir in den meisten Fällen nicht bekommen.

Wir stellen nun die Frage, wann ein Algorithmus gut ist. Wie können wir die Leistungsfähigkeit von Algorithmen messen?

1. Rechenzeit:

Angenommen, wir haben zwei verschiedene Algorithmen $\mathcal{A}_1(\Pi)$ und $\mathcal{A}_2(\Pi)$ zur Lösung des gleichen Problems Π, beide werden durch die gleiche Programmiersprache in ein Programm umgesetzt und beide werden auf der gleichen Maschine unter Verwendung des gleichen Compilers gestartet. Der bessere Algorithmus ist derjenige, der das Resultat in einer kürzeren Zeit ermittelt.

2. Speicherplatzbedarf:

Je weniger Speicherplatz benutzt wird, um so besser ist der Algorithmus.

3. Programmlänge:

Je kürzer das Programm, um so besser der Algorithmus. Oft kann man durch ein komplizierteres Programm die Rechenzeit verkürzen. Leider sind diese Größen abhängig von der benutzten Hardware und Software, die einer ständigen Weiterentwicklung unterliegen. Wir suchen jedoch einen hardware- und software-unabhängigen Gütebegriff für Algorithmen.

Beispiel: Bestimmung des $ggT(m, n)$:

1. Bestimmung des ggT mittels Primzahlpotenz-Darstellung:

Wir bestimmen die Primzahlpotenz-Darstellung von

$$m = p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \ldots \cdot p_k^{\alpha_k}, \quad \alpha_i \geq 0 \quad \text{für alle } i \text{ mit } 1 \leq i \leq k,$$

und

$$n = p_1^{\beta_1} \cdot p_2^{\beta_2} \cdot \ldots \cdot p_k^{\beta_k}, \quad \beta_i \geq 0 \quad \text{für alle } i \text{ mit } 1 \leq i \leq k.$$

Dann ist

$$ggT(m,n) = p_1^{\gamma_1} \cdot p_2^{\gamma_2} \cdot \ldots \cdot p_k^{\gamma_k}, \quad \gamma_i = \min\{\alpha_i, \beta_i\} \quad \text{für alle } i \text{ mit } 1 \leq i \leq k.$$

Um einen Primteiler von m zu bestimmen, müssen wir einen Teiler t von m im Bereich von 2 bis $\sqrt{m}$ finden, denn aus $m = a \cdot b$, mit $1 \leq a < b$, folgt stets $a^2 < m$, also $a < \sqrt{m}$ oder $a \leq \lfloor\sqrt{m}\rfloor$. Wir benutzen dafür die Prozedur "Teilersuche(m)".

```
procedure Teilersuche(m);
    for t:=2 to ⌊√m⌋ do
        if t|m then return(t) else
        return(m ist prim)
    end;
```

Wenn m selbst prim ist, sind wir fertig. Wenn ein Teiler t von m gefunden wurde, setzen wir $m = t \cdot m'$. Anschließend rufen wir die Prozedur Teilersuche(m') auf. Das Verfahren wird solange fortgesetzt, bis alle Primteiler von m gefunden sind. Die Prozedur Teilersuche ist $\sum\limits_{i=1}^{k} \alpha_i$ mal auszuführen.

Gleiches gilt für n. Die Bestimmung des ggT(m,n) auf diese Weise kann ziemlich aufwendig sein. Für verschiedene m und n kann die Laufzeit für dieses Verfahren ganz verschieden sein.

2. Bestimmung des ggT $< m, n >$ mit dem Euklidischen Algorithmus:

Sei O.B.d.A $m \geq n$. Wir führen fortgesetzte Division mit Rest so lange wie möglich aus:

$$\begin{array}{lcll}
m & = & q_1 \cdot n + r_1, & \textit{wobei } q_1 \in \mathbb{Z} \textit{ und } 0 \leq r_1 < n, \\
n & = & q_2 \cdot r_1 + r_2, & \textit{wobei } q_2 \in \mathbb{Z} \textit{ und } 0 \leq r_2 < r_1, \\
r_1 & = & q_3 \cdot r_2 + r_3, & \textit{wobei } q_3 \in \mathbb{Z} \textit{ und } 0 \leq r_3 < r_2, \\
\ldots\ldots & & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\
r_{k-2} & = & q_k \cdot r_{k-1} + r_k, & \textit{wobei } q_k \in \mathbb{Z} \textit{ und } 0 \leq r_k < r_{k-1}, \\
r_{k-1} & = & q_{k+1} \cdot r_k, & \textit{wobei } q_{k+1} \in \mathbb{Z}.
\end{array}$$

Die Division mit Rest ist eindeutig. Wegen $n > r_1 > r_2 > \ldots > r_k \geq 0$ bricht die Divisionskette nach endlich vielen Schritten ab. Der letzte nichtverschwindende Rest r_k ist $< m, n >$, der gesuchte ggT, denn jeder beliebige Teiler t von m und n teilt auch die Reste $r_1, r_2, \ldots, r_k$. Der letzte nichtverschwindende Rest r_k teilt auch $m, n, r_1, r_2, \ldots, r_k$.

Also ist r_k gemeinsamer Teiler von m und n, und jeder andere Teiler t von m und n teilt auch r_k. Folglich ist r_k der ggT von m und n.

Wir schätzen nun die Laufzeit des Algorithmus ab, indem wir untersuchen, wie oft die Division mit Rest im schlechtesten Fall (*worst case*) auszuführen ist. Dazu verwenden wir das

1.13 Lemma

Für $m, n \in I\!N$ mit $m \geq n$ ist stets m **mod** $n < \frac{m}{2}$.

$\diamond$

Beweis des Lemmas: O.B.d.A. sei $m \geq n$. Wenn $n \leq \frac{m}{2}$, so ist $m \bmod n < n \leq \frac{m}{2}$. Ist $n > \frac{m}{2}$, so muss der Quotient q_1 in $m = q_1 \cdot n + r_1$, wobei $0 \leq r_1 < n$ ist, notwendig 1 sein. Also ist $r_1 = m - n < m - \frac{m}{2} = \frac{m}{2}$. $\square$

Wegen Lemma 13 gilt bei einer Folge von Divisionen mit Rest stets $r_{i+1} < \frac{r_{i-1}}{2}$, also $r_{2i-1} < \frac{m}{2^i}$ und $r_{2i} < \frac{n}{2^i}$ für $1 \leq i \leq \lceil \frac{k}{2} \rceil$. Die Divisionsreste werden also abwechselnd bei jedem zweiten Schritt mindestens halbiert. Der letzte nichtverschwindende Rest r_k tritt folglich ($n \leq m$) nach höchstens $\log(m) + \log(n)$, also $\log(m \cdot n)$ Divisionsschritten auf. Wenn wir eine Division mit Rest als eine Grundoperation des Algorithmus zählen, endet der Euklidische Algorithmus im schlechtesten Fall nach höchstens $\log(m \cdot n)$ Schritten. Wenn wir darüberhinaus eine Grundoperation pro Zeiteinheit ausgeführt denken, benötigt der Algorithmus im schlechtesten Fall $O(\log\ m)$ Rechenzeit, wobei $\log(m)$ die Größenordnung der Darstellung der beiden Zahlen m und n (beachte: $m \geq n$) als Bitstrings ist.

Wir haben damit den folgenden Satz bewiesen:

1.14 Theorem (Laufzeit des Euklidischen Algorithmus')

Der Algorithmus Euklid(m, n) bestimmt korrekt $ggT(m, n)$. Seine Laufzeit ist von der Ordnung $O(\log(max(m, n)))$.

$\diamond$

In einer programm-ähnlichen Notation können wir den Euklidischen Algorithmus wie folgt aufschreiben, auch wenn diese Implementierung noch nicht die beste ist:

```
function Euklid(m, n);
    while m > 0 do
        t ← n mod m
        n ← m
        m ← t
    return n.
```

2 Turingmaschinen: Entscheidbarkeit und Berechenbarkeit

Im ersten Kapitel haben wir einfache Probleme und Algorithmen für ihre Lösung kennengelernt. Erinnern wir uns aber an das zehnte Hilbertsche Problem, dass darin bestand, ein Entscheidungsverfahren für die Lösbarkeit von diophantischen Gleichungen zu finden. Es war vermerkt worden, dass ein solcher Algorithmus trotz großer Bemühungen nicht gefunden werden konnte. Nun stellt sich die Frage, ob das etwa daran liegt, dass die Fachwissenschaftler nur noch nicht intensiv danach gesucht haben, oder vielleicht daran, dass ein solcher Algorithmus gar nicht gefunden werden kann, weil es ihn nicht gibt. Tatsächlich ist Letzteres der Fall. Wir haben also hier eine Situation, bei der für ein korrekt formuliertes Problem kein Algorithmus existiert. Wir nennen ein solches Problem **algorithmisch unlösbar**.

Wie können wir aber feststellen, ob ein Problem algorithmisch lösbar oder unlösbar ist? Dazu reicht nun der intuitive Algorithmenbegriff nicht mehr aus, es bedarf vielmehr der Präzisierung dessen, was ein Algorithmus eigentlich ist. Ein präzisierter Algorithmenbegriff muss ein mathematisch exakter Begriff sein. Seit den dreißiger Jahren des 20. Jahrhunderts sind mehrere Präzisierungen des Algorithmenbegriffs entwickelt worden.

In [Tur43] beschrieb Alan M. Turing (1912 – 1954) die nach ihm benannten Maschinen, um einen formalen Zugang zu dem Begriff der „Berechenbarkeit“ zu erhalten. Turingmaschinen simulieren dabei in gewisser Weise die algorithmische Tätigkeit des Menschen (siehe Abb.2):

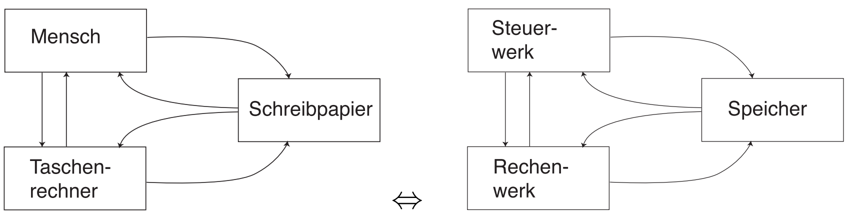


Abbildung 2: Analogie zwischen Berechnungen von Mensch und Maschine

Da wir Algorithmen nicht gut auf der Grundlage von natürlicher Sprache definieren und trotzdem Korrektheit oder Terminationsfragen formal exakt untersuchen können, benutzen wir dazu die Turing-Maschine. Diese Maschinen enthalten alle wesentlichen Bestandteile eines Algorithmus' und werden hier als Präzisierung des Algorithmusbegriffs verwendet werden. Alternative Ansätze und Definitionen findet man (1934) bei Kurt Gödel, (1906 – 1978), Rosser (1935), Stephen C. Kleene (1936), Church oder Emil L. Post im selben Jahr und auch bei Markov (1951). Die Turing'sche These besagt, dass jede intuitiv berechenbare Funktion auch von einer Turing-Maschine berechnet werden kann. Die Church'sche These behauptet das gleiche für den λ-Kalkül. Bislang sind alle formalen Definitionen für Berechenbarkeit als gleich mächtig bewiesen worden. Es ist also ausreichend, wenn man die wesentlichen Einsichten mit Hilfe des Modells der Turing-Maschine gewinnt.

Um nun die recht allgemein gehaltenen Formulierungen der obigen Verfahren auf einem - wenn auch nur formalen - Modell ausführen zu können, wird die Turing-Maschine in Anlehnung an die Arbeitsweise eines einfachen Rechners ähnlich definiert, wie die bisher benutzten abstrakten Maschinen und Automaten.

2.1 Deterministische Turing-Maschinen

Eine Turing-Maschine besitzt eine endliche Kontrolle wie ein endlicher Automat, ein in Felder unterteiltes, zweiseitig unendliches Band und einen Lese-/Schreibkopf (LSK), der in beide Richtungen über das Band von Feld zu Feld geschoben werden kann. Auf jedem Feld des Bandes kann die (hier zunächst deterministische) Turing-Maschine (DTM) mit dem LSK feststellen, welches Zeichen auf dem Feld steht. Steht dort kein Symbol, so wird dies mit einem speziellen Zeichen (hier dem „Schweinegatter" #, engl.: *sharp*) kenntlich gemacht. Dadurch sind alle Felder des Arbeitsbandes links von dem ersten und rechts von dem letzten von # verschiedenen Symbol ausschließlich mit dem Symbol # beschriftet. In der endlichen Kontrolle stehen die Angaben darüber, was die DTM nach Lesen eines Symbols tun soll. Diese Angaben werden mit einer Übergangsfunktion beschrieben, die - zusammen mit ihrem Bild- und Urbildbereich - oft auch als Programm der Turing-Maschine (Turing-Programm) bezeichnet wird.

Das hier beschriebene Modell einer DTM mit einem zweiseitig unendlichen Arbeitsband entspricht der Originaldefinition von Alan M. Turing. Es gibt aber auch Varianten, bei denen das Arbeitsband nur einseitig unendlich ist, wie z.B. in [HUl79, HUl86].

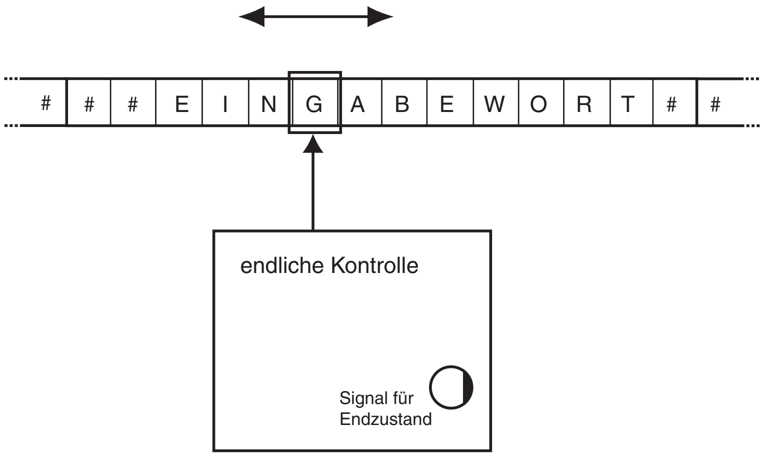


Abbildung 3: Modell einer DTM

2.1 Definition

Eine deterministische Turing-Maschine (DTM) wird beschrieben durch ein Tupel $A := (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ wobei:

> Z endliche Menge von Zuständen
>
> Γ endliches Bandalphabet
>
> Σ endliches Eingabealphabet, mit $\Sigma \subsetneq \Gamma,$ und $\Gamma \cap Z = \emptyset$
>
> $\# \in \Gamma \setminus \Sigma$ ist spezielles Symbol (für das unbeschriebene Feld)
>
> $q_0 \in Z$ Startzustand
>
> $Z_{\text{end}} \subseteq Z$ Menge der Endzustände
>
> $\delta : (Z \times \Gamma) \longrightarrow (\Gamma \times \{L, R, H\} \times Z)$ ist die partielle Übergangs-Funktion.

Die Übergangsfunktion gibt durch $\delta(p, y) = (z, b, q)$ an, was geschehen muss, wenn der LSK der DTM A im Zustand $p \in Z$ ein Zeichen $y \in \Gamma$ liest. Mit (z, b, q) wird festgelegt, dass die DTM A das Zeichen z anstelle von y auf das Feld schreibt, dann die Kopfbewegung $b \in \{L, R, H\}$ ausführt und zuletzt in den Zustand q übergeht. Hierbei bedeutet L (R) die Bewegung des LSK nach Links (bzw. Rechts), während H keine Veränderung der Kopfstellung bewirkt.

$\square$

Wichtige Bemerkungen: Es gibt Varianten dieses Grundmodells, z.B. solche mit mehreren Speicherbändern, Kellerspeichern oder Zählern und mehreren LS-Köpfen, die wir später noch kennenlernen werden. Auch auf das Stehenbleiben des Kopfes mit der Anweisung H kann verzichtet werden, indem der LSK einmal nach rechts (oder links) und dann sofort wieder nach links (bzw. rechts) gesetzt wird ohne die Bandinschrift dabei zu berücksichtigen. Aber ungeachtet bestehender Unterschiede ist festzustellen, dass alle Modelle hinsichtlich der Leistungsfähigkeit im wesentlichen gleich sind. Man kann auch sagen: die verschiedenen Varianten einer Turingmaschine sind **algorithmisch äquivalent**.

Es kann weiter gezeigt werden, dass für jede Turing-Maschine M eine äquivalente Turing-Maschine M' mit nur zwei Bandsymbolen existiert. Die Eingabe- und Bandsymbole von M werden in M' binär codiert. Wäre in diesem Fall das Symbol $\#$ für das unbeschriebene Feld nicht in dem Eingabealphabet Σ, so müssten alle Eingaben unär codiert werden. Das wiederum würde Probleme bei den Komplexitätsbetrachtungen ergeben. Bei größeren Alphabeten Σ jedoch empfiehlt sich die hier vorgenommene Einschränkung $\# \in (\Gamma \setminus \Sigma)$, weil dann das Ende der Eingabe immer (leicht) erkannt werden kann.

Zur bequemen Notation von Turing-Maschinen gibt es unterschiedliche Möglichkeiten:

Die Turing-Tabelle ist die übliche Tabelle der Überführungsfunktion:

Für DTM $A := (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \{a, b, \widehat{a}, \widehat{b}\}, \delta, q_0, \{q_4\})$

Tabelle der Übergangsfunktion (δ von A):

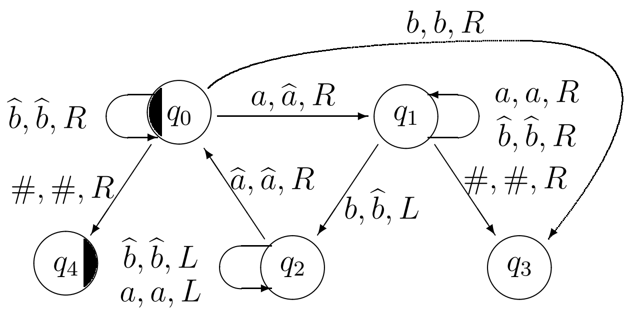
δ	$\#$	a	$\widehat{a}$	b	$\widehat{b}$
q_0	$\#, R, q_4$	$\widehat{a}, R, q_1$	–	b, R, q_3	$\widehat{b}, R, q_0$
q_1	$\#, R, q_3$	a, R, q_1	–	$\widehat{b}, L, q_2$	$\widehat{b}, R, q_1$
q_2	–	a, L, q_2	$\widehat{a}, R, q_0$	–	$\widehat{b}, L, q_2$
q_3	–	–	–	–	–
q_4	–	–	–	–	–

Die Turing-Tafel ist die lineare Auflistung der Elemente der Menge $K := \{(p, y, z, b, q) \mid \delta(p, y) = (z, b, q)\} \subseteq Z \times \Gamma \times \Gamma \times \{L, R, H\} \times Z$. Wegen der Disjunktheit der Alphabete kann beim Aufschreiben auf Kommata und Klammern verzichtet werden.

Turing-Tafel (für A):

$q_0\widehat{b}\widehat{b}Rq_0$	q_1aaRq_1	q_2aaLq_2
q_0bbRq_3	$q_1\widehat{b}\widehat{b}Rq_1$	$q_2\widehat{b}\widehat{b}Lq_2$
$q_0\#\#Rq_4$	$q_1\#\#Rq_3$	$q_2\widehat{a}\widehat{a}Rq_0$
$q_0a\widehat{a}Rq_1$	$q_1b\widehat{b}Lq_2$	–

Zustandsüberführungs- oder Transitions-Diagramm (von A):



2.2 Beispiele für Turingmaschinen

Es folgen nun vier Beispiele einfacher Turingmaschinen. Die elementaren Aufgaben, die diese DTM durchführen, sind besonders unter dem Aspekt interessant, dass komplexe Probleme oft auch dadurch gelöst werden können, indem einfache DTM'en hintereinander ausgeführt werden.

Beispiel 1: (Nachfolgerfunktion in $I\!N$)

Die Nachfolgerfunktion für natürliche Zahlen wird beschrieben durch: $N(x) = x + 1$. Sei z.B. $x = 1399$ und $N(x) = x + 1 = 1400$ gesucht.

Ausgangssituation:

		$\downarrow^{q_0}$					
...	#	1	3	9	9	#	...
Zellennummer	0	1	2	3	4	5	6

Wir geben die Überführungsfunktion in Form einer Turingtabelle an.

Tabelle der Übergangsfunktion für $N(x)$:

(Hier sind jedoch Zeilen und Spalten vertauscht, um so eine kompaktere Darstellung zu erhalten!)

	q_0	q_1	q_2
0	$0, R, q_0$	$1, H, q_2$	$0, H, q_2$
1	$1, R, q_0$	$2, H, q_2$	$1, H, q_2$
2	$2, R, q_0$	$3, H, q_2$	$2, H, q_2$
3	$3, R, q_0$	$4, H, q_2$	$3, H, q_2$
4	$4, R, q_0$	$5, H, q_2$	$4, H, q_2$
5	$5, R, q_0$	$6, H, q_2$	$5, H, q_2$
6	$6, R, q_0$	$7, H, q_2$	$6, H, q_2$
7	$7, R, q_0$	$8, H, q_2$	$7, H, q_2$
8	$8, R, q_0$	$9, H, q_2$	$8, H, q_2$
9	$9, R, q_0$	$0, L, q_1$	$9, H, q_2$
#	$\#, L, q_1$	$1, H, q_2$	$\#, H, q_2$

Die Turingmaschine beginnt ihre Arbeit im Startzustand q_0 und endet im Zustand q_2. Da dort keine Kopfbewegung mehr stattfindet, ist die letzte Spalte der Übergangsfunktion völlig überflüssig. Wenn $\delta(q_2, y)$ für kein $y \in \Gamma$ definiert wäre, so würde sich an der Arbeitsweise der DTM nichts ändern. Solche partiellen Übergangsfunktionen sind durchaus sinnvoll und waren ja auch gestattet!

Man kann den Ablauf der Berechnung einer DTM verfolgen, indem man die Systemzustände der Turing-Maschine als "Momentaufnahmen" einer Berechnung fortlaufend notiert:

Konfigurationenfolge der Berechnung von Beispiel 1:

Schritt 1

		$\downarrow q_0$					
...	#	1	3	9	9	#	...

Schritt 2

			$\downarrow q_0$				
...	#	1	3	9	9	#	...

Schritt 3

				$\downarrow q_0$			
...	#	1	3	9	9	#	...

					$\downarrow q_0$		
...	#	1	3	9	9	#	...

						$\downarrow q_0$	
...	#	1	3	9	9	#	...

					$\downarrow q_1$		
...	#	1	3	9	9	#	...

				$\downarrow q_1$			
...	#	1	3	9	0	#	...

			$\downarrow q_1$				
...	#	1	3	0	0	#	...

			$\downarrow q_2$				
...	#	1	4	0	0	#	...

Beispiel 2: (Übergang von unärer Zahlendarstellung zur Dezimaldarstellung)

Gegeben: $x = \underbrace{|||\ldots|}_{k+1 \text{ Striche}}$, die natürliche Zahl k in Unärdarstellung[5].

Gesucht: x in Dezimaldarstellung.

Idee: Die $(k+1)$ Striche auf dem Eingabeband werden jeweils durch ein Leerzeichen # ersetzt, und das Zwischenergebnis, welches links neben der Strichfolge stehen soll, wird bei 0 beginnend nach jedem Ersetzungs-Schritt um 1 erhöht.

Anfangssituation:

		$\downarrow q_0$						
...	#	|	|	|	...	|	#	...
	0	1	2	3	...	(k+1)		

[5] Um auch 0 darstellen zu können, wird $n \in I\!N$ durch $(n+1)$-maliges Auftreten eines Symbols dargestellt!

Endsituation:

	$\downarrow q_4$						
...	k	#	#	#	...	#	...
	0	1	2	3	...	(k+1)	

Hier steht natürlich im Feld mit der Nummer 0 die letzte Ziffer der Dezimaldarstellung von k.

Tabelle: (Überführungsfunktion für Unär- zu Dezimal- Konvertiertung. Zeilen und Spalten sind wieder vertauscht.)

	q_0	q_1	q_2	q_3
0	$0, R, q_0$	$0, H, q_4$	$1, H, q_0$	$1, R, q_0$
1	$1, R, q_0$	$1, H, q_4$	$2, H, q_0$	$2, R, q_0$
2	$2, R, q_0$	$2, H, q_4$	$3, H, q_0$	$3, R, q_0$
3	$3, R, q_0$	$3, H, q_4$	$4, H, q_0$	$4, R, q_0$
4	$4, R, q_0$	$4, H, q_4$	$5, H, q_0$	$5, R, q_0$
5	$5, R, q_0$	$5, H, q_4$	$6, H, q_0$	$6, R, q_0$
6	$6, R, q_0$	$6, H, q_4$	$7, H, q_0$	$7, R, q_0$
7	$7, R, q_0$	$7, H, q_4$	$8, H, q_0$	$8, R, q_0$
8	$8, R, q_0$	$8, H, q_4$	$9, H, q_0$	$9, R, q_0$
9	$9, R, q_0$	$9, H, q_4$	$0, L, q_3$	$0, L, q_3$
$\vert$	$\vert, R, q_0$	$\#, L, q_2$	$\vert, L, q_2$	$\vert, R, q_0$
#	$\#, L, q_1$	$\#, H, q_4$	$0, H, q_0$	$1, H, q_4$

Es wird empfohlen, an einfachen Beispielen die Arbeitsweise dieser Turingmaschine nachzubilden und die Folge der Konfigurationen aufzulisten.

Beispiel 3: (Addition von natürlichen Zahlen in Unärdarstellung)

Gegeben: Zwei Zahlen $m, n \in \mathbb{N}$ in Unärdarstellung.

Gesucht: Die Summe $m + n$ ebenfalls in Unärdarstellung.

Tabelle: (Überführungsfunktion) für Unär-Addieralgorithmus)

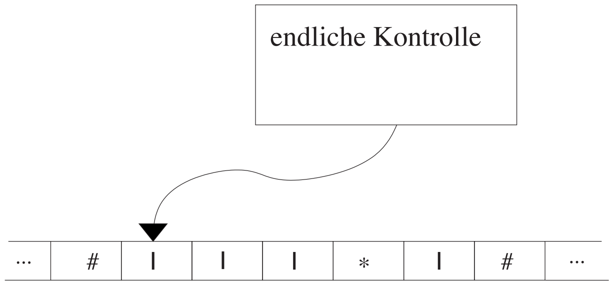
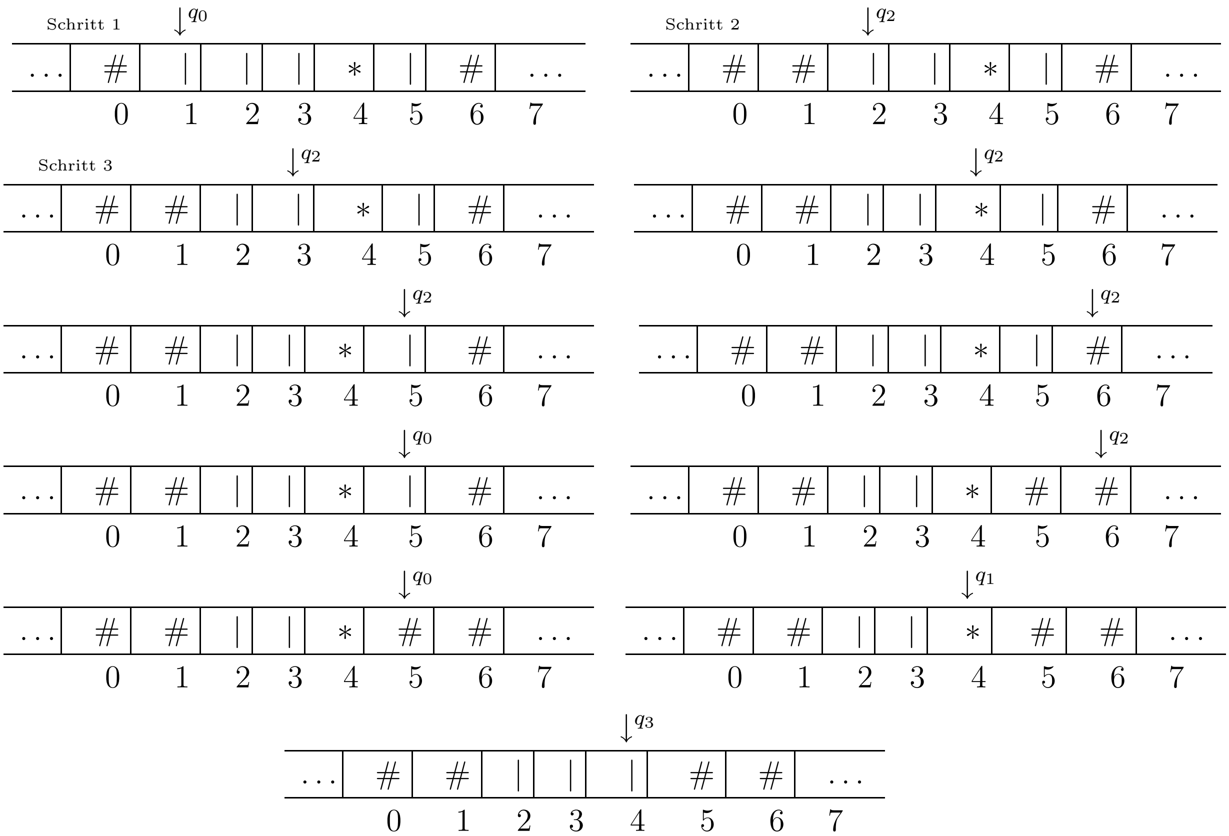


Abbildung 4: Addition von $2 + 0$ als Unärzahlen

	q_0	q_1	q_2	q_3
$\vert$	$\#, R, q_2$	$\vert, L, q_1$	$\vert, R, q_2$	$-$
$\#$	$\#, L, q_1$	$-$	$\#, L, q_0$	$-$
$*$	$-$	$\vert, H, q_3$	$*, R, q_2$	$-$

Abfolge der Situationen:



2.3 Konfigurationen der Turing-Maschine

Der (System-)Zustand der Turing-Maschine zusammen mit der wesentlichen Inschrift des Arbeitsbandes wird in der sog. Konfiguration beschrieben. Diese wird natürlich etwas einfacher sein, als die eben in den Beispielen gezeichneten Bildchen der "Situationen" der DTM. Wesentlich ist diejenige Inschrift des Arbeitsbandes, die vom am weitesten links stehenden bis zum am weitesten rechts stehenden Bandsymbol $y \in \Gamma \setminus \{\#\}$ reicht. Weiterhin wichtig ist die Stellung des LSK **auf**, **vor** oder **hinter** dieser Inschrift. Daher müssen alle #, die zwischen LSK und der wesentlichen Inschrift stehen, in der Konfiguration erfasst werden.

2.2 Definition

Ein Wort $w \in \Gamma^* \cdot Z \cdot \Gamma^*$ heißt <u>Konfiguration</u> der TM $A := (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$. Dabei bedeutet $w = upv$ mit $u, v \in \Gamma^*$ und $p \in Z$, dass A sich im Zustand p befindet, die Bandinschrift uv ist und das erste Zeichen von v sich unter dem LSK befindet. Für den Fall, dass $v = \lambda$ ist, soll # unter dem LSK stehen. Falls $v \neq \lambda$ dann ist $v \in \Gamma^*(\Gamma \setminus \{\#\})$, d.h. das am weitesten rechts stehende Symbol von v ist nicht das Symbol # und rechts von v stehen ausschließlich die Symbole # auf dem Turingband. Entsprechendes gilt für das Wort u: Falls $u \neq \lambda$ dann ist $u \in (\Gamma \setminus \{\#\})\Gamma^*$, d.h. das am weitesten links stehende Symbol von u ist nicht das Symbol # und links von u stehen ausschließlich die Symbole #. Die <u>Menge aller Konfigurationen</u> der Turing-Maschine M sei $KONF_M$.

□

Zusammenfassend sei hier noch einmal dargestellt, wie eine DTM arbeitet:

Entsprechend der Übergangsfunktion δ verändert sie durch Überschreiben des Zeichens auf dem Arbeitsband dessen Inschrift und bewegt ihren Lese-/Schreibkopf nicht oder um jeweils ein Feld nach rechts bzw. links. Ist $\delta(p, y) = (z, b, q)$, so bedeutet dies, dass die TM, wenn sie sich im Zustand p befindet und auf dem Arbeitsband das Zeichen $y \in \Gamma$ liest (wobei # für das leere Feld steht), dieses mit dem Symbol z überschreibt, danach den LSK entweder nicht bewegt ($b = H$), nach rechts ($b = R$) bzw. links ($b = L$) aufs Nachbarfeld setzt, und zuletzt in den Zustand q übergeht. Im Einzelnen wird dieses formal durch die Definition der Schrittrelation auf den Konfigurationen geregelt.

2.3 Definition

Für eine DTM A ist die Schrittrelation $\underset{A}{\vdash} \subseteq KONF_A \times KONF_A$ erklärt durch:

$w \underset{A}{\vdash} w'$ gilt genau dann, wenn für $w = uypxv$ mit $u, v \in \Gamma^*$, $x, y, z \in \Gamma$ und $p, q \in Z$, folgendes gilt:

$$w' = \begin{cases} uqyzv, & \text{falls } \delta(p,x) = (z, L, q) \\ uyzqv, & \text{falls } \delta(p,x) = (z, R, q) \\ uyqzv, & \text{falls } \delta(p,x) = (z, H, q) \end{cases}$$

Hierbei werden die Zeichenketten links und rechts des Zustandssymbols sowohl in w als auch in w' jeweils nur bis zum am weitesten außen stehenden Symbol ungleich $\#$ aufgeschrieben. Also gilt z.B. $a\#pb \underset{A}{\vdash} a\#\#q$, falls $\delta(p, b) = (\#, R, q)$, und $a\#pc \underset{A}{\vdash} aq$, falls $\delta(p, c) = (\#, L, q)$.

Mit $\overset{*}{\underset{A}{\vdash}}$ wird die reflexive, transitive Hülle von $\underset{A}{\vdash}$ bezeichnet. Wenn keine Verwirrung entstehen kann, lassen wir den Index A weg und schreiben anstelle von $\underset{A}{\vdash}$ nur $\vdash$.

□

2.4 Definition

Jede Folge von Konfigurationen $k_1, k_2, k_3, \ldots, k_i, k_{i+1}, \ldots$ die in der Relation $k_1 \vdash k_2 \vdash \ldots \vdash k_i \vdash k_{i+1} \vdash \ldots$ stehen, heißt <u>Rechnung</u> der TM. Eine endliche Rechnung $k_1 \vdash k_2 \vdash \ldots \vdash k_t$ heißt <u>Erfolgsrechnung</u>, wenn $k_1 \in \Gamma^* \cdot \{q_0\} \cdot \Gamma^*$ und $k_t \in \Gamma^* \cdot Z_{\text{end}} \cdot \Gamma^*$ gilt.

□

Turing-Maschinen akzeptieren, ähnlich wie endliche Automaten, formale Sprachen und können andererseits auch zum Berechnen von Funktionen verwendet werden. Letztlich sind dies beides aber nur unterschiedliche Sichtweisen des gleichen Sachverhalts.

2.5 Definition

Für die DTM $A = (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ bezeichnet $L(A)$ die von A akzeptierte Sprache:

$$L(A) := \{w \in \Sigma^* \mid \exists u, v \in \Gamma^* \ \exists q \in Z_{\text{end}} : \ q_0 w \overset{*}{\vdash} uqv\}$$

□

Achtung: Falls eine Turing-Maschine akzeptiert, so akzeptiert sie stets die ganze endliche Bandinschrift, auch wenn sie diese gar nicht vollständig „angeguckt“ hat!

Beispiel:

Die DTM A mit den verschiedenen Darstellungen von Seite 30 akzeptiert die Sprache $L(A) = \{a^n b^n | n \in \mathbb{N}\}$.

2.4 Berechenbarkeit

Präzisierungen des Begriffes *Berechenbarkeit* gab es, wie schon angedeutet, viele verschiedene: λ-Definierbarkeit, berechenbare arithmetische Funktionen, μ-rekursive Funktionen, Markov-Algorithmen, Post'sche Systeme und eben die Turing-Berechenbarkeit.

2.6 Definition

Sei Σ ein Alphabet. Eine (Wort-)Funktion $f : \Sigma^* \underset{p}{\longrightarrow} \Sigma^*$ heißt (Turing-)berechenbar oder partiell rekursiv genau dann, wenn es eine DTM $A = (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ gibt mit $q_0 w \underset{A}{\overset{*}{\vdash}} q_e v$, für einen Endzustand $q_e \in Z_{\text{end}}$ genau dann, wenn $f(w) = v$ ist.

□

Funktionen $f : \mathbb{N} \underset{p}{\longrightarrow} \mathbb{N}$ sind natürlich auch berechenbar wenn man eine geeignete Kodierung vornimmt. Dabei können die Zahlen unär oder binär kodiert werden, und zwar $i \in \mathbb{N}$ bei unärer Kodierung durch die Zeichenkette 0^{i+1}, wobei statt der 0 auch jedes andere Symbol verwendet werden kann. Dadurch ist es möglich, auch die natürliche Zahl $Null$ zu kodieren, was durch $0^0 = \lambda$ nicht möglich wäre. Bei binärer Kodierung wird einfach die Binärdarstellung der Zahl i (ohne führende Nullen) als Code für die Zahl i benutzt.

2.7 Definition

Eine (i.a. partielle) Funktion $f : \mathbb{N}^r \underset{p}{\longrightarrow} \mathbb{N}^s$ mit $r, s \geq 1$ heißt (Turing-)berechenbar oder partiell rekursiv genau dann, wenn eine DTM $A = (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ existiert mit $q_0 0^{m_1+1} 1 \ldots 1 0^{m_r+1} \vdash_A^* p 0^{n_1+1} 1 0^{n_2+1} 1 \ldots 1 0^{n_s+1}$, $p \in Z_{\text{end}}$ und $f(m_1, m_2, \ldots, m_r) = (n_1, n_2, \ldots, n_s)$, falls der Funktionswert definiert ist.

□

Einfache totale berechenbare Funktionen von $\mathbb{N} \times \mathbb{N}$ nach $\mathbb{N}$ sind:

$$f(m,n) := m + n, \quad f(m,n) := m \cdot n, \quad f(m,n) := n^m$$

Dass oft genug partielle Funktionen vorkommen, kennen wir alle von dem Problem bei der Division durch Null: $f : \mathbb{Z} \times \mathbb{Z} \underset{p}{\longrightarrow} \mathbb{Z}$ mit $f(p,q) := \frac{p}{q}$ ist aber nicht nur deswegen partiell, sondern auch, weil nicht jeder Bruch ganzer Zahlen wieder eine ganze Zahl ist.

2.5 Varianten der deterministischen Turing-Maschine

Wenn wir uns das von A. Turing definierte Modell der nach ihm benannten universellen Maschine ansehen, dann bemerken wir, dass das Arbeitsband dort nach beiden Seiten unbegrenzt ist, dass es aber auch in anderen Büchern, z.B. [HU179, HU186], Modelle gibt, bei denen ein einseitig unendliches Arbeitsband verwendet wird.

Letztlich ist dieses Modell genauso mächtig wie das mit dem zweiseitig unendlichem Arbeitsband.

2.8 Definition

Zwei Turingmaschinen A und B heißen genau dann äquivalent, wenn sie die gleiche Sprache akzeptieren, d.h. $L(A) = L(B)$ gilt.

□

2.9 Theorem

Zu jeder DTM A mit *beidseitig* unendlichem Arbeitsband gibt es eine äquivalente DTM B mit *einseitig* unendlichem Arbeitsband und umgekehrt.

◇

Beweis -Idee

Wir nehmen o.B.d.A. zur einfacheren Darstellung an, dass die benutzten Turingmaschinen nur die Kopfbewegungen L und R kennen.

B wird von A simuliert, indem sie sich links neben das Eingabewort eine Markierung schreibt und dann genauso wie B arbeitet. Falls dieses Feld einmal erreicht werden sollte, so muss A die Simulation beenden ohne zu akzeptieren.

A wird von B simuliert, indem B zwei Spuren auf dem Arbeitsband einrichtet, die das Band von A links und rechts des Beginns des Eingabewortes von A darstellen. War $A = (Z_A, \Sigma, \Gamma_A, \delta_A, q_{0,A}, Z_{\text{end}})$, so hat $B = (Z_B, \Sigma, \Gamma_B, \delta_B, q_{0,B}, Z_{\text{end}})$ das Bandalphabet $\Gamma_B := \{[x, y], [z, *] \mid x, y, z \in \Gamma_A\}$, wobei $* \notin \Gamma_A$ neues Symbol ist. Die endliche Kontrolle von B merkt sich, auf welchem Teil des Bandes A bei der Simulation gerade arbeitet und wechselt dies nur, wenn sie ein Zeichen $[y, *]$ liest. Im übrigen führt B jeden Schritt von A auf der Spur aus, die demjenigen Teil von A's Band zugeordnet ist, auf dem A arbeitet. Die jeweils andere Spur ignoriert B, indem sie diese nicht verändert. Die Zustandsmenge von B ist dazu $Z_B := \{[q, O], [q, U] \mid q \in Z\} \cup \{q_{0,B}\}$, wobei $[q, U]$ bedeutet, dass A im Zustand q ist und der LSK links von dem mit 0 bezeichneten Feld des Arbeitsbandes ist. Dieses Feld ist jenes, auf dem sich der LSK in der Anfangskonfiguration von A befand.

Band von A:

...			-5	-4	-3	-2	-1	0	1	2	3	4	5			...

beidseitig unendliches Band

Band von B:

0	1	2	3	4	5										...
$*$	-1	-2	-3	-4	-5										...

einseitig unendliches Band

Ist die Anfangskonfiguration von A beschrieben durch $q_{0,A}x_1x_2x_3x_4 \ldots x_n$, so sei die von B dann $q_{0,B}[x_1, \#][x_2, \#][x_3, \#] \ldots [x_n, \#]$. Die unbeschriebenen Felder des Bandes von B sind dann die Symbole $[\#, \#]$, die mit dem Zeichen $\#$ identifiziert werden.

Wenn wir mit δ_B bzw. δ_A die Überführungsfunktionen von B und A bezeichnen, so definieren wir δ_B wie folgt:

$$\delta_B(q_{0,B}, [a, \#]) := ([x, *], R, [q, O]), \text{ falls } \delta_A(q_{0,A}, a) = (x, R, q), \quad \forall a \in \Gamma_A$$

wenn A sich im ersten Schritt nach rechts *bewegt,* und

$$\delta_B(q_{0,B}, [a, \#]) := ([x, *], R, [q, U]), \text{ falls } \delta_A(q_{0,A}, a) = (x, L, q), \quad \forall a \in \Gamma_A,$$

wenn A sich im ersten Schritt nach links *bewegt.*

Für jedes $[y_1, y_2] \in \Gamma_B$ mit $y_2 \neq *$ sei

$$\delta_B([p, O], [y_1, y_2]) := ([z, y_2], t, [q, O])$$

falls $\delta_A(p, y_1) = (z, t, q)$, mit $t \in \{R, L\}$.

A wird auf der oberen Spur simuliert, da der LSK von A rechts von seiner Anfangsposition steht und

$$\begin{aligned}
\delta_B([p, U], [y_1, y_2]) &:= ([y_1, z], L, [q, U]),\ \textit{falls}\ \delta_A(p, y_2) = (z, R, q)\ \textit{bzw.} \\
\delta_B([p, U], [y_1, y_2]) &:= ([y_1, z], R, [q, U]),\ \textit{falls}\ \delta_A(p, y_2) = (z, L, q)
\end{aligned}$$

A wird auf der unteren Spur simuliert und der LSK von B bewegt sich entgegengesetzt zu dem von A.

Der Spurwechsel geschieht wie folgt:

$$\begin{aligned}
\delta_B([p, O], [x, *]) &:= \delta_B([p, U], [x, *]) := ([z, *], R, [q, O]),\ \textit{falls}\ \delta_A(p, x) = (z, R, q) \\
\delta_B([p, O], [x, *]) &:= \delta_B([p, U], [x, *]) := ([z, *], R, [q, U]),\ \textit{falls}\ \delta_A(p, x) = (z, L, q)
\end{aligned}$$

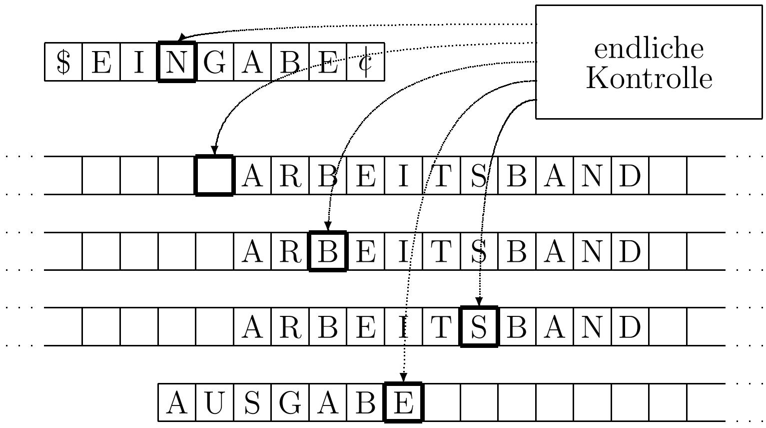
und B geht immer nach rechts.

Es sollte leicht nachzuvollziehen sein, dass jedem Übergang $k_1 \underset{A}{\vdash} k_2$ der TM A ein entsprechender Übergang $k_1' \underset{B}{\vdash} k_2'$ entspricht.

$\square$

So wie eben ein Turingband mit zwei Spuren verwendet wurde, kann man ebenso leicht, nur unter Vergrößerung des Alphabets der Bandzeichen, mehrere Spuren verwenden, um so die einzelnen Informationen darauf deutlicher werden zu lassen. Markierungen von einzelnen Zeichen der Bandinschrift können dann, ohne diese auf der ihr zugeordneten Spur zu verändern, einfach in die darunterliegende Spur geschrieben werden. Das bei einer TM mit k Spuren verwendete Alphabet Γ_A ist dann letztlich das cartesische Produkt $\Gamma_k := \underbrace{\Gamma \times \ldots \times \Gamma}_{k\text{-mal}}$ des Alphabets Γ einer TM mit nur einer einzigen Spur. Diese Überlegungen führen zum Modell der sog. mehrbändigen off-line Turing-Maschine. Die Notwendigkeit von off-line Turing-Maschinen wird sogar offensichtlich, wenn der zur

Verfügung stehenden Platz auf weniger als die Länge der Eingabe beschränkt werden soll.



2.10 Definition

Eine k-Band off-line Turing-Maschine (TM) besitzt k beidseitig unbeschränkte Arbeitsbänder mit jeweils ihrem eigenen LSK sowie ein Eingabeband, auf dem die TM ausschließlich lesen kann, aber den Lese-Kopf dabei in beide Richtungen bewegen darf (daher die Bezeichnung *off-line*). Weiterhin besitzt diese TM ein Ausgabeband, auf dem sie nur schreiben kann und den Schreibkopf ausschließlich von links nach rechts weiterbewegt.

Die Konfiguration einer k-Band off-line TM wird entsprechend definiert, wobei das Gesamtalphabet ein entsprechend großes cartesisches Produkt von Alphabeten ist, die den Bändern einzeln zugeordnet sind.

$\square$

Genauere Angaben sind hier nicht nötig und das folgende Resultat läßt sich mit dem nötigen formalen Aufwand auch exakt beweisen. Plausibel wird es sofort mit Blick auf die schon einmal verwendete Technik der Einteilung des Arbeitsbandes in Spuren.

2.11 Theorem

Zu jeder deterministischen k-Band off-line Turing-Maschine A mit $k \geq 1$ gibt es eine äquivalente DTM B mit nur einem Band.

$\diamond$

Der Beweis sei den Leser(inne)n als Übung überlassen.

2.6 Nichtdeterministische Turing-Maschinen

Ähnlich wie bei den endlichen Automaten werden auch bei Turing-Maschinen Modelle definiert, die in einer Konfiguration nicht nur höchstens eine, sondern eine feste Zahl von verschiedenen möglichen Nachfolgekonfigurationen besitzen. Diese Maschinen werden nichtdeterministische Turing-Maschinen (NTM) genannt und ganz entsprechend definiert:

2.12 Definition

$M = (Z, \Sigma, \Gamma, K, q_0, Z_{\text{end}})$ heißt nichtdeterministische Turingmaschine (NTM) genau dann, wenn zu jedem Paar $(p, y) \in Z \times \Gamma$ eine endliche Zahl von möglichen Übergängen möglich ist, und damit die Übergangs-Funktion als $\delta : Z \times \Gamma \longrightarrow 2^{\Gamma \times \{L,R,H\} \times Z}$ geschrieben werden müsste. Wir verwenden hier stattdessen $K \subseteq Z \times \Gamma \times \Gamma \times \{L, R, H\} \times Z$ als Übergangsrelation. Bei einer nichtdeterministischen TM gibt es zu jedem Zustand q und jedem Symbol x unter dem LSK im allgemeinen mehrere verschiedene Möglichkeiten für die Folgekonfiguration. Alle anderen Definitionen entsprechen denen für die DTM.

□

Auch bei Turing-Maschinen sind deterministische und nichtdeterministische Modelle äquivalent.

2.13 Theorem

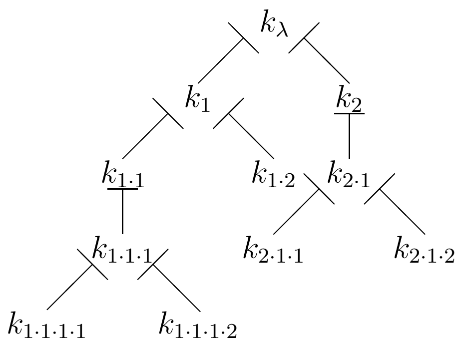
Jede von einer nichtdeterministischen Turing-Maschine erkannte Sprache (bzw. berechnete Funktion) kann auch von einer deterministischen Turing-Maschine erkannt (bzw. berechnet) werden, kurz: NTM $\widehat{=}$ DTM.

◇

Beweis

Da jede DTM auch eine NTM ist, bleibt nur noch zu zeigen, wie eine beliebige NTM $A := (Z, \Sigma, \Gamma, K, q_0, Z_{\text{end}})$ durch eine DTM $B := (Z', \Sigma, \Gamma', \delta_B, q'_0, Z'_{\text{end}})$ simuliert werden kann. Zu jedem Paar $(p, y) \in Z \times \Gamma$ existiert eine begrenzte Anzahl von Kanten in K, die wir auch als Menge $\delta(p, y)$ schreiben können. Sei $r := \max\{|\delta(p, y)|\,|p \in Z \wedge y \in \Gamma\}$ die maximal mögliche Zahl von verschiedenen Nachfolgekonfigurationen zu einer existierenden Konfiguration der fest vorgegebenen NTM A.

So wie die Knoten der i-ten Schicht eines k-nären Baumes mit Tupeln $w \in I\!N^i$, den sogenannten Baum-Adressen (*tree-domain*), eindeutig bezeichnet werden, so werden dies auch die jeweils möglichen Nachfolge-Konfigurationen. Statt eines Tupels $(n_1, n_2, \ldots, n_i) \in I\!N^i$, schreiben wir $n_1 \cdot n_2 \cdot \ldots \cdot n_i$, wie bei der Numerierung hierarchischer Inhaltsverzeichnisse. Mit k_λ werde die Anfangskonfiguration von A bezeichnet.



$k_{u\cdot i}$ ist für $1 \leq i \leq r$ diejenige Konfiguration, welche die NTM A aus k_u erreicht, wenn A sich im Zustand p befindet, das Symbol y liest und dann – bei einer vorausgesetzten linearen Ordnung der Elemente in jeder der Mengen $\delta(p, y) \subseteq \Gamma \times \{L, R, H\} \times Z$ – das i-te Element der Menge $\delta(p, y)$ benutzt. Statt der Adresse $\lambda \cdot i$ schreiben wir bei dieser Adressierungsmethode kurz i. Die DTM B hat drei Arbeitsbänder, wovon das erste die Eingabe von A enthält und auf dem zweiten Band Wörter $w \in I\!N^i$ in der sogenannten lexikalischen Ordnung auf $\bigcup\limits_{i\geq 1} I\!N^i$ generiert werden, d.h. nach aufsteigender Zahl der Komponenten und bei gleicher Komponentenzahl lexikographisch. Diese Ordnung ist total, und entspricht der bekannten Breitensuche, so dass jede Bezeichnung einer im Prinzip denkbaren (Nachfolge-) Konfiguration von k_0 in A erzeugt werden kann. Nach jeder auf dem zweiten Band erzeugten Numerierung $w \in I\!N^i$ (in Punktschreibweise), simuliert B die Rechnung von A auf dem dritten Band, indem sie die durch w vorgegebenen Überführungen vornimmt, sofern dies in der TM A überhaupt möglich ist. Erreicht B dabei eine Endkonfiguration von A, so akzeptiert B. Da jede endliche Rechnung von A auf diese Weise irgendwann einmal vorkommt, wird von B natürlich genau die Sprache $L(A)$ akzeptiert.

$\square$

2.14 Definition

Als Baum-Adressen bezeichnet man meist die unter $\prec^{\text{lg-lex}}$ geordnete Menge $\{1, \ldots, k\}^*$, wobei hier die Zahlen $1, 2, \ldots, k \in I\!N$ und die daraus gebildeten Wörter nicht als

Dezimalzahlen aufgefaßt werden, sondern als eine unter $\prec$ linear geordnete Menge von eineindeutigen Bezeichnungen der Knoten von k-nären Bäumen in der vorher beschriebenen Punktnotation. Das leere Wort λ ist dann die Bezeichnung der Wurzel.

□

Meist möchte man Turing-Maschinen nicht nur zum Akzeptieren von Wortmengen oder zum Berechnen von Funktionen verwenden, sondern auch um komplizierte Probleme algorithmisch zu lösen oder Ja-Nein-Fragen so zu entscheiden.

2.7 Entscheidbarkeit und Aufzählbarkeit

2.15 Definition

Eine Menge $M \subseteq \Sigma^*$ heißt (relativ zu Σ^*) entscheidbar genau dann, wenn die charakteristische Funktion $\chi_M : \Sigma^* \to \{0, 1\}$ berechenbar ist. Die Klasse aller entscheidbaren Mengen wird mit $\mathcal{REC}$ (*recursive sets*) bezeichnet.

□

Die von Turing-Maschinen akzeptierten Sprachen nennt man auch aufzählbare Mengen und verwendet häufig eine Definition, die die Ähnlichkeit zu den abzählbaren Mengen besonders deutlich macht.

2.16 Definition

Eine Menge $M \subseteq \Sigma^*$ heißt (rekursiv) aufzählbar genau dann, wenn $M = \emptyset$ ist, oder eine totale Turing-berechenbare Funktion $g : I\!N \to \Sigma^*$ existiert, für die $g(I\!N) = M$ ist. Die Familie aller aufzählbaren Mengen wird mit $\mathcal{RE}$ (*recursively enumerable sets*) bezeichnet.

□

Neben den entscheidbaren und den aufzählbaren Mengen gibt es auch solche, die nicht einmal aufzählbar sind. Solche Mengen werden wir bald kennenlernen.

Damit der Begriff *aufzählbar* in diesem Zusammenhang deutlicher wird, sehen wir uns das folgende Ergebnis an.

2.17 Theorem

Eine Menge M ist genau dann aufzählbar, wenn eine k-Band off-line TM existiert, die jedes Wort der Menge M genau einmal auf ihr Ausgabeband schreibt. Dies ist genau dann der Fall, wenn $M = L(A)$ für eine DTM A ist.

$\diamond$

Beweis

$M = L(A)$ für DTM $A \rightarrow \exists$ k-Band off-line TM B *und* M ist aufzählbar:

B erzeugt Paare $(i, j) \in \mathbb{N} \times \mathbb{N}$ nacheinander in lexikalischer Ordnung $<^{\text{lg-lex}}$ auf $\mathbb{N}^2$. Im Beispiel gilt: $(0,0) <^{\text{lg-lex}} (0,1) <^{\text{lg-lex}} (1,0) <^{\text{lg-lex}} (0,2) <^{\text{lg-lex}} (1,1) <^{\text{lg-lex}} (2,0) <^{\text{lg-lex}} (0,3) <^{\text{lg-lex}} (1,2) <^{\text{lg-lex}} \cdots$. Allgemein gilt: $(x,y) <^{\text{lg-lex}} (r,s)$ genau dann, wenn **entweder** $(x+y) < (r+s)$ **oder**$[(x+y) = (r+s)$ und $s < y]$. Die Reihenfolge kann einfach bestimmt werden: Zuerst werden aufsteigend Summen der zwei Komponenten generiert, die dann bei gleicher Komponentensumme in den Vektoren nach aufsteigender erster Komponente erzeugt werden. Für jedes erzeugte Paar (i, j) wird dann das i-te Wort $w_i \in \Sigma^*$ bezüglich der lexikalischen Ordnung auf Σ^* generiert, und sodann die DTM A genau j Schritte auf dem Wort w_i simuliert. Akzeptiert A in genau j Schritten, so schreibt B das Wort $w_i\#$ auf das Ausgabeband. Da jede endliche Rechnung von A auf jedem $w \in \Sigma^*$ vorkommt und bei Akzeptierung die deterministische TM A auch nur genau eine Rechnung macht, wird jedes von A akzeptierte Wort genau einmal auf das Ausgabeband geschrieben, säuberlich durch # getrennt. Auf die gleiche Weise kann eine TM konstruiert werden, die bei einer Eingabe $i \in \mathbb{N}$ nur das i-te erzeugte Wort als gesuchten Funktionswert für das Argument i auf ihr einziges Band schreibt.

k-Band off-line TM $B \rightarrow \exists$ TM $A : M = L(A)$:

A wird als TM mit $k + 2$ Spuren definiert, deren Spuren den Bändern von B zugeordnet sind und die B simuliert. Dabei vergleicht sie nacheinander jedes (auf der dem Ausgabeband von B zugeordneten Spur) erzeugte Wort w mit der ihr vorgelegten (auf die erste Spur geschriebenen) Eingabe. Stimmen beide überein, so akzeptiert A.

M aufzählbar $\rightarrow \exists$ TM $A : M = L(A)$:

Sei C die TM, die $g(\mathbb{N}) = M$ berechnet. A geht ähnlich wie oben vor: Bei Eingabe w installiert A einen Zähler $i \in \mathbb{N}$ und simuliert C mit der Eingabe i. Die von C erzeugte

Ausgabe v wird mit der Eingabe w verglichen. Stimmen beide überein, so akzeptiert A. Stimmen sie nicht überein, so wird i inkrementiert und die Simulation von C mit dem neuen i wiederholt.

$\square$

Was wir nun sehen und beweisen wollen, ist folgende Beziehung zwischen den bislang kennengelernten Klassen von Mengen:

Klasse der **entscheidbaren Mengen** $\subsetneq$ Klasse der **aufzählbaren Mengen** $\subsetneq$ Klasse der **abzählbaren Mengen** $\neq$ Klasse der **überabzählbaren Mengen**.

Dass es überabzählbare, d.h. nicht mehr abzählbare Mengen gibt, ist aus anderen Veranstaltungen (z.B. **F1**, **F2** oder **M1**) bekannt.

Bevor wir sehen, dass es Mengen gibt, die nicht aufzählbar sind, beachten wir einen Zusammenhang zwischen diesen und den entscheidbaren Mengen.

2.18 Theorem

Eine Menge $M \subseteq \Sigma^*$ ist entscheidbar genau dann, wenn M und $\Sigma^* \setminus M$ beide aufzählbar sind.

$\diamond$

Beweis

Da jede entscheidbare Menge aufzählbar ist (vgl. Theorem 2.17), braucht nur noch die Umkehrung bewiesen zu werden: Seien A_M und $A_{\overline{M}}$ Turing-Maschinen, die M bzw. $\overline{M}$ akzeptieren, aber nicht notwendigerweise stets halten. Daraus konstruieren wir eine TM B_M, die M entscheidet. B_M verwendet A_M und $A_{\overline{M}}$ leicht modifiziert, indem die Folgekonfigurationen immer abwechselnd erzeugt werden und jede TM den nächsten Übergang nur machen kann, wenn die andere nicht angehalten hatte. Beide Maschinen bekommen die gleiche Eingabe w und werden solange simuliert, bis die erste angehalten hat und das Ergebnis „$w \in M$?“ entschieden wurde.

$\square$

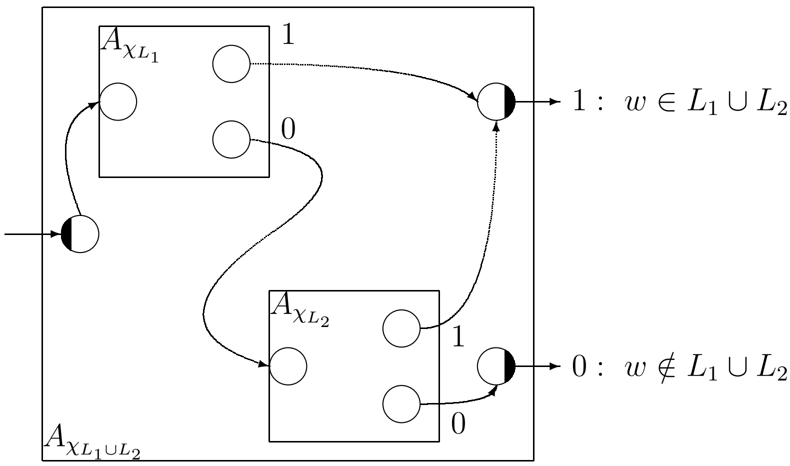
2.19 Theorem

Die Klasse der entscheidbaren Mengen bildet eine boolesche Mengen-Algebra.

◇

Beweis

Da eine Menge M entscheidbar ist, wenn ihre charakteristische Funktion χ_M berechenbar ist, brauchen wir die TM, die die Funktion χ_M berechnet, für das Komplement $\overline{M} := \Sigma^* \setminus M$ von M nur leicht zu ändern: die Ausgaben der Symbole 1 und 0 werden nur vertauscht. So erhalten wir eine TM zur Berechnung von $\chi_{\overline{M}}$ und auch $\overline{M}$ ist eine entscheidbare Menge. Wir sehen weiter an dem unten stehenden Bild, wie eine TM entworfen werden kann, die die charakteristische Funktion der Vereinigung zweier entscheidbarer Mengen berechnet. In der Skizze sind die Turing-Maschinen angedeutet, die die charakteristischen Funktionen der Mengen L_1 und L_2 berechnen. Der Zustand links ist jeweils der frühere Startzustand und die beiden Zustände rechts in jedem Bild einer dieser Turing-Maschinen waren die jeweiligen Endzustände. Diese werden entsprechend der eingezeichneten Kanten miteinander gekoppelt und die entstehende TM berechnet dann die charakteristische Funktion der Menge $L_1 \cup L_2$. Die DTM zur Entscheidung der Menge $L_1 \cap L_2 = \Sigma^* \setminus ((\Sigma^* \setminus L_1) \cup (\Sigma^* \setminus L_2))$ erhält man unter Anwendung dieser Konstruktionen ebenso einfach.



□

Das Cantor'sche Diagonalverfahren aus dem Beweis der Nichtabzählbarkeit der abzählbar unendlichen Zahlenfolgen kann auch verwendet werden, um auf der Basis von zwei bekannten aufzählbaren Mengen eine neue zu definieren, die dann nicht mehr aufzählbar sein kann. Dies ist der kürzeste Beweis für die Existenz einer solchen Menge.

Aus der Veranstaltung **F2** kennen wir den Begriff der Gödelisierung einer Menge M als Bezeichnung für eine totale, injektive, berechenbare Funktion $g : M \to I\!N$ von $M \subseteq \Sigma^*$ in die Menge $I\!N$, für die range(g) eine entscheidbare Menge ist, und auch g^{-1} berechenbar ist.

In den meisten Fällen werden wir die Gödelisierungsabbildung selbst nicht benötigen, und werden für eine Turing-Maschine A anstelle von $g(A)$ dann $\langle A \rangle$ notieren, wobei A dann als Turingtafel angesehen wird.

In den hier interessierenden Fällen, wird $I\!N = \text{range}(g)$ in der Regel durch $G^* = \text{range}(g)$, für ein endliches Alphabet G, ersetzt werden. Die lexikalische Ordnung[6] $\prec^{\text{lg}-\text{lex}}$ auf G^* ist total, und definiert eine weitere Gödelisierung $g_{lex} : G^* \to I\!N$, die dann der Abbildung g nachgeschaltet werden kann, damit letztlich $g_{lex} \circ g$ die verlangte Gödelisierung $\langle M \rangle$ von M in $I\!N$ darstellt.

Obige Gödelisierung ist nur als Beispiel gedacht. Für das folgende Theorem ist es im Prinzip völlig egal, welche Kodierungsfunktion $\langle \cdot \rangle : M \to G^*$ für die Turing-Maschinen gewählt wird, solange diese Funktion nur total, injektiv und berechenbar ist, und auch $\langle \cdot \rangle^{-1}$ berechenbar ist.

2.20 Theorem

Sei G ein Alphabet zur Kodierung von Turing-Maschinen bzw. Wörtern, sowie w_i das i-te Wort und A_i die i-te DTM in der lexikalischen Aufzählung der Wörter $\langle A_i \rangle \in G^*$. Dann ist die Menge $L_d := \{w_i \mid w_i \notin L(A_i)\}$ nicht aufzählbar.

◇

Beweis

Dass G^* in der lexikalischen Ordnung aufzählbar ist, ist offensichtlich. Die Teilmenge von G^* aller derjenigen Wörter, die Kodierung einer Turing-Maschine sind, ist ebenso

[6] Für $u, v \in \Sigma^*$ gilt $u \prec^{\text{lg}-\text{lex}} v$ genau dann, wenn entweder $|u| < |v|$ oder $u \prec^{\text{lex}} v$, wobei $\prec^{\text{lex}}$ die gewöhnliche lexikographische Anordnungsrelation ist. Es kann somit von dem Ersten, dem Zweiten und allgemein von dem i-ten Wort gesprochen werden.

aufzählbar, weil eine TM existiert, die für jedes Wort $w \in G^*$ entscheiden kann, ob dieses die zulässige Kodierung einer TM ist. Betrachtet man die unendliche Matrix mit den Spalten $w_1, w_2, w_3, \ldots$, den Zeilen $A_1, A_2, A_3, \ldots$ und den Einträgen: 1, falls $w_i \in L(A_i)$ bzw. 0, falls $w_i \notin L(A_i)$, so entspricht L_d gerade der Diagonalen, von der nur die Einträge 0 Berücksichtigung fanden. Es bleibt nur noch zu zeigen, dass L_d selbst nicht aufzählbar sein kann.

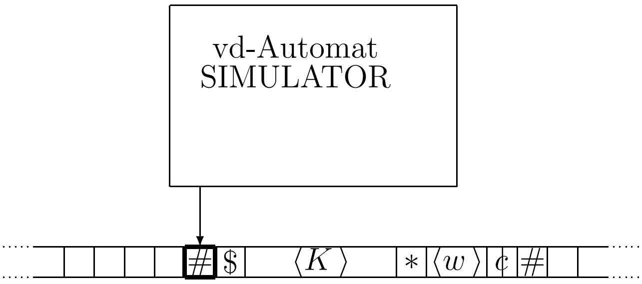
Angenommen, es sei L_d doch aufzählbar, d.h. es gibt eine TM A_j in der Aufzählung aller (Kodierungen von) Turing-Maschinen die L_d akzeptiert: $L_d = L(A_j)$. Nun gilt für das Wort w_j entweder $w_j \in L_d$ oder $w_j \notin L_d$. In beiden Fällen ergibt sich ein Widerspruch wie folgt: $w_j \in L_d \stackrel{\text{Def. von } L_d}{\longrightarrow} w_j \notin L(A_j) \stackrel{\text{Annahme}}{\longrightarrow} w_j \notin L_d$. Andererseits gilt genauso folgende Schlußkette: $w_j \notin L_d \stackrel{\text{Annahme}}{\longrightarrow} w_j \notin L(A_j) \stackrel{\text{Def. von } L_d}{\longrightarrow} w_j \in L_d$. In beiden Fällen ergibt sich also ein Widerspruch und die Annahme L_d sei aufzählbar ist nicht mehr zu halten.

$\square$

Wir stellen nun fest, dass eine Menge, die nicht aufzählbar ist, natürlich auch nicht entscheidbar sein kann. Mithin sind (wegen Theorem 2.18) $G^* \setminus L_d$ und auch L_d keine entscheidbaren Mengen. $G^* \setminus L_d = \{w_i \mid w_i \in L(A_i)\}$ ist jedoch eine aufzählbare Menge, wie die Leser(innen) bitte durch eine(n) eigene(n) Beweisidee (Beweis) zeigen mögen.

2.8 Universelle Turing-Maschine

Jeder endliche Automat ist natürlich eine spezielle Turing-Maschine, die das Arbeitsband nur von links nach rechts lesen und niemals darauf schreiben kann. Die Zustandsübergänge in der endlichen Kontrolle der TM entsprechen denen im zu simulierenden Automaten. Der Nachteil dabei ist jedoch, dass für jeden endlichen Automaten eine eigene Turing-Maschine zu entwerfen ist. Dies passt natürlich wenig zu dem Modell eines allgemeinen Algorithmusbegriffes. Eine Methode *jeden beliebigen* endlichen Automaten mit einer einzigen Turing-Maschine simulieren zu können wäre viel sinnvoller und ist tatsächlich leicht möglich.



Die obenstehend skizzierte TM kann als Simulator eines beliebigen, vollständigen DFA $A = (Z, \Sigma, K, \{q_0\}, Z_{\text{end}})$ benutzt werden. Jeder Zustand $q_i \in Z := \{q_0, q_1, \ldots, q_n\}$ wird als Zeichenkette z^{i+1} kodiert und jedes Symbol $x_i \in \Sigma := \{x_1, x_2, \ldots, x_n\}$ als x^i. Um die Gödelisierung $\langle K \rangle$ der Kantenmenge K zu erhalten, fehlen noch Symbole, um die Endzustände von anderen zu unterscheiden, Trennsymbole und Markierungszeichen. Wird die Kante $(q_i, x_j, q_k) \in K$ durch $-z^{i+1}x^j z^{k+1}$ kodiert, wobei das Zeichen $-$ (bzw. $+$) vor dem ersten z, die Tripel aus K voneinander trennt, und $+$ (im Unterschied zu $-$) den Zustand als Endzustand markiert, so kann K – wie auch das Eingabewort w des endlichen Automaten – über dem Alphabet $\{-, +, x, z\}$ kodiert werden.

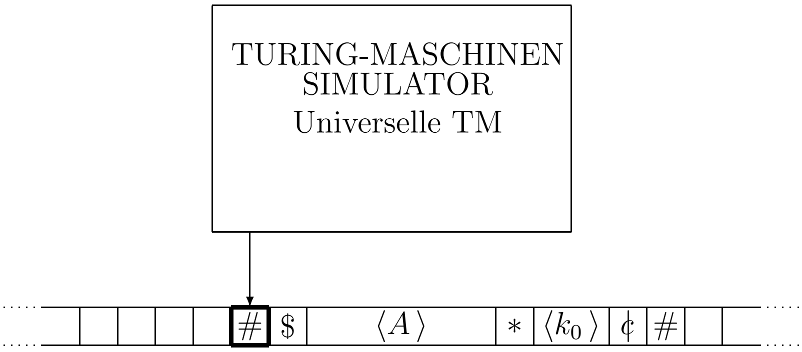
Zusätzliche Symbole können dann noch markieren, welches der bei der Simulation von A gerade erreichte Zustand ist. Es sollte klar sein, dass jeder solchermaßen kodierte DFA von der TM simuliert werden kann, indem diese, durch Markieren des in A nach Einlesen des Symbols x_j erreichten Zustandes in der kodierten Kantenmenge K, dort den abgelesenen Folgezustand aufsucht und dann diesen markiert. Bei stets erfolgender Markierung des jeweils gelesenen Buchstabens der (kodierten) Eingabe w von A kann die TM dann bei Erreichen des Endes von $\langle w \rangle$ feststellen, ob $w \in L(A)$ gilt.

Nach demselben Schema kann man eine TM entwerfen, die sogenannte *universelle Turing-Maschine* (UTM), die jede beliebige andere Turing-Maschine A simuliert. Eine beliebige TM A wird der UTM in ähnlicher Weise in kodierter (gödelisierter) Form als Zeichenkette $\langle A \rangle$ vorgesetzt, wie dies eben mit den endlichen Automaten geschehen war.

2.21 Definition

Für eine beliebige DTM A mit Anfangskonfiguration k_0 sei $\langle A \rangle$ (bzw. $\langle k_0 \rangle$) die Kodierung von A (bzw. von k_0) über dem Alphabet $G := \{0, 1\}$.

Die DTM $U := (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ heißt Universelle Turing-Maschine (UTM), wenn sie mit der Anfangskonfiguration $q_0\langle A \rangle\langle k_0 \rangle$ beginnend folgendes leistet: Für jeden Übergang $k_i \vdash k_j$, den die DTM A nach dem Start in k_0 macht, gibt es in U entsprechende Übergänge von $\langle A \rangle\langle k_i \rangle$ in mehreren Schritten nach $\langle A \rangle\langle k_j \rangle$, wobei hier nur die relevante Bandinschrift von U gemeint ist, ohne die Stellungen ihres LSK und der Zustände.
□



Es ist bei der Simulation der TM A durch die UTM nur wichtig, dass die Konfigurationen von A in der gleichen Reihenfolge durchlaufen werden. Dazwischen liegen natürlich viele andere Konfigurationen der UTM, die diese zu der Simulation braucht. Dazu muss diese in der Turing-Tafel von A nachsehen, in welchem Zustand A sich gerade befindet und in welchen Nachfolgezustand A mit welcher Aktion zu gehen hat. Entsprechend wird dann die (kodierte) Konfiguration $\langle k_0 \rangle$ geändert.

2.9 Das Halteproblem

Nicht alle Probleme sind entscheidbar. In diesem Abschnitt lernen wir einige unentscheidbare Probleme kennen und geben dafür unterschiedliche Beweismethoden an.

Das Halteproblem lautet:

Gegeben: Ein Computerprogramm P und ein Input x für P.
Frage: Kommt P für x jemals in einen Stopp-Zustand?

In Anbetracht der Tatsache, dass das Halteproblem für die Erkennung von Endlosschleifen in Programmen von Bedeutung ist, hat die Antwort auf diese Frage eine große ökonomische Bedeutung für die Programmierung. Das Halteproblem ist jedoch unentscheidbar (algorithmisch unlösbar), d.h. es gibt *keinen* Algorithmus, der für ein *beliebiges* Programm P und einen *beliebigen* Input x für P entscheidet, ob P mit Eingabe x stoppt oder nicht. Wir beweisen dieses mit einer speziellen Technik des Widerspruchsbeweises, der sogenannten Selbstanwendung. Diese führte auch in anderen Fällen zu Widersprüchen.

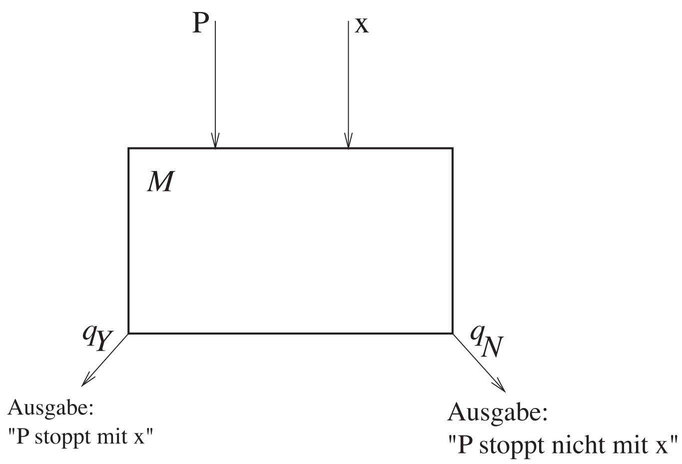
2.22 Theorem

Das HALTEPROBLEM ist unentscheidbar.

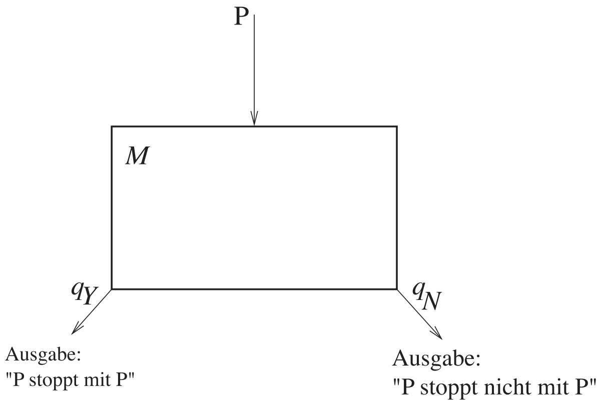
◇

Beweis

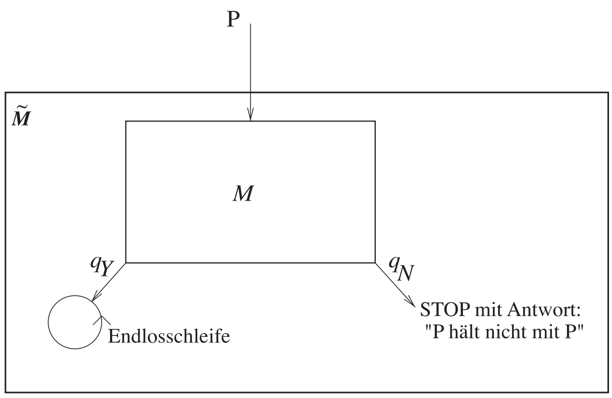
Angenommen, es gibt eine TM $\mathcal{M}$ (d.h. einen Algorithmus, der das Halteproblem für P mit Eingabe x entscheidet) wie angedeutet:



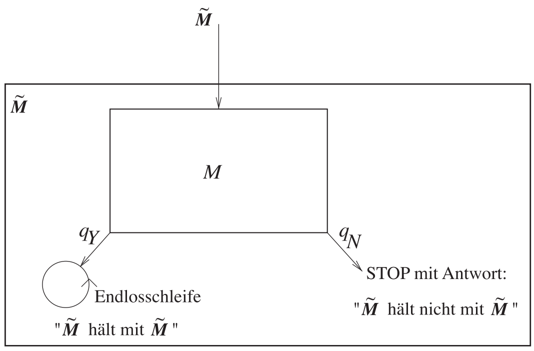
Spezialisierung : $x \equiv P$



Nun wird eine neue Turingmaschine $\tilde{M}$ konstruiert, die stets in eine Endlosschleife eintritt wenn $\mathcal{M}$ in q_Y stoppt:



Spezialfall: $P = \tilde{M}$



Wenn also $\tilde{M}$ stoppt so hat $\tilde{M}$ eine Endlosschleife und wenn $\tilde{M}$ eine Endlosschleife hat so stoppt $\tilde{M}$.

Die Annahme der Entscheidbarkeit des Halteproblems führt zu einem offensichtlichen Wiederspruch! □

Die Unentscheidbarkeit des Halteproblems besagt aber nicht, dass man etwa nicht entscheiden kann, ob ein spezieller Algorithmus für eine Eingabe s stoppt (oder für alle Eingaben stoppt).

2.10 Weitere Beispiele von Algorithmen

1. **Primzahltest:**

   Input: $n \in I\!N$.
   Frage: Ist n eine Primzahl?

   Hierzu kann folgender (trivialer) Algorithmus angegeben werden:

   Algorithmus:

   - Sei $n > 1$.
   - Für $k = 2$ bis $n - 1$ prüfe, ob $k \mid n$.

2. **Großer Fermatscher Satz:**

   Input: $n \in I\!N$ $(n > 2)$.
   Frage: Gibt es natürliche Zahlen $a, b, c \in I\!N$ mit

$$a^n + b^n = c^n? \tag{9}$$

Zum Beweis des Satzes von FERMAT: Der englische Mathematiker ANDREW WILES hat 1996, also ca. 200 Jahre, nachdem FERMAT seine Vermutung aufgestellt hatte, bewiesen, dass für kein $n > 3$ drei Zahlen a, b, c gefunden werden können, die der Gleichung (9) genügen. Der Beweis kann hier leider nicht geführt werden.

weitere Beispiele zum Halteproblem:

1. Wir betrachten den einfachen Algorithmus:

   Algorithmus $\mathcal{A}_1$:

   Input: $n \in I\!N$

(a) Setze $k := n$;

(b) solange $k \neq 1$ setze $k := k - 2$;

(c) STOP

Es kann hier leicht entschieden werden, für welche Eingaben der Algorithmus stoppt und für welche nicht. Offenbar stoppt $\mathcal{A}_1$ dann und nur dann, wenn n ungerade ist.

2. Nun betrachten wir einen anderen Algorithmus:

Algorithmus $\mathcal{A}_2$:

Input: $n \in I\!N$

(a) Setze $k := n$;

(b) solange $k \neq 1$ tue

i. wenn k gerade, dann setze $k := \frac{k}{2}$;

ii. andernfalls (k ungerade) setze $k := 3k + 1$;

(c) STOP

Beispiel: Wir wählen: $n = 17$.

$\mathcal{A}_2$ erzeugt sukzessive die Zahlenfolge: 17,52,26,13,40,20,10,5,16,8,4,2,1.

Problem (des Hamburger Mathematikers Kollatz)**:** Es ist offen, ob $\mathcal{A}_2$ für *jedes n* stoppt.

Es gibt nun eine Reihe weiterer Probleme, deren Unentscheidbarkeit nachgewiesen ist. Der Nachweis der Unentscheidbarkeit eines Problems erfolgt meist durch **Reduktion** auf ein schon als unentscheidbar bewiesenes Problem und nicht direkt wie mit der Diagonalisierungsmethode oder des Widerspruchs bei Selbstanwendung.

2.11 Reduktion des Halteproblems

Wir kodieren hier nun das schon vorgestellte Halteproblem in der Menge H als Menge von Wörtern, die die Kodierungen von Turing-Maschinen und deren Eingaben sind.

2.23 Definition

Sei $H := \Big\{ \langle A \rangle \langle w \rangle \Big| \text{ die TM } A \text{ hält bei Eingabe von } w \in \{0,1\}^* \text{ an} \Big\} \subseteq \{0,1\}^*.$

$\square$

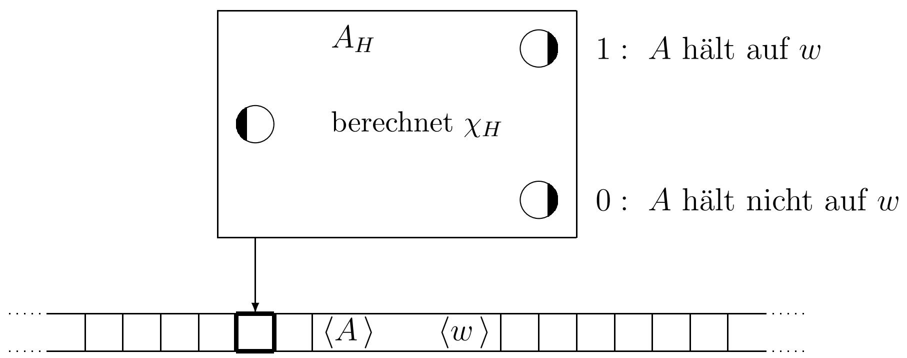
2.24 Theorem (Halteproblem für Turing-Maschinen)

Die Menge H aus Definition 2.23 ist nicht entscheidbar.

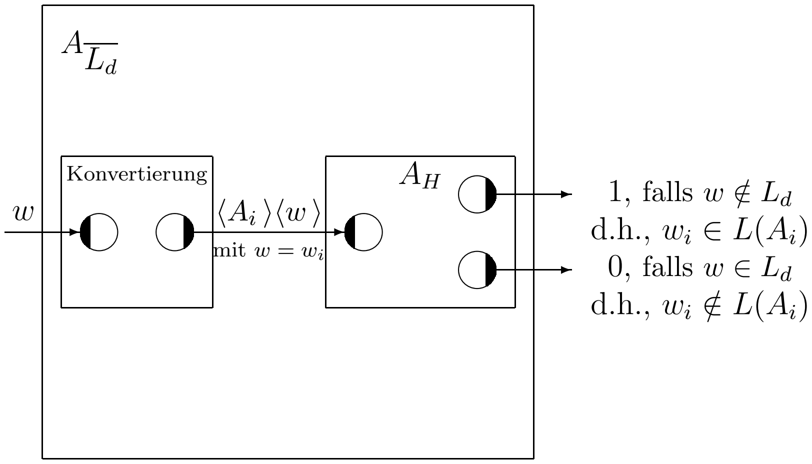
$\diamond$

Beweis (durch Reduktion)

Wir reduzieren die Menge H auf das Komplement von L_d. Mit L_d ist das Komplement $\overline{L_d}$ bekanntlich ebenfalls eine unentscheidbare Menge. Wir nehmen an, dass H entscheidbar ist, und es daher eine TM A_H für die Berechnung der charakteristischen Funktion χ_H gibt.



Unter Verwendung der TM A_H konstruieren wir eine TM, die die charakteristische Funktion von $\overline{L_d}$ berechnen könnte:



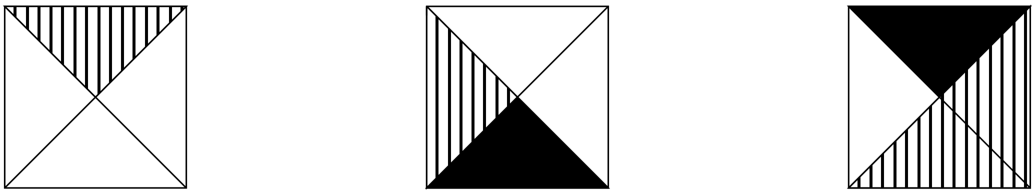
Diese neue TM $A_{\overline{L_d}}$ konstruiert aus der Eingabe von w zunächst den Index $i \in I\!N$ von w in der lexikalischen Ordnung, einfach durch Aufzählen aller Wörter in aufsteigender Reihenfolge. In gleicher Weise bestimmt sie die Kodierung der i-ten TM in der Aufzählung aller Kodierungen von Turing-Maschinen. Das Wort $\langle A_i \rangle \langle w_i \rangle$ wird dann der TM A_H als Eingabe vorgesetzt, und A_H bestimmt dann, ob A_i auf w_i hält. Tut sie dies, so kann anschließend leicht bestimmt werden, ob $w_i \in L(A_i)$ ist oder nicht. Der Widerspruch ist überzeugend, denn $\overline{L_d}$ ist ja als nicht entscheidbare Menge bekannt.

□

Hiermit ist eine weitere nichtentscheidbare Menge gefunden worden: mit L_d ist nun auch die Menge H unentscheidbar (*non-recursive*). Neben L_d ist mit dem Komplement $\{0,1\}^* \setminus H$ der Menge H jetzt eine weitere nicht aufzählbare Menge bekannt, denn H ist leicht als aufzählbar nachzuweisen und wenn auch das Komplement von H aufzählbar wäre, so müsste H entscheidbar sein, was nicht sein kann.

2.12 Das Kachelproblem

Eine Bodenfläche sei mit Kacheln auszulegen. Die Kacheln sind gleich groß, haben aber verschiedene Muster, z.B. (1), (2) oder (3).



(1) (2) (3)

Beim Aneinanderlegen dürfen nur entsprechende Seiten aneinandergelegt werden (Die Kacheln sollen orientiert sein, d.h. dürfen nicht gedreht werden!)

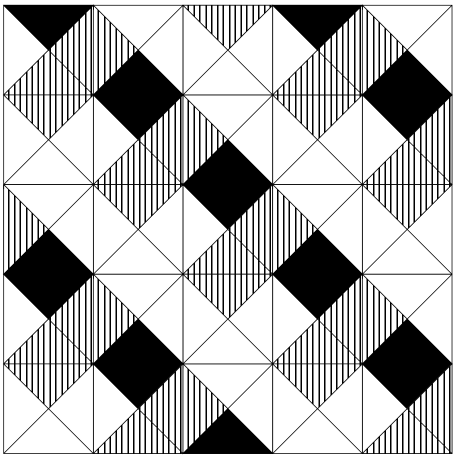


Abbildung 5: Mögliche Kachelung für das Kachelproblem

KACHELPROBLEME:

Gegeben: Eine Menge $\mathcal{R} = \{K_1, K_2, \ldots, K_r\}$ von r Kacheltypen, $n \in I\!N$, und beliebig viele Kacheln von jedem Typ.

Frage A: Kann man den ersten Quadranten der euklidischen Ebene korrekt kacheln?

Frage B: Kann man eine Fläche der Größe $n \times n$ korrekt kacheln?

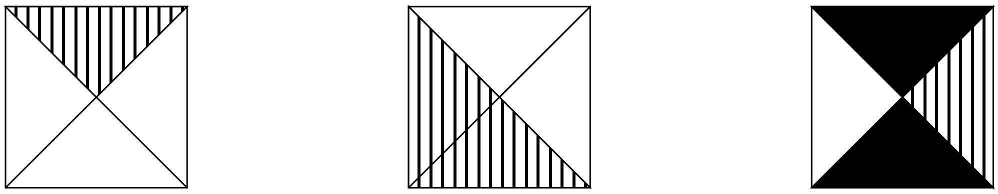
2.25 Theorem

Das Kachelproblem **A** ist unentscheidbar und für das Kachelproblem **B** gibt es zur Zeit nur deterministische Verfahren mit exponentiellem Zeitaufwand, denn das Problem ist $\mathcal{NP}$-vollständig.

◇

Der Beweis soll hier nicht geführt werden (siehe [Gru97]).

Beispiel: $n = 3, \mathcal{R} = \{K_1, K_2, K_3\}$



K_1 K_2 K_3

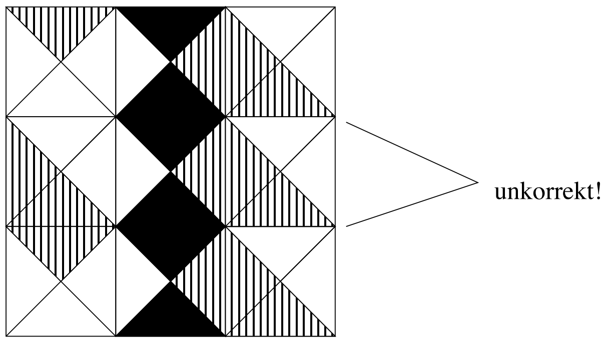


Abbildung 6: Kachelung

2.13 Weitere unentscheidbare Probleme

Da universelle Programmiersprachen die Möglichkeit geben, Interpreter (also UTM's) zu konstruieren, ist folglich die Termination dieses universellen Interpreters auf beliebigen Eingaben nicht entscheidbar. Es ist sogar schlimmer, denn sogar bei den von Haus aus stets terminierenden Programmen ist es nicht entscheidbar, ob sie etwas Sinnvolles berechnen.

2.26 Definition

Mit $\mathcal{M}_T$ sei die Menge aller (Kodierungen von) Turing-Maschinen M bezeichnet, die totale Funktionen $f_M : \Sigma^* \to \{0, 1\}$ berechnen.

□

2.27 Theorem

Es gibt keinen Algorithmus, der für eine beliebige Funktion $f_M \in \mathcal{M}_T$ entscheidet, ob $f_M(w) = 1$ für mindestens ein $w \in \Sigma^*$ gilt.

◇

Beweis (wie so häufig indirekt)

Angenommen, $\mathcal{P} := \{M \in \mathcal{M}_T \mid \exists w \in \Sigma^* : f_M(w) = 1\}$ sei eine entscheidbare Menge und $A_\mathcal{P}$ die TM, die die charakteristische Funktion $\chi_\mathcal{P}$ von $\mathcal{P}$ berechnet. Es gälte also:

$$\chi_\mathcal{P}(\langle M \rangle) = \begin{cases} 1, & \text{falls } M \in \mathcal{P} \\ 0, & \text{sonst} \end{cases}$$

Wir wählen folgende Teilmenge von $\mathcal{P}$ um einen Widerspruch herzuleiten: Die Klasse aller Turing-Maschinen $M_{B,w}$, die bei Eingabe von $n \in \mathbb{N}$ genau n Schritte der TM B bei Eingabe von w simulieren. $M_{B,w}$ soll halten und eine 1 drucken, wenn die TM B in genau n Schritten auf w hält und eine 0, wenn B dies nicht tut. Nun gilt folgendes:

$\chi_\mathcal{P}(\langle M_{B,w} \rangle) = 1$ *gdw.* $M_{B,w} \in \mathcal{P}$ *gdw.* $M_{B,w}$ druckt eine 1 bei Eingabe von n *gdw.* B hält auf w nach n Schritten an *gdw.* $w \in L(B)$.

Damit würde $A_\mathcal{P}$ aber das Halteproblem entscheiden, weil für jede TM B und jede Eingabe w die TM $M_{B,w}$ dazu konstruiert werden kann.

□

Damit ist nun z.B. folgende Frage unentscheidbar:

> **Eingabe:** eine beliebige stets haltende Turing-Maschine
>
> **Frage:** wird bei allen Eingaben stets die Ausgabe 0 berechnet?

Man gewinnt als Korollar auch folgendes Ergebnis:

2.28 Korollar

Es ist unentscheidbar, ob eine beliebige, durch eine TM definierte, entscheidbare Menge M leer ist.

◇

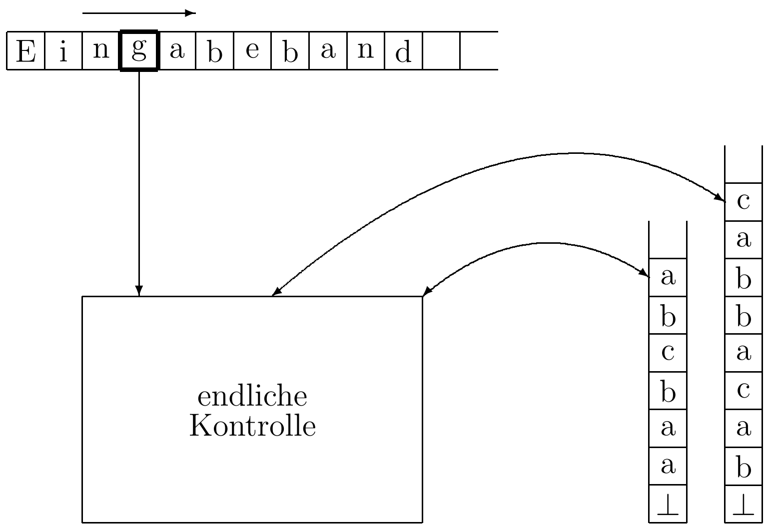
Beweis

Es gilt $M = \emptyset$ *gdw.* es gibt kein w mit $\chi_M(w) = 1$ *gdw.* $A_M \notin \mathcal{P}$

□

2.14 Weitere universelle Automatenmodelle

Turing-Maschinen haben immer die Fähigkeit, auf ihren Arbeitsbändern an jeder erreichbaren Stelle zu lesen und zu schreiben. Zur Akzeptierung von kontextfreien Sprachen verwendet man bekanntlich meist Kellerautomaten, die einen Keller als nur an einem Ende veränderbares Arbeitsband benutzen und deren Eingabeband auch nur in einer Richtung durchlaufen und dabei auch nur gelesen werden darf (*one-way* oder auch *on-line*). Wenn wir jedoch mehr als nur einen Keller, und diese alle mit einer gemeinsamen Steuerung verwenden, so erhalten wir den vollen Leistungsumfang von Turing-Maschinen! Wir definieren hier nun eine Turing-Maschine mit mehreren Kellern, anstelle des sonst üblichen Kellerautomaten aus anderen Veranstaltungen.



Maschine mit zwei unabhängigen Kellern

2.29 Definition

Ein k-Keller-Automat ist eine k-Band off-line TM mit einem one-way Eingabeband und k Arbeitsbändern, die alle am linken Ende einseitig beschränkt sind und nur in folgender Weise benutzt werden können: Der LSK liest ausschließlich das von # verschiedene Symbol am rechten Ende. Eine Veränderung des Kellers besteht aus mehreren Schritten der TM: Das oberste Symbol wird gelesen und nach dem Zustandsübergang *gelöscht, nicht verändert* oder es wird ein weiteres Symbol auf den Keller *geschrieben*. Nach jeder Kellerveränderung bewegt sich der LSK wieder über das letzte von # verschiedene Symbol. Um das linke Ende des Arbeitsbandes zu erkennen, ist jeweils zu Beginn ein spezielles Kellerbodenzeichen, z.B. $\perp$, vor das erste Symbol gesetzt worden. Es sollte klar sein, in welcher Weise eine TM diese Operationen ausführen muss.

□

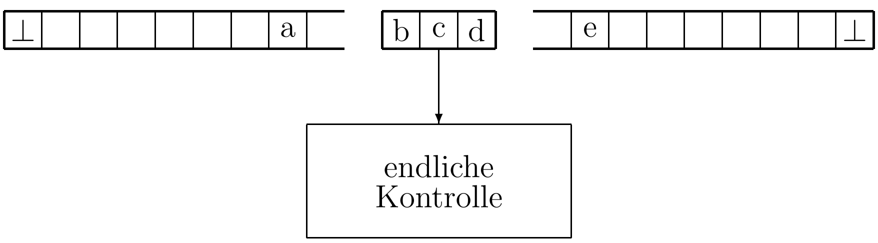
2.30 Theorem

Jede aufzählbare Menge wird auch von einem 2-Keller-Automaten akzeptiert.

◇

Beweis

Wir legen die beiden Keller waagerecht mit den oberen Enden gegeneinander zeigend. Dazwischen ist ein Pufferfeld für drei Symbole. Auf diese Weise wollen wir das beidseitig unbeschränkte Turing-Band mit den zwei Kellern simulieren. Die drei Symbole in dem Pufferfeld sind die Zeichen links, unter und rechts neben dem LSK der TM und dieses wird in der endlichen Kontrolle der 2-Keller-Maschine gehalten.



Bewegt sich der LSK nach dem Ersetzen des gelesenen Symbols nach rechts, so wird das linke Symbol aus dem Puffer auf den linken Keller geschrieben und das oberste Symbol des rechte Kellers in den Puffer ans rechte Ende gesetzt, nachdem vorher im Puffer die

Symbole alle um eine Position nach links geschoben wurden. Für die Bewegung des LSK nach links gilt entsprechendes (vertausche in der Formulierung *links* mit *rechts*). Wenn das Kellerbodenzeichen im Puffer auftaucht, so befindet sich der LSK auf einem leeren Feld mit Inschrift # und entsprechend wird dann der Keller behandelt. Die Details mögen die Leser selbst ausführen.

□

Nach Definition ist umgekehrt ein k-Keller-Automat eine spezielle Turing-Maschine, kann also auch nicht mehr leisten als diese.

Bedeutet es nun *geringere* Leistungsfähigkeit für einen 2-Keller-Automaten, wenn dieser seine Keller nur mit einem einzigen Symbol beschriften kann? Keller-Automaten, die ihren Keller *nur so* benutzen können, nennt man Zähler-Automaten.

2.31 Definition

Ein k-Zähler-Automat ist ein k-Keller-Automat, bei dem das Bandalphabet für die Keller (zusätzlich zu dem benutzten Kellerboden-Zeichen $\perp$) nur aus einem einzigen Symbol, z.B. $*$, besteht.

□

Üblicherweise identifiziert man den Kellerinhalt $\perp *^n$ mit der Zahl $n \in I\!N$ und addiert bzw. subtrahiert 1 von dem Inhalt des i-ten Zählers k_i. Jeder k-Zähler-Automat kann also als modifizierter endlicher Automat gesehen werden, der bei den Zustandsübergängen die Einweg-Eingabe liest, die Zähler verändert oder diese auf Null abfragt. In der Maschinen-Tafel eines 3-Zähler-Automaten A bedeutet z.B. der Ausdruck $(p, x, +1, -1, 0, q)$, dass A im Zustand p und bei dem Eingabesymbol x folgendes tun kann: den ersten Zähler k_1 um eins erhöhen ($k_1 := k_1 + 1$), den zweiten Zähler k_2 um eins erniedrigen, sofern dies noch möglich ist (**if** $k_2 \geq 0$ **then** $k_2 := k_2 - 1$) und prüfen ob der Zähler k_3 leer ist ($k_3 = 0?$).

2.32 Theorem

Eine Menge M ist genau dann aufzählbar, wenn sie von einem Automaten der folgenden Art erkannt werden kann:

1. Einer deterministischen Turing-Maschine (DTM).

2. Einer nichtdeterministischen Turing-Maschine (NTM).
3. Einem 2-Keller-Automaten.
4. Einem 2-Zähler-Automaten.

◇

Beweis

1., 2. und 3. wurden schon gezeigt in den Sätzen 2.13, 2.17, und 2.30. Was noch fehlt, ist der Nachweis, dass jeder 2-Zähler-Automat genauso viel kann wie ein 2-Keller-Automat. Wir werden dazu einen Keller zunächst mit zwei Zählern und danach den so entstehenden 4-Zähler-Automaten erneut mit zwei Zählern simulieren.

Für ein Kelleralphabet $\Gamma = \{y_1, y_2, \ldots, y_{k-1}\}$ mit $k-1$ Symbolen wird der Kellerinhalt $\$y_{i_1}y_{i_2}y_{i_3}\ldots y_{i_m}$ durch die Zahl $i_1 \cdot k^{m-1} + i_2 \cdot k^{m-2} + \ldots + i_{m-1} \cdot k + i_m$ in eindeutiger Weise k-när kodiert. Die Keller-Operationen werden wie folgt auf einem Zähler z durchgeführt, wobei der zweite Zähler für die nötigen Berechnungen zu Hilfe genommen wird:

push(y_i) entspricht der Änderung $z := z \cdot k + i$

pop entspricht der Änderung $z := \lfloor \frac{z}{k} \rfloor$

top$= y_i$ entspricht dem Test $i = z \bmod k$

Auf diese Weise erhält man einen 4-Zähler-Automaten, der den 2-Keller-Automaten simuliert. Diese vier Zähler kodieren wir nun anders, indem wir ihren vier Inhalten $i, j, k, l \in I\!N$, eineindeutig die Zahl $2^i 3^j 5^k 7^l$ zuordnen. Wird einer der ursprünglichen vier Zähler verändert, so muss jetzt diese Zahl „nur“ mit einer der vier zugeordneten Primzahlen $2, 3, 5$ oder 7 multipliziert bzw. dividiert werden. Geht das Ergebnis der Division ohne Rest auf, so ist der entsprechende Zähler um 1 verringert. Bleibt bei der Division ein Rest, so war der simulierte Zähler gerade auf 0 gesunken. Auch dies kann mit Hilfe des zweiten zur Verfügung stehenden Zählers erledigt werden. Details schenken wir uns hier und verweisen für die Umkehrung wieder auf die Definition.

□

3 Andere Präzisierungen der Berechenbarkeit

3.1 Primitiv-rekursive Funktionen

Ein weiteres zur DTM äquivalentes Berechnungsmodell bilden die primitiv-rekursiven Funktionen.

Die dahinterliegende Idee besteht darin, komplexere Funktionen aus einfacheren aufzubauen.

3.1 Definition

Die Menge $\mathcal{PR}$ der primitiv-rekursiven Funktionen besteht aus den folgenden Basisfunktionen und den sich daraus durch primitive Rekursion ergebenden Funktionen:

1. **Basisfunktionen:**

   Basisfunktionen sind die folgenden Funktionen:

   - Nachfolgerfunktion: $S : \mathbb{N} \to \mathbb{N} = \{0, 1, 2, \ldots\}$ mit $S(n) := n + 1$.
   - konstante Funktionen: $C_q^r : \mathbb{N}^r \to \mathbb{N}$ für $r, q \in \mathbb{N}$ mit:

     $$C_q^r(n_1, n_2, \ldots, n_r) := q$$

     für alle r-Tupel $(n_1, n_2, \ldots, n_r) \in \mathbb{N}^r$. Die nullstellige Konstante $q \in \mathbb{N}$ wird nun so ausgedrückt: $C_q^0 := q$.
   - Projektionsfunktionen: $U_i^r : \mathbb{N}^r \to \mathbb{N}$ mit:

     $$U_i^r(n_1, n_2, \ldots, n_r) := n_i$$

     für alle r-Tupel $(n_1, n_2, \ldots, n_r) \in \mathbb{N}^r$.

2. Substitution:
   Sind $f : \mathbb{N}^r \to \mathbb{N}$ und $g_1, g_2, \ldots, g_r : \mathbb{N}^m \to \mathbb{N}$ in $\mathcal{PR}$, so ist auch die Funktion $h : \mathbb{N}^m \to \mathbb{N}$ mit

   $$h(n_1, n_2, \ldots, n_m) := f(g_1(n_1, \ldots, n_m), \ldots, g_r(n_1, \ldots, n_m))$$

   in $\mathcal{PR}$.

3. primitive Rekursion:
   Sind $f : \mathbb{N}^r \to \mathbb{N}$ und $g : \mathbb{N}^{r+2} \to \mathbb{N}$ in $\mathcal{PR}$, so ist auch die folgende Funktion $h : \mathbb{N}^{r+1} \to \mathbb{N}$, die den folgenden **Rekursionsgleichungen** genügt:

$$a) \quad h(0, n_1, n_2, \ldots, n_r) := f(n_1, \ldots, n_r), \tag{10}$$

$$b) \quad h(S(n), n_1, \ldots, n_r) := g(n, h(n, n_1, \ldots, n_r), n_1, \ldots, n_r), \tag{11}$$

   in $\mathcal{PR}$.
   Man sagt: "h entsteht durch primitive Rekursion aus f und g".

4. Andere Funktionen sollen nicht zu $\mathcal{PR}$ gehören.

□

Jede primitiv-rekursive Funktion $f \in \mathcal{PR}$ ist eine **totale** Funktion, d.h. sie ist überall definiert. Außerdem ist $f(x)$ stets eindeutig bestimmt.

Beispiele:

1. **Addition in $\mathbb{N}$:** Die Addition in $\mathbb{N}$ ist durch die folgenden Rekursionsgleichungen definiert: $add : \mathbb{N}^2 \to \mathbb{N}$ mit

$$\begin{aligned} add(0, n) &:= U_1^1(n), \\ add(S(m), n) &:= g(m, add(m, n), n)) \\ g(x, y, z) &:= S(U_2^3(x, y, z)). \end{aligned}$$

   Übliche Kurzform:

$$\begin{aligned} 0 + n &:= n, \\ (m + 1) + n &:= (m + n) + 1. \end{aligned}$$

2. **Multiplikation in $\mathbb{N}$:** Die Multiplikation in $\mathbb{N}$, $mult : \mathbb{N}^2 \to \mathbb{N}$, ist definiert durch:

$$\begin{aligned} mult(0, n) &:= C_0^1(n), \\ mult(S(m), n) &:= g(m, mult(m, n), n) \\ g(x, y, z) &:= add(U_2^3(x, y, z), U_3^3(x, y, z)). \end{aligned}$$

Kurzform:

$$\begin{aligned} 0 \cdot n &:= 0, \\ (m+1) \cdot n &:= m \cdot n + n. \end{aligned}$$

3. **Exponentialfunktion:** $exp(n) = 2^n$ ist in $\mathcal{PR}$-Darstellung:

$$\begin{aligned} exp(0) &:= S(C_0^0), \\ exp(S(n)) &:= g(n, exp(n)), \\ g(x,y) &:= mult(C_2^2(x,y)), U_2^2(x,y)). \\ &\text{oder alternativ auch} \\ g(x,y) &:= mult(S(S(C_0^0)), U_2^2(x,y)). \end{aligned}$$

Etwas einfacher, und durch abkürzende Schreibweisen auf das Wesentliche verkürzt, kann diese primitive Rekursion auch so dargestellt werden:

$$\begin{aligned} exp(0) &:= 1, \\ exp(n+1) &:= 2 \cdot exp(n). \end{aligned}$$

Es gilt das

3.2 Theorem

Jede primitiv-rekursive Funktion $f : I\!N^r \to I\!N$ ist Turing-berechenbar.

◇

Die Umkehrung des Satzes ist jedoch **falsch!**

Denn es gilt:

3.3 Theorem

Nicht jede Turing-berechenbare Funktion ist total.

◇

Zum Beweis und nähere Einzelheiten siehe die Literaturliste.

Man kann nun die **Frage** stellen:

Stimmen wenigstens die totalen Turing-berechenbaren Funktionen mit den primitiv-rekursiven Funktionen überein?

Auch die Antwort darauf ist: NEIN, denn es gibt totale Turing-berechenbare Funktionen, die nicht primitiv-rekursiv sind. Ein Beispiel dafür ist die ACKERMANN-FUNKTION:

3.4 Definition

Die Ackermann-Funktion ist wie folgt definiert:

a) $f(0, n) := n + 1$,

b) $f(m + 1, 0) := f(m, 1)$,

c) $f(m + 1, n + 1) := f(m, f(m + 1, n))$.

□

Die Ackermann-Funktion ist Turing-berechenbar, ihre Funktionswerte sind für alle Argumente eindeutig bestimmt.

3.5 Theorem

Die Ackermann-Funktion ist nicht primitiv-rekursiv.

◇

Der Beweis dieses Satzes beruht auf dem folgenden

3.6 Lemma

Sei $g : \mathbb{N}^r \to \mathbb{N}$ primitiv-rekursiv. Dann gibt es ein $c \in \mathbb{N}$, so dass

$$g(n_1, n_2, \ldots, n_r) < f(c, n_1 + n_2 + \ldots + n_r)$$

für alle $(n_1, n_2, \ldots, n_r) \in \mathbb{N}^r$.

◇

Beweisidee von Theorem 3.5:

Angenommen, die Ackermann-Funktion f ist primitiv-rekursiv. Dann ist auch die Funktion

$$g(n) := f(n,n)$$

primitiv-rekursiv. Denn $g(n) = f(U_1^2(n,m), U_1^2(n,m)) \in \mathcal{PR}$. Nach Lemma 3.6 gibt es ein $c \in I\!N$ mit

$$g(n) < f(c,n) \quad \text{für alle} \quad n \in I\!N.$$

Für das spezielle $c = n$ gilt dann aber:

$$g(c) < f(c,c) = g(c).$$

Das ist aber ein Widerspruch. Folglich ist die Ackermann-Funktion nicht primitiv-rekursiv.

□

Die primitiv-rekursiven Funktionen reichen also nicht aus, um alle berechenbaren Funktionen zu charakterisieren. Es ist dafür eine umfassendere Funktionenklasse nötig, die wir nun definieren werden.

3.2 μ-rekursive Funktionen

Wir definieren zunächst den μ-Operator:

3.7 Definition

Sei $f : I\!N^{n+1} \to I\!N$ eine beliebige partielle (im allgemeinen also nicht totale) Funktion und $(x_1, x_2, \dots, x_n)$ ein beliebiges, festes n-Tupel aus $I\!N^n$. Dann betrachten wir die Folge von existierenden (!) Funktionswerten:

$$\begin{aligned} y_0 &:= f(x_1, x_2, \dots, x_n, 0), \\ y_1 &:= f(x_1, x_2, \dots, x_n, 1), \\ &\vdots \\ y_k &:= f(x_1, x_2, \dots, x_n, k), \\ &\vdots \end{aligned}$$

und fragen, wann in der Folge zum ersten Mal der Wert 0 auftritt. Sei etwa $y_k = 0$. Wenn dann $f(x_1, x_2, \dots, x_n, i)$ für alle i mit $0 \leq i < k$ definiert ist, dann setzen wir:

$$\psi(x_1, x_2, \dots, x_n) := \mu y[f(x_1, x_2, \dots, x_n, y) = 0] := y_k$$

ψ ist eine arithmetische Funktion mit $\psi : \mathbb{N}^{n+1} \to \mathbb{N}$. Wir sagen: ψ wird **durch den μ-Operator angewandt auf f definiert** oder ψ entsteht aus f durch **Minimalisierung**.

□

Wenn in der Folge $f(x_1, x_2, \ldots, x_n, i), 0 \leq i < k$, niemals der Wert 0 auftritt und wenn $f(x_1, x_2, \ldots, x_n, k)$ nicht definiert ist, dann sei $\psi(x_1, x_2, \ldots, x_n)$ nicht definiert.

3.8 Definition

Eine Funktion f heißt μ-rekursiv (partiell-rekursiv), wenn f entweder

1. eine primitiv-rekursive Grundfunktion ist oder
2. durch Substitution aus μ-rekursiven Funktionen entsteht oder
3. durch primitive Rekursion aus μ-rekursiven Funktionen entsteht oder
4. durch Anwendung der Minimalisierung aus μ-rekursiven Funktionen entsteht.

□

3.9 Theorem

Die Ackermann-Funktion ist μ-rekursiv.

◇

Zum Beweis siehe [6].

Schließlich formulieren wir noch einen Satz, der die Äquivalenz all dieser Berechnungsmodelle ausdrückt:

3.10 Theorem

Für eine n-stellige Wortfunktion $f : (\Sigma^*)^n \to \Sigma^*$ sind folgende Aussagen äquivalent:

1. f ist durch eine 1-DTM berechenbar,
2. f ist durch eine k-DTM berechenbar,
3. f ist durch eine NDTM brechenbar,
4. f ist μ-rekursiv (partiell-rekursiv),
5. f ist durch eine RAM berechenbar,

$$\vdots$$

$\diamond$

Beweis: Diese sind in mehreren Büchern der Literaturliste zu finden. Das in obigem Theorem erwähnte Maschinenmodell der Registermaschine (RAM) betrachten wir im folgenden Kapitel 4.

4 Registermaschinen

4.1 Das RAM-Modell

Turing-Maschinen sind ausgezeichnete Modelle von universellen Computern, um die grundlegenden Probleme der Berechenbarkeit, Beschreibbarkeit und Komplexitätstheorie zu untersuchen. Die im vorigen Kapitel 2 beschriebenen Automaten können alle als Erweiterungen des endlichen Automaten angeshen werden: Der Übergang von einem Zustand der endlichen Kontrolle in den nächsten ist nur möglich, wenn eine gewünschte (und daher an den Zustandsübergang gekoppelte) Operation auf dem Speichermedium durchgeführt werden kann.

In diesem Kapitel betrachten wir ein anderes grundlegendes Modell eines universellen, sequentiellen Computers. Dieses Modell ist besonders geeignet die Hauptprobleme beim Entwurf und bei der Analyse von Algorithmen zu untersuchen. Die Registermaschine (*random access machine*, *RAM*) modelliert einen sequentiellen Computer nach dem von Neumann–Prinzip. Sie ist das wichtigste Modell, um den Entwurf und die Analyse von Algorithmen für sequentielle Computer zu untersuchen.

Die Parallele RAM mit globalem Speicher, die sogenannte *PRAM* als das wichtigste Modell für den Entwurf und die Analyse von *parallelen* Algorithmen, werden wir hier nicht betrachten.

Turing–Maschinen sind ausreichend, um Fragen der Berechenbarkeit zu untersuchen. *Eine TM hat jedoch mit einem modernen Computer wenig gemeinsam.*

Das wichtigste Beispiel eines Modells realer sequentieller Computer sind **Registermaschinen**. Diese Rechnermodelle besitzen einen Speicher ähnlich dem eines modernen Mikroprozessors. Der **Speicher** wie auch das **Eingabeband** bestehen jeweils aus einer Folge von Registern, die nicht nur einzelne Symbole, sondern auch ganze Zahlen beliebiger Größe speichern können. Ein **Programm** für eine Registermaschine (RAM) ist mit einem in einer Assembler- oder höheren Programmiersprache geschriebenen Programm vergleichbar.

Bei der RAM werden zusätzlich noch zwei verschiedene Varianten unterscheiden: Die *Bit-RAM*, bei der die Register beliebige natürliche (ganze) Zahlen enthalten dürfen, und die sogenannte *arithmetische RAM*, bei der die Speicherinhalte aus einem beliebigen

Körper, wie z.B. $\mathbb{R}$ oder $\mathbb{Q}$, stammen können. Weitere Modellvarianten entstehen bei Einschränkung des erlaubten Befehlssatzes.

Das Eingabeband einer RAM besteht aus einer abzählbaren Folge von Registern $x_1, x_2, \ldots$

Der Speicher einer RAM besteht aus einer abzählbaren Folge von Registern $R_0, R_1, \ldots$. Der Index i eines Registers dient als **Adresse**. Register R_0 dient als **Akkumulator**.

Die aktuelle Konfiguration einer RAM lässt sich eindeutig aus dem Inhalt der Datenspeicher und dem Befehlszähler IC bestimmen. Am Anfang einer Berechnung sind alle Register mit 0 initialisiert.

Eine RAM wird durch ein Programm spezifiziert. Dies ist eine endliche, fortlaufend numerierte Folge $p_1, p_2, \ldots$ von Befehlen aus einer vorgegebenen Befehlsmenge $\mathcal{P}$.

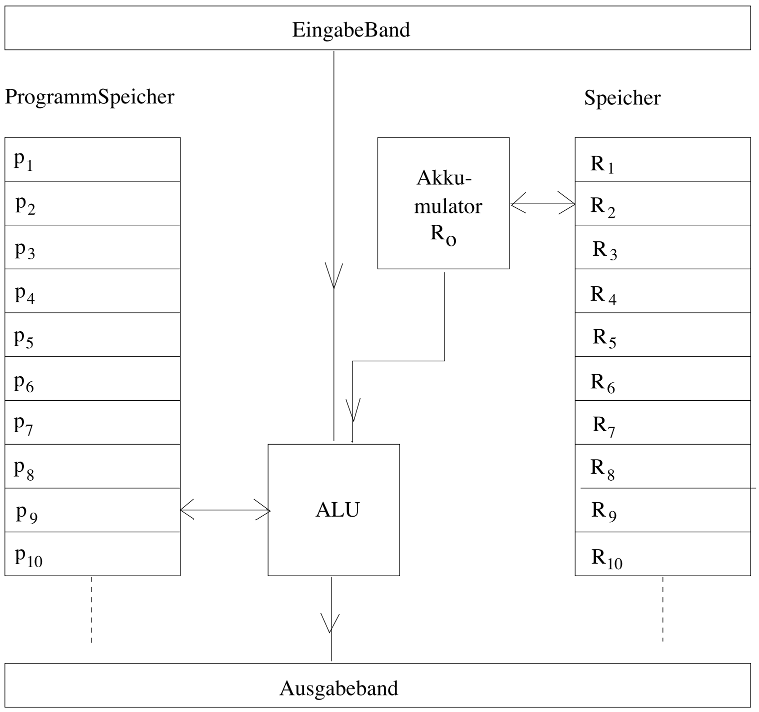


Abbildung 7: RAM – Random Access Machine

Die Befehlsmenge $\mathcal{B}$ einer RAM umfaßt die in Tabelle 1 aufgelisteten Grundoperationen. *Die exakte Natur der Befehle ist nicht entscheidend, solange die RAM–Befehle in etwa den Befehlen von realen Computern entsprechen.*

Operationen		Operand
READ	op	$= i$ – die Zahl i
WRITE	op	
LOAD	op	i – Inhalt des Registers R_i
STORE	op	
ADD	op	$*i$ – indirekte Adresse
SUB	op	(Inhalt des Registers R_j, wobei j
		der Inhalt des Registers R_i ist)
MULT	op	(nur bei RAM$_*$)
DIV	op	(nur bei RAM$_*$)
JUMP	op	
JZERO	label	
JGZERO	label	label – Befehlsadresse
JGZERO	label	

Tabelle 1: RAM-Befehle

Es gibt hier **I/O–Operationen**, **arithmetische Operationen** und **JUMP–Operationen**. Außer JUMP– und I/O–Operationen werden alle mit Hilfe des Registers R_0 (Akkumulator) ausgeführt. Tabelle 2 zeigt, wie sich die Konfiguration der Maschine durch Ausführung eines Befehls ändert.

Die Befehle des Programms werden in sequentieller Ordnung ausgeführt, bis eine Halte- oder JUMP-Instruktion vorkommt.

Im allgemeinen definiert ein RAM–Programm eine partielle Abbildung vom Eingabeband auf das Ausgabeband. Zwei Fälle sind wichtig:

1. *Berechnung von Funktionen:*
   Wenn ein RAM–Programm $\mathcal{P}$ vom Eingabeband stets n Zahlen $x_1, \ldots,$ x_n liest, dann arbeitet und danach stets m Zahlen $y_1, \ldots, y_m$ auf das Ausgabeband schreibt, dann sagt man, dass $\mathcal{P}$ eine Funktion $f : I\!N^n \longrightarrow I\!N^m$ berechnet.

   Es ist nicht schwer zu zeigen, dass RAM–Programme für Bit-RAM's jede partiell rekursive Funktion, und nur diese, berechnen können. **Beispiel:**
   In Abb. 8 gibt es ein RAM–Programm, um die Funktion $f(n) = n^n$ zu berechnen.

Eine Beschreibung desselben Programms in einer höheren Programmiersprache ist in Abb. 9 zu sehen. Aus Abb. 8 kann man auch ersehen, wie man die Befehle der höheren Programmiersprache in RAM–Instruktionen übersetzen kann.

```
           READ    1          read r1
           LOAD    1
           JGTZ    pos        if r1 ≤ 0 then write 0
           WRITE   =0
           JUMP    endif
pos:       LOAD    1          r2 ← r1
           STORE   2
           LOAD    1
           SUB     =1         r3 ← r1 − 1
           STORE   3
while:     LOAD    3
           JGTZ    continue   while r3 > 0 do
           JUMP    endwhile
continue:  LOAD    2
           MULT    1          r2 ← r2 * r1
           STORE   2
           LOAD    3
           SUB     =1         r3 ← r3 − 1
           STORE   3
           JUMP    while
endwhile:  WRITE   2          write r2
endif:     HALT
```

Abbildung 8: $f(n) = n^n$ (RAM–Programm)

2. *Akzeptieren von Sprachen:*
   Ein RAM–Programm kann man auch als einen Akzeptor einer Sprache interpretieren.

   Sei $\Sigma = \{a_1, \ldots, a_m\}$ ein Alphabet. (Das Symbol $a_i, 1 \leq i \leq m$ wird durch die natürliche Zahl i kodiert (Schreibweise: $a_i^{(k)} = i$).)

   Ein RAM–Programm $\mathcal{P}$ akzeptiert eine Sprache $\mathcal{L} \subsetneq \Sigma^*$ folgenderweise: Das Ein-

```
BEGIN
  READ r1;
    IF r1 ≤ 0 THEN WRITE 0
    ELSE
      BEGIN
        r2 ← r1;
        r3 ← r1 - 1;
        WHILE r3 > 0 DO
          BEGIN
            r2 ← r2 * r1;
            r3 ← r3 - 1
          END;
        WRITE r2
      END
END
```

Abbildung 9: $f(n) = n^n$ (Hochsprache)

gabewort $w = w_1 \dots w_k$ wird in den Feldern des Eingabebandes gespeichert, w_i als die Zahl $w_i^{(k)}$ im i–ten Feld, und im $(k+1)$-ten Feld wird 0 (als Markierung des Endes des Eingabewortes) gespeichert.

Dieses Eingabewort wird durch ein Programm $\mathcal{P}$ akzeptiert, wenn $\mathcal{P}$ zum Schluss der Berechnung 1 in das erste Feld des Ausgabebandes schreibt und hält. Die Sprache, die $\mathcal{P}$ akzeptiert ist die Menge der Wörter, die $\mathcal{P}$ akzeptiert.

Es ist nicht schwer zu zeigen, dass RAM–Programme für Bit-RAM's genau die rekursiv aufzählbaren Sprachen akzeptieren.

Beispiel:
In Abbildung 11 und 10 sind ein Hochsprache–Programm und ein RAM-Programm abgebildet, um die Sprache $\mathcal{L}_0 \subsetneq \{1,2\}^*$ zu akzeptieren, deren Wörter dieselbe Anzahl von Einsen und Zweien besitzt.

	LOAD	=0	$d \leftarrow 0$
	STORE	2	
	READ	1	read x
while:	LOAD	1	while x $\neq$ 0 do
	JZERO	endwhile	
	LOAD	1	if x $\neq$ 1
	SUB	=1	
	JZERO	one	
	LOAD	2	then d $\leftarrow$ d – 1
	SUB	=1	
	STORE	2	
	JUMP	endif	
one:	LOAD	2	else d $\leftarrow$ d + 1
	ADD	=1	
	STORE	2	
endif:	READ	1	
	JUMP	while	
endwhile:	LOAD	2	if d = 0 then write 1
	JZERO	output	
	HALT		
output:	WRITE	=1	
	HALT		

Abbildung 10: Erkennung von $\mathcal{L}_0 \subsetneq \{1,2\}^*$ (RAM)

```
BEGIN
  1 d ← 0;
  1 READ x;
  1 WHILE x ≠ 0 DO
    2 BEGIN
      3 IF x ≠ 1 THEN d ← d - 1 ELSE d ← d + 1;
      3 READ x
    2 END;
  1 IF d = 0 THEN WRITE 1
END
```

Abbildung 11: Erkennung von $\mathcal{L}_0 \subsetneq \{1,2\}^*$ (Hochsprache)

Tabelle 2: Die Befehle einer RAM

	Instruktion	Bedeutung
1.	LOAD a	$c(0) \leftarrow v(a)$
2.	STORE i	$c(i) \leftarrow c(0)$
	STORE $*i$	$c(c(i)) \leftarrow c(0)$
3.	ADD a	$c(0) \leftarrow c(0) + v(a)$
4.	SUB a	$c(0) \leftarrow c(0) - v(a)$
5.	MULT a	$c(0) \leftarrow c(0) \times v(a)$
6.	DIV a	$c(0) \leftarrow \lfloor c(0) \div v(a) \rfloor$
7.	READ i	$c(i) \leftarrow$ derzeitiges Eingabesymbol
	READ $*i$	$c(c(i)) \leftarrow$ derzeitiges Eingabesymbol. Der Kopf auf dem Eingabeband geht in beiden Fällen um ein Feld nach rechts.
8.	WRITE a	$v(a)$ wird in das Feld auf dem Ausgabeband geschrieben über dem sich gerade der Bandkopf befindet. Danach wird der Kopf um ein Feld nach rechts bewegt.
9.	JUMP b	Der *location counter* (Befehlsregister) wird auf die mit b markierte Anweisung gesetzt.
10.	JGTZ b	Der *location counter* wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) > 0$ ist, sonst auf die nachfolgende Anweisung.
11.	JZERO b	Der *location counter* wird auf die mit b markierte Anweisung gesetzt, wenn $c(0) = 0$ ist, sonst auf die nachfolgende Anweisung.
12.	HALT	Programmausführung beenden.

c – Speicherabbildung

$c(i)$ Der Inhalt des Registers R_i.

$v(a)$ – Der Wert von a

$$\begin{aligned} v(=i) &= i \\ v(i) &= c(i) \\ v(*i) &= c(c(i)) \end{aligned}$$

4.2 Komplexitätsmaße für RAMs

Es gibt zwei Arten von Komplexitätsmaßen für RAMs.

Uniforme Komplexitätsmaße:

Uniformes Zeitmaß:	Uniformes Platzmaß:
1 Schritt = 1 Zeiteinheit	1 Register = 1 Platzeinheit

Diese Maße sind einfach, aber wenn eine RAM sehr lange Zahlen verarbeitet, sind diese Maße weniger geeignet. Daher wird oft für RAMs das *logarithmische Komplexitätsmaß* verwendet. Dieser Wert ergibt sich als Summe der logarithmischen Kosten der Einzelschritte bzw. Register. Um diese Kosten zu bestimmen, benutzen wir die folgende Funktion $l : I\!N \longrightarrow I\!N$, die die Länge der Binärdarstellung der Zahlen bestimmt:

$$l(i) = \begin{cases} \lfloor \log i \rfloor + 1 & i \neq 0 \\ 1 & i = 0 \end{cases}$$

Die Kosten eines einzelnen Schrittes (RAM–Instruktion) sind in Tabelle 3 (Seite 80) aufgelistet und hängen von der Länge der Operanden ab:

Operand a	Kosten t(a)
=i	l(i)
i	l(i) + l(c(i))
∗i	l(i) + l(c(i)) + l(c(c(i)))

2.1 Definition

Die *uniforme (logarithmische) Zeitkomplexität* eines RAM–Programms ist die Summe der uniformen (logarithmischen) Kosten aller Schritte des Programms während des Programmablaufs.

Die *uniforme (logarithmische) Platzkomplexität* eines RAM–Programms ist die Summe der uniformen (logarithmischen) Kosten aller Register, die dieses Programm während des Ablaufs benutzt.

Beim logarithmischen Kostenmaß für den Platz muss allerdings noch eine, hier nicht näher beschriebene, Korrekturgröße addiert werden!

Tabelle 3: Kosten einer RAM–Instruktion

1.	LOAD a	t(a)
2.	STORE i	l(c(0)) + l(i)
	STORE $*i$	l(c(0)) + l(i) + l(c(i))
3.	ADD a	l(c(0)) + t(a)
4.	SUB a	l(c(0)) + t(a)
5.	MULT a	l(c(0)) + t(a)
6.	DIV a	l(c(0)) + t(a)
7.	READ i	l(input) + l(i)
	READ $*i$	l(input) + l(i) + l(c(i))
8.	WRITE a	t(a)
9.	JUMP b	1
10.	JGTZ b	l(c(0))
11.	JZERO b	l(c(0))
12.	HALT	1

Das *logarithmische Platzmaß* eines Registers (während eines Programmablaufs) ist die maximale Länge $l(i)$ aller Zahlen i, die in diesem Register während des Programmablaufs gespeichert wurden.

Beispiel:
Betrachten wir das RAM–Programm, das $f(n) = n^n$ berechnet (siehe Abb. 9 und 8). Es ist offensichtlich, dass für die uniformen Komplexitätsmaße die Zeitkomplexität O(n) und die Platzkomplexität O(1) ist. Die Zeitkomplexität wird durch die `while-loop` und die `MULT`–Instruktion dominiert. Diese Operation wird n–mal ausgeführt.

Bevor die `MULT`–Instruktion das i–te mal ausgeführt wird, enthält der Akkumulator n^i und das Register 1 enthält n.

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n^2 \log n)$
Platzkomplexität	$O(1)$	$O(n \log n)$

Die logarithmischen Kosten der i–ten `MULT`–Operation sind deshalb

$l(n^i) + l(n) \approx (i+1)\log n$ und die Gesamtkosten aller `MULT`–Operationen sind

$$\sum_{i=1}^{n-1}(i+1)\log n = O(n^2 \log n).$$

Beispiel:
Tabelle 4 zeigt die Komplexitätsmaße für das Programm, das die Sprache $\mathcal{L}_0$ akzeptiert.

Tabelle 4: Programm–Komplexitätsmaße für $\mathcal{L}_0$

	uniforme Kosten	logarith. Kosten
Zeitkomplexität	$O(n)$	$O(n \log n)$
Platzkomplexität	$O(1)$	$O(\log n)$

In vielen Fällen ist es notwendig, die logarithmischen Komplexitätsmaße zu benutzen, um realistische Ergebnisse zu erhalten.

Z.B. wissen wir schon, dass jede Turing–Maschine durch einen Zählerautomaten mit 2 Zählern simuliert werden kann (in denen sehr große Zahlen gespeichert sind). D.h. jede partiell rekursive Funktion kann durch ein RAM–Programm berechnet werden, dessen Platzkomplexität O(1) für das uniforme Platzkomplexitätsmaß ist! (In diesem Fall ist die uniforme Platzkomplexität offensichtlich nicht realistisch. Die logarithmische Platzkomplexität liefert viel aussagekräftigere Ergebnisse.)

Manchmal bezeichnet man die RAM mit unserer Menge von Befehlen als RAM_* und die RAM *ohne die Operation MULT und DIV* als RAM_+. Sind die in den Registern speicherbaren Zahlen stets natürliche Zahlen, so spricht man von einer *Bit-RAM.* Können rationale oder reelle Zahlen benutzt werden, so spricht man von einer *arithmetischen RAM.*

Die `MULT`–Operation ist sehr mächtig. Z.B. kann man mit n `MULT`–Operationen die Funktion $f(n) = n^{2^n}$ berechnen. Die Zeitkomplexität solcher Folge von `MULT`–Operationen ist $O(n)$ bezüglich des uniformen Komplexitätsmaßes und $O(2^n)$ – exponentiell mehr – bzgl. des logarithmischen Zeitkomplexitätsmaßes.

4.3 Simulation von bzw. mit Turing-Maschine

RAMs sind geeignete Modelle, um die Komplexität von algebraischen und kombinatorischen Optimierungs–Problemen zu untersuchen. Da RAM und TM zwei sehr verschiedene Modelle sind, entsteht die Frage, wie gut sie einander simulieren können.

4.1 Theorem

Eine $T(n)$–zeitbeschränkte Mehrband DTM M kann durch eine RAM_+ simuliert werden, die im uniformen Maß $O(T(n))$-zeitbeschränkt ist und die im logarithmischen Maß $O(T(n) \cdot \log T(n))$-zeitbeschränkt ist.

$\diamond$

Beweise finden interessierte Leser in [HUl79], [Pau78] oder in [Rei90].

4.2 Theorem

Eine im logarithmischen Maß $T(n)$–zeitbeschränkte RAM_+ (d.h. RAM ohne Multiplikation und Division) kann durch eine $O(T^2(n))$–zeitbeschränkte 5–Band DTM M simuliert werden.

$\diamond$

Man kann auch ein allgemeines Resultat zeigen: Jede im logarithmischen Maß T(n)–zeitbeschränkte RAM_* (d.h. auch mit Multiplikation und Division) kann durch eine O($T^2(n)$) zeitbeschränkte DTM simuliert werden.

Man kann für alle unsere Modelle von Turing–Maschinen, für RAMs mit logarithmischen Zeitmaß, und für RAM_+ mit uniformen Zeitmaß auch zeigen, dass jede von diesen Maschinen jede andere mit „*polynomiellem Zeitverlust und konstantem Platzverlust*“ simuliert.

Die Resultate aus diesem und aus Kapitel 8 implizieren auch, dass die folgenden zwei Komplexitätsklassen, die sehr wichtig sind, nicht von dem einzelnen Modell der erwähnten Klasse von Automaten abhängen:

$$\mathcal{P}$$

die Klasse der Sprachen (algorithmischen Probleme), die man in Polynomialzeit erkennen (lösen) kann und

$\mathcal{P}Space$

die Klasse der Sprachen (algorithmischen Probleme), die man mit polynomiell viel Platz erkennen (lösen) kann.

Es stellt sich nun auch die Frage, ob die aufzählbaren Mengen ebenfalls durch eine bestimmte Art von Grammatiken erzeugt oder generiert werden können, genauso, wie wir dies von den kontextfreien Sprachen kennen, die sich einerseits mit kontextfreien Grammatiken generieren lassen, andererseits aber ebenso von 1-Keller-Automaten akzeptiert werden können. Dies werden wir im nächsten Kapitel untersuchen.

5 Chomsky Typ-0 und Typ-1 Grammatiken

5.1 Sprachfamilien und Abschlusseigenschaften

Die Sprachklassen, die wir bisher kennengelernt hatten, waren auf unterschiedlichste Weisen definiert worden. Alle diese Verfahren haben ihre Vor- und Nachteile: endliche Ausdrücke sind knapp und leicht verständlich, Grammatiken spezifizieren mehr die Syntax, während Automaten oder Maschinen die Implementationen von Algorithmen in den Vordergrund stellen. Es ist wichtig zu wissen, mit welchen Automatenmodellen Sprachen akzeptiert werden können, die durch gewisse Typen von Grammatiken generiert werden. Ein erstes Beispiel dafür waren die hinlänglich bekannten regulären Mengen mit ihren *rationalen Ausdrücken*, den *endlichen Automaten* (NFA oder DFA, was immer am besten geeignet ist) oder den *einseitig linearen kontextfreien (Typ-3) Grammatiken*. Ein anderes Beispiel wird z.B. die Menge $\mathcal{L}_0 = \mathcal{R}e$ sein, die nach Theorem 5.5 wahlweise durch Typ-0 Grammatiken (wir werden gleich kennenlernen, was das ist) oder über Turing-Maschinen definiert werden kann.

Damit wir Möglichkeiten bekommen, bisher unbekannte und neue Klassen von formalen Sprachen einzustufen und diese mit schon bekannten zu vergleichen, benutzen wir beweisbare Eigenschaften über diese Familien von Sprachen. Der Zweig der Theoretischen Informatik, der sich damit hauptsächlich befasst, ist die sogenannte AFL-Theorie (*Abstract Families of Languages*).

5.1 Definition

Eine Menge $\mathcal{L}$ heißt Sprachfamilie genau dann, wenn folgendes gilt:

1. Es existiert eine Sprache $L \in \mathcal{L}$ mit $L \neq \emptyset$, d.h. folglich gilt auch $\mathcal{L} \neq \emptyset$.

2. Für jede Sprache $L \in \mathcal{L}$ gibt es ein endliches Alphabet Σ_L mit $L \subseteq \Sigma_L^*$.

□

Alle bisher definierten Klassen von Sprachen waren in diesem Sinne *Sprachfamilien*. Operationen, die man auf Sprachen anwendet, wie z.B. den mengentheoretischen Durchschnitt kann man zwar auch auf die Sprachfamilien anwenden, da diese ja selbst Mengen sind, jedoch ist dies im allgemeinen nicht das Ziel. Offensichtlich ergeben sich dabei wenig neue Gesichtspunkte: $\mathcal{Cf} \cap \mathcal{Reg} = \mathcal{Reg}$ ist bekannt, denn jede reguläre Menge ist auch eine kontextfreie Sprache.

Was wir eigentlich betrachten wollen, ist hier eher der Durchschnitt der Sprachen aus den beiden Familien. Man bezeichnet in der AFL Theorie dann auch korrekterweise diese Operationen auf den Sprachen als *Operatoren* auf den Sprachfamilien, um so den Unterschied deutlich zu machen. Wir wollen das hier nicht weiter ausführen und betrachten hier nur sogenannte Abschlußeigenschaften.

5.2 Definition

Eine Sprachfamilie $\mathcal{L}$ ist abgeschlossen unter einer Operation $o : 2^{\Sigma^*} \times 2^{\Sigma^*} \to 2^{\Sigma^*}$ genau dann, wenn $o(L_1, L_2) \in \mathcal{L}$, für alle $L_1, L_2 \in \mathcal{L}$ gilt.

$\square$

Beispiel: die Familie der kontextfreien Sprachen ist unter der Operation Vereinigung abgeschlossen, denn für zwei kontextfreie Sprachen L_1 und L_2 ist ihre Vereinigung $L_1 \cup L_2$ wieder eine kontextfreie Sprache.

5.2 Typ-0 oder Phrasenstruktur-Grammatik

Wenn wir unendliche Mengen definieren und behandeln wollen, brauchen wir stets endliche Beschreibungen von ihnen. Die übliche Darstellung von Mengen mit Hilfe von Prädikaten, denen die Elemente der Menge genügen sollen, ist eine häufig benutzte Weise. Für manche Wortmengen kennen wir Darstellungen über endliche Ausdrücke, wie z.B. die rationalen Ausdrücke für reguläre Mengen. Grammatiken, wie sie N. Chomsky 1956 bis 1959 einführte, sehen wir eigentlich in jeder Syntaxdefinition einer Programmiersprache. Ableitungs-Kalküle, wie sie auch in der Logik verwendet werden, stammen letzlich von Emil Post 1936 und als Ersetzungssysteme für Zeichenketten schon 1914 von Axel Thue. Die Thue-Systeme bilden den Ausgangspunkt zur Definition von kontextsensitiven Grammatiken, denn sie erweitern den Gedanken der Symbolersetzung auf ganze Zeichenketten. Gerade so, wie eine Umformung von $(a + b)^2$ zu $a^2 + 2ab + b^2$ als Ersetzung der ersten Zeichenkette durch die zweite gesehen werden kann.

Wir unterscheiden Ableitungs-Kalküle, meist formale Systeme genannt, von den oft generative Grammatiken genannten Ersetzungs-Kalkülen. Letztere stammen von den sogenannten *semi-Thue Systemen* ab und bildeten letztlich die Grundlage für die sogenannten Chomsky-Grammatiken. Daher beginnen wir mit der Definition dieser Systeme, die den eigentlichen Beginn der Theorie der formalen Sprachen bedeuteten:

5.3 Definition

Ein semi-Thue System (STS) über dem Alphabet Σ ist eine (endliche oder unendliche) Teilmenge $S \subseteq \Sigma^* \times \Sigma^*$, notiert meist als S anstelle des korrekten Tupels (Σ, S). Die Elemente $(u, v) \in S$ nennt man Produktionen und schreibt diese meistens als $u \longrightarrow v$. Die zu S gehörende einschrittige Ableitungsrelation $\underset{(S)}{\Longrightarrow} \subseteq \Sigma^* \times \Sigma^*$, ist wie folgt erklärt:

$$w_1 \underset{(S)}{\Longrightarrow} w_2 \text{ wenn } w_1 = \alpha u \beta,\ w_2 = \alpha v \beta \text{ für } \alpha, \beta \in \Sigma^* \text{ sowie } u \longrightarrow v \in S \text{ gilt.}$$

Die reflexive, transitive Hülle von $\underset{(S)}{\Longrightarrow}$ wird wie üblich mit $\overset{*}{\underset{(S)}{\Longrightarrow}}$ bezeichnet, und ist die von S definierte Ableitungsrelation. Weitere Relationen auf der Basis von $\underset{(S)}{\Longrightarrow}$ sind für $n \in I\!N$ wie folgt erklärt:

$$\overset{n}{\underset{(S)}{\Longrightarrow}} \ := \ \overset{n-1}{\underset{(S)}{\Longrightarrow}} \circ \underset{(S)}{\Longrightarrow}$$

wobei $\overset{0}{\underset{(S)}{\Longrightarrow}} := Id_S$. Dies beschreibt die Ableitungen in genau n Schritten. Den Index (S) lassen wir weg, wenn dadurch keine Verwirrung entstehen kann.

$\square$

5.4 Definition

Eine Typ-0 oder Phrasenstruktur-Grammatik wird durch ein Tupel $G := (V_N, V_T, P, S)$ spezifiziert. Hierbei gilt:

- V_N ist ein endliches Alphabet von Nonterminalen (oder auch: syntaktische Kategorie, metalinguistische Variable oder Hilfszeichen).

V_T ist endliches Alphabet von Terminalsymbolen mit $V_N \cap V_T = \emptyset$.

$V := V_N \cup V_T$ ist das Gesamtalphabet von G.

$S \in V_N$ ist das Startsymbol.

$P \subseteq V^* V_N V^* \times V^*$ ist endliche Menge von Produktionen (oder Regeln), also ein spezielles STS über V.

Eine Produktion $(u, v) \in P$ wird auch hier meist als $u \longrightarrow v$ geschrieben.

Die von G generierte oder erzeugte Sprache ist $L(G) := \{w \in V_T^* \mid S \underset{(P)}{\overset{*}{\Longrightarrow}} w\}$, und mit $\mathcal{L}_0$ wird die Familie aller von Typ-0 Grammatiken erzeugbaren Sprachen bezeichnet. D.h., $\mathcal{L}_0 := \{L \mid L = L(G) \text{ für eine Typ-0 Grammatik } G\}$.

$\square$

Das durch die Menge der Produktionen P definierte STS im Index (P) wird meist durch (G) ersetzt, wenn dies überhaupt nötig sein sollte. Eine einzelne Grammatik wird oft nur durch Angabe ihrer Produktionen notiert, wobei dann die Nonterminale stets die Großbuchstaben sind, während die Terminalsymbole mit kleinen Buchstaben abgekürzt werden.

Beispiel:
G sei gegeben durch die Produktionen:

$$S \longrightarrow abc,\ H_r b \longrightarrow bH_r,\ bH_l \longrightarrow H_l b,$$

$$S \longrightarrow aH_r bc,\ H_r c \longrightarrow H_l bcc,\ aH_l \longrightarrow aaH_r,\ aH_l \longrightarrow aa$$

Dann gilt $L(G) = \{a^n b^n c^n \mid n \geq 1\}$.

5.5 Theorem

Jede von einer Turing-Maschine akzeptierte Sprache kann auch von einer Typ-0 Grammatik generiert werden und umgekehrt, kurz: $\mathcal{L}_0 = \mathcal{R}e$.

$\diamond$

Beweis

$\mathcal{L}_0 \subseteq \mathcal{R}e$:
Zu der Typ-0 Grammatik G konstruiert man eine NTM, bei der das Eingabewort w auf eine Spur des Arbeitsbandes kopiert wird und auf einer zweiten Spur zu Beginn nur das Startsymbol S von G steht. Die NTM führt die einzelnen Ableitungsschritte auf der zweiten Spur aus, indem die linke Seite einer Produktion von G, in der dort stehenden Satzform, durch die zugehörige rechte Seite ersetzt wird. Die NTM schafft sich dabei den nötigen Platz, falls die rechte Seite der Produktion länger als die linke ist. War die rechte Seite kürzer, so werden die entstehenden freien Felder durch Zusammenschieben beider entstandenen Teile vernichtet. Die NTM prüft nach jeder Ersetzung nach, ob das Ergebnis mit der Eingabe übereinstimmt und akzeptiert bei Gleichheit. Es ist klar, das diese NTM genau $L(G)$ akzeptiert, denn wenn das Wort abgeleitet werden kann, so existiert diese Ableitung auch in einer der durchgeführten Rechnungen der NTM.

$\mathcal{R}e \subseteq \mathcal{L}_0$:
Wir nehmen an, das $L \in \mathcal{R}e$ von einer DTM $A := (Z, \Sigma, \Gamma, \delta, q_0, Z_{\text{end}})$ akzeptiert wird, die ein beidseitig unendliches Band besitzt. Die Grammatik G zu dieser DTM wird zunächst aus dem Startsymbol S jede beliebige Anfangskonfiguration $q_0 w \in Z\Sigma^*$ erzeugen und rechts folgend das Wort w nocheinmal als Kopie. Dies ist mit leichter Änderung der Grammatik für die Sprache $\{ww \mid w \in \Sigma^*\}$ möglich. Aus S wird also die Menge $\{\$Q_0 w \cent w \mid w \in \Sigma^*\}$ abgleitet. Diese und alle folgenden Satzformen werden in dem Teil, der die TM Konfigurationen beschreiben wird, entsprechend der Übergangsfunktion δ verändert. Dabei wird zu jedem Zustand $q_i \in Z$ das Nonterminal Q_i zusammen mit den linken und rechten Nachbarsymbolen ersetzt. Die Zeichen $\$, \cent \notin \Gamma$ fungieren dabei als Randmarkierungen. Jedes abgeleitete Wort, das einer Konfiguration entspricht, die einen Endzustand aus Z enthält, wird durch Löschen aller Zeichen zwischen den Randsymbolen $\cent$ und $\$$ sowie der Randsymbole selbst in ein Terminalwort überführt. Die Produktionen der Grammatik für die Änderungen der Konfigurationen sind dann folgende, wobei wir diejenigen zur Generierung der Menge $\{\$Q_0 w \cent w \mid w \in \Sigma\}$, weggelassen haben.

$$
\begin{aligned}
\forall y \in \Gamma: \quad yQ_i x &\longrightarrow Q_j yz, \text{ falls } \delta(q_i, x) = (q_j, z, L) \\
\$Q_i x &\longrightarrow \$Q_j \# z, \text{ falls } \delta(q_i, x) = (q_j, z, L) \\
Q_i x &\longrightarrow zQ_j, \text{ falls } \delta(q_i, x) = (q_j, z, R) \text{ und } x \neq \cent
\end{aligned}
$$

$$Q_i¢ \longrightarrow zQ_j¢, \text{ falls } \delta(q_i, \#) = (q_j, z, R)$$

Zum Erzeugen des akzeptierten Wortes benötigen wir noch Produktionen zum Löschen des gesamten Teils zwischen dem \$ und dem ¢ Symbol:

$$\begin{array}{rrcl} & Q_i & \longrightarrow & F, \text{ falls } q_i \in Z_{\text{end}} \\ \forall y \in \Gamma: & Fy & \longrightarrow & F, \\ \forall y \in \Gamma: & yF & \longrightarrow & F, \\ & \$F¢ & \longrightarrow & \lambda \end{array}$$

Was übrig bleibt, ist das akzeptierte Wort $w \in V_T^*$. Das Nonterminalalphabet ist erklärt durch $V_N := (\Gamma \setminus \Sigma) \cup \{Q_i \mid q_i \in Z\} \cup \{S, \$, ¢, F\}$ disjunkt vereinigt mit den Hilfszeichen, die zur Erzeugung von $\{\$Q_0 w¢w \mid w \in \Sigma^*\}$ benötigt werden. Als Terminalalphabet wird natürlich $V_T := \Sigma$ gewählt. Es gilt dann $L(G) = L(A)$ für die DTM A.

□

Diese Konstruktion würde prinzipiell auch bei einer NTM funktionieren, nur müsste dann statt der Übergangsfunktion δ entsprechend die Kantenmenge K in der Formulierung verwendet werden.

5.3 Typ-1 oder Kontextsensitive Grammatik

5.6 Definition

Eine Typ-0 Grammatik $G = (V_N, V_T, P, S)$ wird genau dann Typ-1 oder kontextsensitive Grammatik genannt, falls $u \longrightarrow v \in P$ nur dann gilt, wenn *entweder* $u = \alpha A \beta$, $v = \alpha w \beta$ mit $A \in V_N$, $w \in V^+$ und $\alpha, \beta \in V^*$ *oder* $u = S, v = \lambda$ und S kommt in keiner Produktion auf der rechten Seite vor.

Die Familie der kontextsensitiven Sprachen wird mit $\mathcal{C}s$ oder $\mathcal{L}_1$ abgekürzt, und es ist $\mathcal{L}_1 := \{L \mid L = L(G) \text{ für eine Typ-1 Grammatik } G\}$.

Bei Chomsky heißen die einseitig linearen Grammatiken Typ-3 Grammatiken, während die kontextfreien Grammatiken als Typ-2 Grammatiken bezeichnet wurden. Entsprechend wurden dann die dazugehörigen Sprachfamilien mit $\mathcal{L}_3$ und $\mathcal{L}_2$ abgekürzt.

□

In kontextsensitiven Grammatiken wird ein Nonterminal nur im Kontext der Wörter α und β ersetzt. Ist dieser nicht vorhanden, so liegt nur noch eine *kontextfreie Grammatik* vor, die bis auf die Sonderregel $S \longrightarrow \lambda$ auch noch λ-frei ist. In der ursprünglichen Definition von N. Chomsky kam die praktische Regelung für das Startsymbol mit der Produktion $S \longrightarrow \lambda \in P$ noch nicht vor.

Betrachten wir die bisher definierten Sprachfamilien, so werden allein aus den Definitionen schon einige Glieder der nachfolgenden Inklusionskette sichtbar. Es gilt $\mathcal{Reg} = \mathcal{L}_3 \subseteq \mathcal{Cf} = \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{Rec} \subseteq \mathcal{Re} = \mathcal{L}_0$, und wir werden zeigen, dass alle Inklusionen dieser sogenannten Chomsky-Hierarchie echt sind. Die Ungleichheit der Familien $\mathcal{Reg}$ und $\mathcal{Cf}$ wurde schon früher bewiesen und die echte Inklusion $\mathcal{L}_2 \subsetneq \mathcal{L}_1$ folgt aus der Tatsache, dass die nicht kontextfreie Sprache $\{a^n b^n c^n \mid n \in I\!N\}$ eine kontextsensitive Grammatik besitzt. Die Familie der entscheidbaren Mengen $\mathcal{Rec}$ wurde von Chomsky noch nicht dieser Hierarchie zugerechnet, aber wir werden sehen, dass jede kontextsensitive Sprache entscheidbar ist. Dass es aufzählbare Mengen gibt, die nicht entscheidbar sind, hatten wir in Kapitel 2 ebenfalls bewiesen, so dass lediglich noch die folgenden Ergebnisse zu beweisen sind: $\mathcal{L}_1 \subseteq \mathcal{Rec}$, $\mathcal{L}_1 \neq \mathcal{Rec}$ und $\{a^n b^n c^n \mid n \in I\!N\} \in \mathcal{L}_1$.

Bevor wir dies tun, werden wir eine praktischere Form von Grammatiken definieren, mit denen ebenfalls genau die kontextsensitiven Sprachen generiert werden können.

Eine Produktion der Form $AB \longrightarrow BA$ ist nicht kontextsensitv (es gibt keinen Kontext α und β) aber in der Wortlänge nicht verkürzend. Keine kontextsensitive Produktion (bis auf $S \longrightarrow \lambda$) ist verkürzend, und man kann zeigen, dass dies eine charakterisierende Eigenschaft für die Typ-1 Sprachen ist.

5.7 Definition

Eine Typ-0 Grammatik $G = (V_N, V_T, P, S)$ heißt <u>monoton</u>, falls $\forall u \longrightarrow v \in P : (|u| \leq |v|)$ oder ($u = S, v = \lambda$ und S kommt in keiner Produktion rechts vor).

$\mathcal{MON} := \{L \mid L = L(G)$ für eine monotone Grammatik$\}$ bezeichnet die Familie der von diesen Grammatiken erzeugbaren Sprachen.

□

5.8 Theorem

Es gilt: $\mathcal{Cs} = \mathcal{L}_1 = \mathcal{MON}$

◇

Beweis

$\mathcal{L}_1 \subseteq \mathcal{MON}$ ist offensichtlich, da alle kontextsensitiven Produktionen monoton sind.

Um $\mathcal{MON} \subseteq \mathcal{L}_1$ zu zeigen, konstruieren wir zu beliebig vorgegebener monotoner Grammatik $G := (V_N, V_T, P, S)$ die äquivalente Typ-1 Grammatik $G' := (V_N', V_T, P', S)$ mit $L(G) = L(G')$. Wir definieren dazu V_N' mit Hilfe neuer Symbole $\binom{A}{k}$ so:

$$V_N' := \left\{ \binom{A}{k} \mid A \in V \text{ und } 1 \leq k \leq |P| \right\} \cup V$$

Die neue Menge der Produktionen P' wird für jede Produktion aus P einen ganzen Satz von kontextsensitiven Produktionen enthalten, den wir für jede einzelne wie folgt beschreiben: Sei die k-te Produktion von P die Produktion $A_1A_2 \ldots A_n \longrightarrow B_1B_2 \ldots B_m$, so werden folgende neue Produktionen definiert:

$$\begin{array}{rcl}
A_1A_2 \ldots A_n & \longrightarrow & \binom{A_1}{k} A_2 \ldots A_n \\
\binom{A_1}{k} A_2 \ldots A_n & \longrightarrow & \binom{A_1}{k}\binom{A_2}{k} \ldots A_n \\
& \vdots & \\
\binom{A_1}{k}\binom{A_2}{k} \ldots \binom{A_{n-1}}{k} A_n & \longrightarrow & \binom{A_1}{k} \ldots \binom{A_n}{k}\binom{B_{n+1}}{k} \ldots \binom{B_m}{k} \\
\binom{A_1}{k} \ldots \binom{A_n}{k}\binom{B_{n+1}}{k} \ldots \binom{B_m}{k} & \longrightarrow & B_1 \binom{A_2}{k} \ldots \binom{B_m}{k} \\
& \vdots & \\
B_1B_2 \ldots \binom{B_m}{k} & \longrightarrow & B_1B_2 \ldots B_m
\end{array}$$

Da die Nonterminalzeichen von V_N' für je zwei Produktionen von P alle disjunkt sind, kann eine einmal begonnene Ableitung von solch einem zusammengehörenden Regelsatz aus P' nur erfolgreich terminieren, wenn er ganz zu Ende ausgeführt wird. Einen noch formaleren Beweis mögen ungläubige Leser(innen) selbst ausführen.

□

5.4 Kontextsensitivität und Entscheidbarkeit

5.9 Theorem

Jede kontextsensitive Sprache ist entscheidbar, d.h. $\mathcal{L}_1 \subseteq \mathcal{R}ec$.

◇

Beweis

Für eine Sprache $L \in \mathcal{L}_1$ sei $L = L(G)$ für eine monotone Grammatik G, was nach Theorem 5.8 möglich ist. Um nun zu entscheiden ob ein beliebiges Wort w in L vorkommt, betrachten wir alle möglichen Ableitungen $S \overset{*}{\underset{(G)}{\Longrightarrow}} v$ mit $|v| \leq |w|$. Da es davon stets nur endlich viele ohne Wiederholungen einer Satzform gibt, und weil auch jede darin vorkommende Satzform nicht länger als das Wort w sein kann, kann dieses mit einer TM leicht geprüft werden.

□

Die Frage, ob nun vielleicht umgekehrt jede entscheidbare Menge auch kontextsensitiv ist, kann wieder mit einem Diagonalbeweis negativ entschieden werden.

5.10 Theorem

Es gilt: $\mathcal{L}_1 \subsetneq \mathcal{R}ec$

◇

Beweis

Wir konstruieren die entscheidbare Menge $L_e \notin \mathcal{L}_1$: Sei $\langle G_1 \rangle, \langle G_2 \rangle, \dots$ eine Aufzählung aller monotonen Grammatiken. Dies ist möglich, weil für jedes Wort über dem Kodierungs-Alphabet $\{0,1\}$ entschieden werden kann, ob es die Gödelisierung einer monotonen Grammatik ist.

Weiter sei $f : \{0,1\}^* \to \mathbb{N}$ eine berechenbare Bijektion, die jedem Wort w die Zahl $f(w) = i$ zuordnet, wenn w das i-te Wort in der lexikalischen Ordnung auf $\{0,1\}^*$ ist. Nun definieren wir:

$$L_e := \{w \mid w \notin L(G_{f(w)})\}$$

Wir beobachten, dass L_e entscheidbar ist: Für ein beliebiges Wort $v \in \{0,1\}^*$ berechnet man $f(v)$, konstruiert dann die monotone Grammatik $G_{f(v)}$ und testet wie im Beweis zu Theorem 5.9, ob $v \notin L(G_{f(v)})$ gilt.

$L_e \notin \mathcal{L}_1$ folgt nun durch Widerspruchsbeweis: Falls $L_e \in \mathcal{L}_1$ wäre, so gäbe es eine monotone Grammatik G_n mit $L(G_n) = L_e$. Für das n-te Wort u der lexikalischen Aufzählung von $\{0,1\}^*$ mit $f(u) = n$ gilt nun *entweder* $u \in L_e$ *oder* $u \notin L_e$. In beiden Fällen entsteht ein Widerspruch wie folgt:

$$u \in L_e \stackrel{\text{Def. } L_e}{\longrightarrow} u \notin L(G_n) \stackrel{\text{Annahme}}{\longrightarrow} u \notin L_e$$

Andererseits ergibt sich auch:

$$u \notin L_e \stackrel{\text{Def. } L_e}{\longrightarrow} u \in L(G_n) \stackrel{\text{Annahme}}{\longrightarrow} u \in L_e$$

Mithin der Widerspruch zur Annahme $L_e \in \mathcal{L}_1$.

□

Nachdem wir sahen, dass die Typ-0 Sprachen von allgemeinen Turing-Maschinen akzeptiert werden, stellt sich die Frage, wie die Maschinen aussehen könnten, die zu den kontextsensitiven Sprachen gehören. Natürlich kann jede kontextsensitiven Sprache auch von einer Turing-Maschine akzeptiert werden. Da Turing-Maschinen jedoch Sprachen akzeptieren können, die nicht mehr kontextsensitiv sind, muss deren Leistungsfähigkeit passend eingeschränkt werden!

5.5 Linear beschränkte Automaten

5.11 Definition

Ein linear beschränkter Automat (LBA) ist eine NTM, die bei beliebiger Eingabe w auf dem Arbeitsband höchstens $c \cdot |w|$ Felder bis zur Akzeptierung besucht. Dabei ist $c \in \mathbb{R}^+$ eine Konstante, die nicht von w abhängt. Arbeitet die Turing-Maschine bei gleicher Beschränkung ihres Arbeitsbandes deterministisch, so wird der linear beschränkte Automat mit DLBA abgekürzt.

Die Familie der von LBA's bzw. DLBA's akzeptierten Sprachen wird mit $\mathcal{LBA}$ bzw. $\mathcal{DLBA}$ bezeichnet.

□

5.12 Theorem

Es gilt: $\mathcal{LBA} = \mathcal{L}_1$

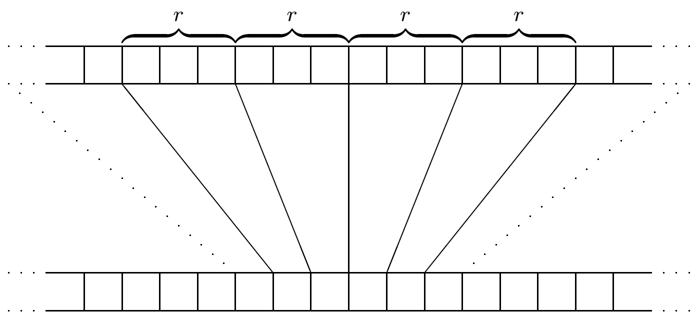
◇

Beweis

Da $\mathcal{L}_1 = \mathcal{MON}$ ist, zeigen wir zunächst $\mathcal{MON} \subseteq \mathcal{LBA}$:

Die Konstruktion im Beweis von Theorem 5.5 für eine NTM, die eine Typ-0 Sprache akzeptiert, wird auch hier durchgeführt. Weil die vorgegebene Grammatik monoton war, muß diese NTM lediglich jedesmal prüfen, ob die Länge der Satzform auf der zweiten Spur länger wird als das Eingabewort. In diesem Fall darf die NTM nicht akzeptieren. Daher ist diese leicht modifizierte NTM tatsächlich ein LBA. Für jedes Wort v das auf der zweiten Spur erzeugt wird und dessen Länge gleich der von w ist, prüft dieser LBA nach, ob $v = w$ gilt. Tritt dieser Fall ein, so akzeptiert der LBA das Wort w. Also akzeptiert der aus der monotonen Grammatik konstruierte LBA die gesamte Sprache L.

Für den Beweis von $\mathcal{LBA} \subseteq \mathcal{MON} = \mathcal{L}_1$ konstruieren wir zu einem beliebigen LBA A, der mit Platzbeschränkung $c \cdot |w|$ arbeitet, eine monotone Grammatik. Falls nun die Konstante c größer als 1 ist, können wir die Inschrift des Arbeitsbandes des LBA nicht als Satzformen beim Ableiten verwenden. Die Konstruktion im Beweis von Theorem 5.5 ist leider auch nicht zu verwenden, weil diese Grammatik nicht monoton ist. Wir zeigen also zunächst, dass jeder LBA statt mit $c \cdot |w|$ stets auch nur mit genau $|w|$ Arbeitsfeldern auskommen kann. Sei also der LBA A wie oben so, dass er mit $c \cdot |w|$ benutzten Arbeitsfeldern akzeptiert. Ist $0 < c \leq 1$, so ist nichts zu beweisen. Für $c > 1$ konstruieren wir den LBA B, der mit $|w|$ Platz auskommt und auch $L = L(A)$ akzeptiert. B hat als Bandalphabet die Menge Y^r, wobei Y das entsprechende Alphabet von A war und r eine noch festzulegende natürliche Zahl sein wird. Jedes Element aus Y^r stellt einen *Block* von r, auf dem Arbeitsband von A nebeneinander liegenden, Zeichen dar. Bewegt sich der LSK von A in solch einem Block, so kann dies der LBA B ohne Kopfbewegung nur innerhalb seiner endlichen Kontrolle tun.



B braucht auf diese Weise nur $\frac{c}{r} \cdot |w|$ Felder seines Arbeitsbandes zu betreten. Dass das Bandalphabet von B so viel größer werden mußte, ist dabei unerheblich. Wählt man $r := \min(n \in I\!N \mid n \geq c)$, so sind das höchstens $|w|$ Felder.

Zu dem nichtdeterministischen LBA $B = (Z, \Sigma, \Gamma, K, q_0, Z_{\text{end}})$ konstruiert man nun eine Grammatik $G := (V_N, V_T, P, S)$, bei der fast jedes Nonterminalsymbol vier Spuren besitzt und die wesentlichen Satzformen alle aus $|w|$ Zeichen bestehen werden. Die Anfangs-Konfiguration $q_0 w$ für ein beliebiges Wort $w \in \Sigma^*$ wird von S aus mit folgenden Produktionen *erzeugt*:

$$S \longrightarrow A \begin{bmatrix} x \\ x \\ \cent \\ \# \end{bmatrix}, \; A \longrightarrow A \begin{bmatrix} x \\ x \\ \# \\ \# \end{bmatrix} \text{ und } A \longrightarrow \begin{bmatrix} x \\ x \\ \$ \\ q_0 \end{bmatrix}, \text{ für alle } x \in \Sigma$$

Dabei markieren \$ und ¢ den linken bzw. den rechten Rand der Eingabe $w \in \Sigma^*$, die in den ersten beiden Spuren der Nonterminale steht. Das Symbol für den Zustand des LBA's steht immer in der letzten Spur desjenigen Zeichens x, das der LSK von B gerade besucht. Die anderen Produktionen beschreiben dann die Übergänge der Konfigurationen anhand der Maschinen-Tafel K. Für alle Symbole $x_1, x_2, x_3 \in \Sigma$, $y_1, y_2 \in \Gamma$ werden für jedes Tupel $(p, y, z, L, q) \in K$ – bzw. jedes $(p, y, z, R, q) \in K$ – folgende Produktionen in G aufgenommen:

$$\begin{bmatrix} x_1 \\ y_1 \\ \# \\ \# \end{bmatrix} \begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix} \longrightarrow \begin{bmatrix} x_1 \\ y_1 \\ \# \\ q \end{bmatrix} \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix}, \; \begin{bmatrix} x_1 \\ y_1 \\ \$ \\ \# \end{bmatrix} \begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix} \longrightarrow \begin{bmatrix} x_1 \\ y_1 \\ \$ \\ q \end{bmatrix} \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix} \text{ sowie}$$

$$\begin{bmatrix} x_1 \\ y_1 \\ \$ \\ \# \end{bmatrix}\begin{bmatrix} x_2 \\ y \\ ¢ \\ p \end{bmatrix} \longrightarrow \begin{bmatrix} x_1 \\ y_1 \\ \$ \\ q \end{bmatrix}\begin{bmatrix} x_2 \\ z \\ ¢ \\ \# \end{bmatrix}$$

Für die Bewegungen nach rechts lauten die Produktionen ganz entsprechend:

$$\begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix}\begin{bmatrix} x_3 \\ y_2 \\ \# \\ \# \end{bmatrix} \longrightarrow \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix}\begin{bmatrix} x_3 \\ y_2 \\ \# \\ q \end{bmatrix}, \begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix}\begin{bmatrix} x_3 \\ y_2 \\ ¢ \\ \# \end{bmatrix} \longrightarrow \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix}\begin{bmatrix} x_3 \\ y_2 \\ ¢ \\ q \end{bmatrix} \text{ sowie}$$

$$\begin{bmatrix} x_2 \\ y \\ \$ \\ p \end{bmatrix}\begin{bmatrix} x_3 \\ y_2 \\ ¢ \\ \# \end{bmatrix} \longrightarrow \begin{bmatrix} x_2 \\ z \\ \$ \\ \# \end{bmatrix}\begin{bmatrix} x_3 \\ y_2 \\ ¢ \\ q \end{bmatrix}$$

Zum Schluß müssen alle Nonterminale genau dann in das entsprechende Terminalsymbol überführt werden, wenn ein Endzustand bei der Berechnung des LBA erreicht wurde.

Die Produktionen dafür sind für alle $x, z \in \Sigma$, für $y \in \Gamma$ und $\& \in \{\#, \$, ¢\}$ wie folgt erklärt:

$$\begin{bmatrix} x \\ y \\ \& \\ q \end{bmatrix} \longrightarrow x \text{ falls } q \in T \text{ und } \begin{bmatrix} x \\ y \\ \& \\ \# \end{bmatrix} z \longrightarrow xz \text{ bzw. } z \begin{bmatrix} x \\ y \\ \& \\ \# \end{bmatrix} \longrightarrow zx$$

Mit diesen Produktionen kann also ein terminales Wort genau dann erzeugt werden, wenn es eine Erfolgsrechnung für w im LBA mit $|w|$ Speicherplatz gibt.

□

Mit Hilfe von Charakterisierungen von Sprachklassen durch Automaten, lassen sich in vielen Fällen einige Abschlußeigenschaften der Sprachfamilien leichter beweisen als mit Grammatiken. Dies ist zum Beispiel beim Beweis des folgenden Theorems zu sehen:

5.13 Theorem

Die Familie der kontextsensitiven Sprachen $\mathcal{L}_1 = \mathcal{LBA}$ ist gegenüber Durchschnittsbildung abgeschlossen.

$\diamond$

Beweis

Seien A und B zwei LBA's für $L_A := L(A)$ und $L_B := L(B)$. Man kann ohne größere Schwierigkeiten einen LBA C konstruieren, der A auf einer Spur des Arbeitsbandes und B auf einer zweiten Spur simuliert. C akzeptiert das Wort w genau dann, wenn w von A und B akzeptiert wird. Der Platzbedarf von C übersteigt den von A und B nicht.

$\square$

Die Beweise für die Abgeschlossenheit von $\mathcal{L}_1$ gegenüber *Vereinigung* oder *nicht-löschendem Homomorphismus*, sind natürlich einfacher mit Grammatiken zu zeigen. Dass auch das Komplement einer kontextsensitiven Sprache wieder eine kontextsensitive Sprache ist, wurde 1987 von N. Immerman (Yale Univ.) und – unabhängig davon – von R. Szelépcsenyi (ČSSR) bewiesen und ist daher in einigen Lehrbüchern noch nicht enthalten. Im Beweis werden platzbeschränkte Turing-Maschinen verwendet und das Resultat ist zudem noch allgemeiner formuliert (siehe dazu [Weg93]).

6 Algorithmen-Techniken

Wenn wir einen Algorithmus zu einer Problemlösung programmieren sollen, so ist zunächst die Lösung zu finden und dann der Entwurf für seine Umsetzung in einem Computer-Programm (Implementierung) zu erstellen. Diese Lösung ist auf Korrektheit und Effizienz hin zu überprüfen. Das Kodieren in einer Programmiersprache steht erst am Ende dieses Vorgehens. Wenn eine (mathematische) Lösung des Problems gefunden wurde, so ist die Frage nach der besten Methode zu deren algorithmischen Umsetzung eine Herausforderung, für die leider keine allgemeingültigen Regeln formuliert werden können. Dennoch gibt es unterschiedliche Techniken, die sich in verschiedensten Bereichen bewährt haben. Die Wichtigsten wollen wir in diesem Abschnitt an typischen Beispielen illustrieren. Wenn Sie sich mit diesen vertraut gemacht haben, können Sie prüfen, welche sich bei Ihrem Algorithmen-Design Problem am besten anwenden lässt. Die mathematischen Analysemethoden verschieben wir jedoch auf Kapitel 7

6.1 Rekursiv oder iterativ?

Aus der Mathematik kennen wir viele rekursive Definitionen. In diesem Zusammenhang wird oft die Fakultätsfunktion $n!$ zitiert, die für natürliche Zahlen $n \in I\!N$ rekursiv, d.h. „durch Rückgriff auf sich selbst“ definiert werden kann.

Rekursive Programme werden abgearbeitet, indem sie sich solange selbst aufrufen, bis ein Stopkriterium erreicht wird. Auch die Binomialkoeffizienten lassen sich mit Hilfe des Pascal'schen Dreiecks rekursiv erklären. Viele Algorithmen lassen sich durch die Benutzung von rekursiven Verfahren oft einfach formulieren, aber deren Umsetzung in Programme muss dieser Idee nicht unbedingt folgen. Betrachten Sie die Fakultätsfunktion in ihrer rekursiven Definition:

$$fac(n) := \begin{cases} 1 & \text{, für } n = 0 \\ n \cdot fac(n-1) & \text{, sonst.} \end{cases}$$

Direkt als rekursives Programm (in einer fiktiven Programmiersprache) geschrieben könnte dies so formuliert werden:

```
FUNCTION fac(n)
  BEGIN
    IF n = 0
      THEN
        RETURN 1
      ELSE
        RETURN n·fac(n-1)
  END.
```

Abbildung 12: $fac(n) := n!$ (Hochsprache, rekursiv programmiert).

Aber die gewöhnliche Definition von $n!$ als das Produkt aller natürlichen Zahlen von 1 bis n würde eine andere, iterative Lösung suggerieren:

```
FUNCTION fac(n)
  BEGIN
    DECL prod INTEGER;
      prod ⟵ 1;
        FOR i FROM 1 TO n DO
        prod ⟵ prod·i
    RETURN prod
  END.
```

Abbildung 13: $fac(n) = n!$ (Hochsprache, iterativ programmiert).

Welche Methode soll nun verwendet werden? Dies Frage kann immer nur mit Blick auf das Problem entschieden werden.

Beide Methoden haben nun leider einen gravierenden Nachteil: Wenn wir real existierende Computer betrachen, so ist deren Darstellung von ganzen Zahlen (Integer) oft auf 32 Bit beschränkt. Bereits ab $13! = 6227020800$ kann dann das Ergebnis nicht mehr dargestellt werden! Die beste Methode ist, die Werte von $n!$ für $n = 0$ bis etwa $n = 13$ in einer Tabelle abzuspeichern, und die Werte bei Bedarf daraus zu entnehmen! Das geht etwa 50 mal schneller als eine programmierte Berechnung!

Im Fall der Fakultätsfunktion ist die Antwort auf obige Frage einfach: diese Funktion sollte gar nicht programmiert, sondern nur durch Tabellennachschau implementiert werden!

Natürlich gibt es Beispiele, die tatsächlich rekursiv gelöst werden, und dann auch effektiv sind.

Das sogenannte **Binäre Suchen** ist weitläufig bekannt: Auf den Objekten des Suchraums, hier einer Menge A, muss eine lineare Ordnung definiert sein. Wenn ein Element x in A gesucht wird, so wird zunächst geprüft, ob vielleicht schon $A = \{x\}$ (`Antwort: x in A vorhanden`), oder $|A| = 1$ aber $A \neq \{x\}$ (`Antwort: x nicht in A vorhanden`) gilt. Andernfalls partitioniert man den Suchraum in zwei möglichst gleich große Teile A_1 und A_2, d.h.: $A = A_1 \uplus A_2$ mit $|A_1| = \lceil \frac{1}{2} |A| \rceil$ und $|A_2| = \lfloor \frac{1}{2} |A| \rfloor$, sowie $max(A_1) < min(A_2)$. Dann wird festgestellt, welcher der unten formulierten drei Fälle (3. bis 5.) vorliegt und entsprechend weiter verfahren. Insgesamt könnte das Verfahren also so formuliert werden:

1. $|A| = 1$ und $A \neq \{x\}$: (`Antwort: x nicht in A vorhanden`)
2. $A = \{x\}$: (`Antwort: x in A vorhanden`)
3. $x \leq max(A_1)$: Suche x in A_1 mit dem gleichen Verfahren,
4. $x \geq min(A_2)$: Suche x in A_2 mit dem gleichen Verfahren
5. $max(A_1) < x < min(A_2)$: (`Antwort: x nicht in A vorhanden`)

Da jedesmal höchstens mit einem ungefähr halbiertem Suchraum fortgefahren wird bis das Element gefunden wurde arbeitet das Verfahren schlimmstenfalls mit $O(\log n)$ Schritten[7], so dass in einem Feld mit einer Millionen Zahlen höchstens 20 Schritte nötig sind um ein bestimmtes Element zu finden.

Oft nutzen kompliziertere Verfahren die Rekursive Struktur zusammen mit der Methode des sogenannten "Divide and Conquer".

6.2 Divide and Conquer

Das lateinische Zitat "divide et impera" (teile und herrsche) geht vermutlich nicht auf Julius Cäsar zurück, sondern wohl auf den französischen König Ludwig den XI (15.

[7] Ein vollständiger Binärbaum mit n Blättern hat eine Tiefe (oder Höhe) von $\log_2 n$. Für Binärbäume t mit der Eigenschaft, dass jeder Knoten zwei Teilbäume hat, deren Blätterzahl sich höchstens um Eins unterscheiden gilt: $\mathit{Tiefe}(t) = \lceil \log_2 n \rceil$. Finden Sie den Beweis dafür?

Jahrhundert), der damit eher meinte: entzweie und herrsche! Das Divide and Conquer Verfahren arbeitet im Wesentlichen rekursiv und nützt genau dieses aus: Das zunächst große Problem wird in kleinere und damit leichter angreifbare Teile geteilt. Jeder der kleineren Teile wird nun separat nach der gleichen Methode (!) gelöst und die Ergebnisse dann wieder passend zusammengesetzt.

Wir demonstrieren dies zunächst an Beispielen:

Beispiel 1: Mergesort:

Gegeben: eine Folge von Elementen aus einer geordneten Menge: $a = (a_1, a_2, \ldots, a_n)$.

Gesucht: eine Folge $\bar{a} = (a_{i_1}, a_{i_2}, \ldots, a_{i_n})$ mit den gleichen Elementen und mit

$$a_{i_1} \leq a_{i_2} \leq a_{i_3} \leq \ldots \leq a_{i_n}.$$

Algorithmus:

1. Zerlegung von a in zwei (fast) gleichgroße Teilfolgen:

$$a' = (a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}),$$
$$a'' = (a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n).$$

2. Sortieren von a' und a'' mit dem gleichen Verfahren zu den Ergebnissen: $\bar{a}', \bar{a}''$.

3. Mischen der sortierten Folgen $\bar{a}', \bar{a}''$ zu einer sortierten Folge $\bar{a}$.

Abschätzung der Zeitkomplexität für den Algorithmus MERGESORT:

Die Zeitkomplexität setzt sich zusammen aus der Anzahl der einzelnen Arbeitschritte in 1.), 2.) und 3.):

1.) benötigt nur einen Schritt,
3.) benötigt $O(n)$ Schritte,
2.) wenn $T_A(n)$ die Zeitkomplexitätsfunktion von MERGESORT ist,
so benötigt das Sortieren der Teilfolgen $\bar{a}'$ und $\bar{a}''$ jeweils $T_A\left(\frac{n}{2}\right)$.

Die Zeitkomplexitätsfunktion für MERGESORT ergibt sich dann aus der Rekursionsgleichung:

$$T_A(n) = 2 * T_A\left(\frac{n}{2}\right) + O(n),$$

mit der Lösung:

$$T_A(n) = O(n \log n).$$

Dies Ergebnis werden wir auf Seite 140 aus Kapitel 7 auch formal mit dem allgemeinen Analyseergebnis zu *devide-and-conquer* Verfahren beweisen.

Beispiel 2: Problem des dichtesten Punktepaares:
(ein Problem aus der algorithmischen Geometrie - computational geometry)

Gegeben: eine Menge P von n Punkten in der Ebene: $p_i = (x_i, y_i), \quad i = 1, 2, \ldots, n.$
Die

Gesucht: zwei Punkte minimaler Distanz.
Distanz (Entfernung) zweier Punkte ist wie folgt definiert:

$$d(p_i, p_k) := \sqrt{(x_i - x_k)^2 + (y_i - y_k)^2}$$

Die Distanz wird in der euklischen Metrik angegeben.

Wir betrachten hier nur den einfachsten Spezialfall:

Eindimensionaler Fall:

Gegeben: Eine Menge P von n Punkten auf einer Geraden:

$$\begin{aligned} P &= \{p_1, p_2, \ldots, p_n\} \\ &= \{x_1, x_2, \ldots, x_n\}, \quad \textit{wobei } x_i \textit{ dieAbszissevon } p_i \textit{ ist.} \end{aligned}$$

Gesucht: zwei Punkte minimaler Distanz.

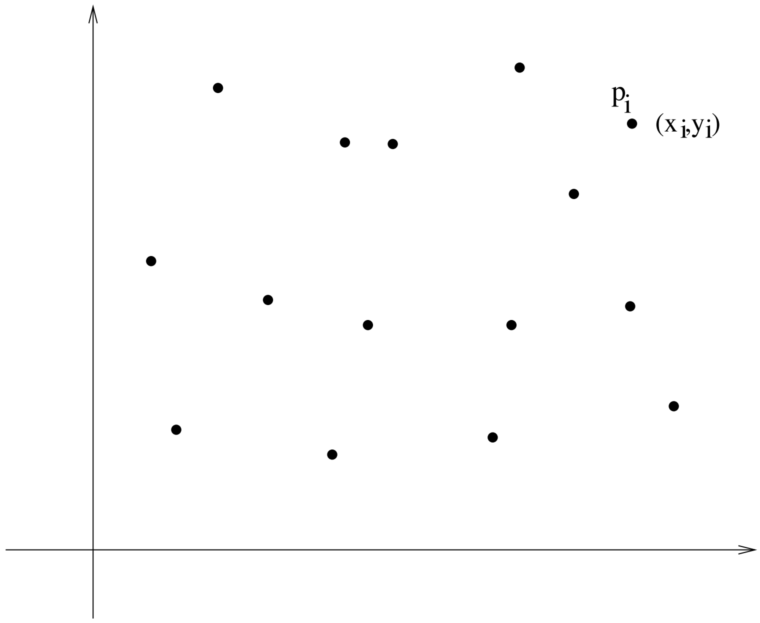


Abbildung 14: Menge von n Punkten in der Ebene

Algorithmus dichtestes Punktepaar auf einer Linie:

1. Zerlegung von P in P_1 und P_2 durch einen Trennungspunkt $x = \alpha$ ($\alpha \notin P$), so dass $x_i \in P_1 \Leftrightarrow x_i < \alpha$ und $x_j \in P_2 \Leftrightarrow x_j > \alpha$.

2. Bestimmung des dichtesten Paares in P_1 und des dichtesten Paares in P_2:
   Rekursion:

   > δ:= minimale Distanz der beiden Paare, die in P_1 und P_2 gefunden wuden.

3. Sei x^1_{max} größtes Element in P_1 und x^2_{min} kleinstes Element in P_2 :
   Wir bilden:
   $$\delta' := \mid x^2_{min} - x^1_{max} \mid .$$

4. $\delta^* := \min\{\delta, \delta'\}$ ist der gesuchte dichteste Abstand von Punkten in P.

Erst an dieser letzten Stelle, wo die Distanz der Punkte nahe an der Trennungslinie zwischen verschiedenen Gebieten gefunden und berücksichtigt werden muss, wird die Verallgemeinerung dieser Idee auf den mehrdimensionalen Fall kompliziert!

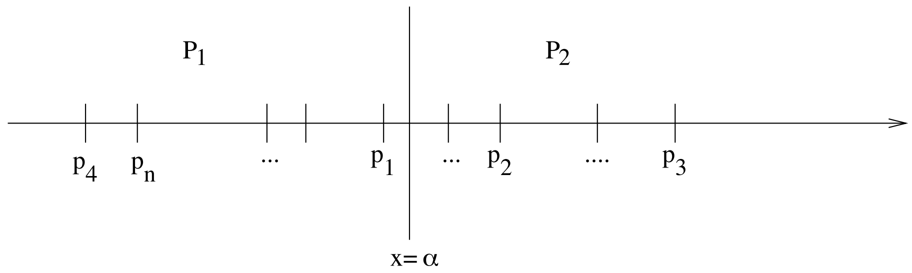


Abbildung 15: n Punkte auf einer Geraden

Beispiel 3: ***Multi Precision Multiplication*:**
(ein Problem aus dem Hardware-Software Design)

Die Aufgabe ist es, bei Ausnutzung der vorhandenen Hardware-Möglichkeiten einer 32-Bit Multiplikation eine präzise Multiplikation von 64-Bit, 128-Bit oder größeren Zahlen zu erreichen. Seien $x, y \in \{0,1\}^n$ zwei gleichlange Binärdarstellungen der Zahlen $[x]_2$, und $[y]_2$ deren Produkt p gesucht wird. Wenn n sehr groß und gerade ist, so können wir die Darstellungen x und y halbieren zu $x = x_1x_2$ und $y = y_1y_2$ mit $|x_1| = \frac{1}{2}|x|$ und $|y_1| = \frac{1}{2}|y|$. Im Folgenden schreiben wir zur leichteren Lesbarkeit statt $[x]_2$ und $[y]_2$ kurz $[x]$, bzw. $[y]$.

Entsprechend der Arithmetik von Binärzahlen gilt nun:

$$\begin{aligned} p = [x] \cdot [y] &= \Big([x_1] \cdot 2^{\frac{n}{2}} + [x_2]\Big) \cdot \Big([y_1] \cdot 2^{\frac{n}{2}} + [y_2]\Big) \\ &= [x_1] \cdot [y_1] \cdot 2^n + ([x_1] \cdot [y_2] + [x_2] \cdot [y_1]) \cdot 2^{\frac{n}{2}} + [x_2] \cdot [y_2] \qquad (12) \end{aligned}$$

In Gleichung 12 sehen wir vier Multiplikationen, jedoch fanden 1962/63 Karatsuba und Ofmann, wie man darin den Faktor vor $2^{\frac{n}{2}}$ mit nur 3 Multiplikationen Darstellen kann:

$$([x_1] \cdot [y_2] + [x_2] \cdot [y_1]) = ([x_1] + [x_2]) \cdot ([y_1] + [y_2]) - ([x_1] \cdot [y_1] + [x_2] \cdot [y_2]) \quad (13)$$

In Gleichung 13 werden nur 3 Multiplikationen benutzt, und die in Gleichung 12 weiterhin verwendeten Multiplikationen $[x_1] \cdot [y_1]$ und $[x_2] \cdot [y_2]$ brauchen nicht erneut errechnet werden! Daher reichen sogar für das gesamte Produkt $p = [x] \cdot [y]$ insgesamt drei Multiplikationen plus einige Additionen, Zwischenspeicherungen und das kostenfreie Anhängen von n (oder $\frac{n}{2}$) Nullen um die Multiplikationen mit 2^n (bzw. $2^{\frac{n}{2}}$) einfach zu bewerkstelligen.

```
Eingabe sind Binärdarstellungen x = x1x2 und y = y1y2 mit
|x1| = |x2| = |y1| = |y2|.

 PRODUKT (x, y)
   DECL s1, s2, p1, p2, p3, t, u1, u2 INTEGER, q1, q2, v BITSTRING;
     s1 ⟵ [x1] + [x2];   s2 ⟵ [y1] + [y2];
     p1 ⟵ [x1] + [y2];   p2 ⟵ [x2] + [y1];
     p3 ⟵ s1 · s2;   t ⟵ p1 + p2;
     u2 ⟵ u · 2^(n/2); (⋆ kostenfreies Anhängen von Nullen ⋆)
      v ⟵ q1q2  (⋆ kostenfreie Konkatenation der Binär-
                   darstellungen von p1 = [q1] und p2 = [q2],
                   mit dann [v] = [x1] · [y1] · 2^n + [x2] · [y2].  ⋆)
   RETURN p ⟵ [v] + u2
```

Abbildung 16: *Multi Precision Multiplication*, rekursiv anzuwenden.

Mit einer *divide-and-conquer*-Technik erhält man ein Multiplikationsverfahren, welches die Multiplikation von Binär-Zahlen der Länge n (n sehr groß, also extrem große Zahlen der Werte 2^n) mit $O(n^{\log 3}) = O(n^{1{,}585})$ Multiplikationen von Bitzahlen einer festen Länge (z.B. 32 Bits) brechnet. Damit die Leser nicht lange selber knobeln müssen, hier der Basis-Algorithmus in einer Programmiersprachen nahen Form (siehe Abbildung 16).

In einer Implementation sind natürlich alle Zahlen nur über ihre Binärdarstellungen auszudrücken und nicht etwa in andere Darstellungen umzurechnen!

Bei Ausnutzung aller Parallelisierbarkeiten kann so leicht ein Verfahren gefunden werden, mit dem die Multiplikation von Binärzahlen der Länge n in einer Zeit der Größenordnung $\theta(\log n)$ erhalten werden kann.

Wir werden in Kapitel 7 noch weitere Verfahren kennen und analysieren lernen, die die überragende Stärke der *divide-and-conquer*-Technik aufzeigt.

6.3 Greedy-Algorithmen

Greedy (=gierige) Algorithmen werden meist bei Problemen eingesetzt, bei denen es darum geht, aus einer Menge von Objekten eine Teilmenge mit bestimmten Eigenschaften auszuwählen. Greedy-Algorithmen wählen aus der Eingabemenge sukzessive Elemente heraus und entscheiden ad hoc, ob diese zu der Lösungsmenge gehören sollen. Beide Entscheidungen sind endgültig und können nicht wieder rückgängig gemacht werden! Typischer Weise arbeiten solche Algorithmen nach dem folgenden Schema:

```
LÖSUNG ⟵ ∅
 SOLANGE M ≠ ∅ TUE
   x ⟵ ein aus der Eingabemenge M entferntes Element;
     WENN x zu der Lösungsmenge passt, dann
     LÖSUNG ⟵ LÖSUNG ∪ {x}
```

Abbildung 17: Rahmenverfahren der Greedy-Algorithmen

Greedy-Algorithmen suchen in der Regel nur lokal optimale Lösungen, die aber bisweilen auch global die besten sind.

Ein gutes Beispiel, bei dem ein Greedy-Algorithmus zum Erfolg führt, ist die Suche nach einem aufspannenden Baum mit minimalem Gewicht (auch als minimales Gerüst bezeichnet) in einem ungerichteten, und kantengewichteten Graphen.

Beispiel 4: Minimales Gerüst:

Gegeben: Ein Graph $G = (V, E)$, ungerichtet, zusammenhängend,
$l : E \to I\!N$ eine Kanten-Bewertung, wobei $l(e) =$ die "Länge" der Kante $e \in E$) bezeichnet,
$V = \{v_1, v_2, \ldots, v_n\}$,
$E = \{e_1, e_1, \ldots, e_m\}$.

Gesucht: Ein Gerüst T von G mit minimaler Länge.

Ein *Gerüst* von $G = (V, E)$ ist ein Teilgraph $T = (V, F)$ von $G = (V, E)$, der alle Knotenpunkte von G enthält, zusammenhängend ist und eine minimale Anzahl von Kanten hat. T ist also ein Baum und wird oft als (auf-)spannender Baum (*spanning tree*) bezeichnet. Die *Länge* von T ist:

$$l(T) = \sum_{e \in F} l(e).$$

$l(T)$ ist für dieses Problem die "Zielfunktion", wobei ein minimales $l(T)$ gesucht wird.

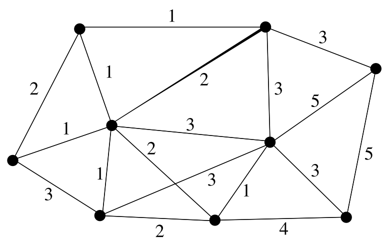


Abbildung 18: ein ungerichteter Graph mit Kantenbewerung

"Lösungskandidaten" sind zunächst alle möglichen Gerüste T von G, von denen eines mit der kleinsten Länge gesucht wird. Da es mehrere verschiedene Gerüste mit der gleichen Gesamtlänge geben kann, muss dieses nicht eindeutig sein! Der *Lösungsraum* läßt sich also schreiben als: $\mathcal{L}_G = \{T_G \mid T_G \text{ ist Gerüst von } G\}$.

Wir suchen also eine optimale Lösung für das Problem-Beispiel (G, l):
$T_G^* \in \mathcal{L}_G$ ist "optimale" Lösung aus $\mathcal{L}_G \Leftrightarrow T_G^* \in \mathcal{L}_G$ mit $l(T_G^*) \leq l(T_G)$ für alle $T_G \in \mathcal{L}_G$.

Die *Anzahl der Lösungen* ist $|\mathcal{L}_G|$. Es gilt dabei:

$$1 \leq |\mathcal{L}_G| \leq |V|^{|V|-2} (= n^{n-2}) \qquad \text{mit } |V| = n.$$

Dabei ist $|V|^{|V|-2}$ die Anzahl der verschiedenen Gerüste des vollständigen Graphen K_n.

Für weiterführende Informationen über Minimalgerüste sei auf die graphentheoretische Literatur verwiesen.

Es gibt mehrere Algorithmen zur Bestimmung eines Minimal-Gerüsts eines Graphen.

Wir betrachten beispielsweise den folgenden Algorithmus von Prim/Dijkstra.

Algorithmus MINGERÜST von Prim/Dijkstra:

Sei $G = (V, E)$ ein ungerichteter Graph mit der Kantenbewertung l:

1. Initialisierung: Setze $X := \emptyset; U := \emptyset$.
2. Wähle einen beliebigen Knotenpunkt $v \in V$ aus.
   Setze $X := \{v\}; V := V - \{v\}$.
3. Betrachte alle Kanten $(a, b) \in E$ mit

$$a \in X \text{ und } b \in V \qquad (\text{oder } b \in X \text{ und } a \in V).$$

Wähle unter diesen Kanten eine kürzeste aus. O.B.d.A. sei (a_1, b_1) eine solche. Dann gilt:

$$l(a_1, b_1) \leq l(a, b) \quad \text{für alle } (a, b) \in E \; mit \; a \in X, b \in V.$$

4. Bilde $\begin{aligned} X &:= X \cup \{b_1\}; \\ U &:= U \cup \{(a_1, b_1)\}; \\ V &:= V - \{b_1\}. \end{aligned}$

5. Wenn $V \neq \emptyset$ goto 3.;

6. STOP.

Output: $T = (V, U)$.
Der Graph T ist ein Minimalgerüst von G.

Wenn wir die in Schritt 3 jeweils ausgewählten Kanten (z.B. rot) färben, dann bilden am Ende des Algorithmus die gefärbten Kanten ein Minimalgerüst.

Komplexität: Im Schritt 3. sind höchstens m Vergleiche auszuführen. Schritt 3. wird $(n-1)$-mal aufgerufen.

$$\rightarrow T_A = O(m * n)$$

Die **untere Schranke** ist $\Omega(m)$, da jede Kante mindestens einmal geprüft werden muss. (Anderenfalls kann nicht sichergestellt werden, dass das betrachtete Gerüst minimal ist.)

Bei dieser Komplexitätsabschätzung wurde bisher nicht auf die Datenstruktur zur Darstellung des Graphen geachtet und auch nicht auf die verschiedenen Suchverfahren die in Schritt 3. zur Auswahl der kürzesten Kante benutzt werden können. Tut man dieses und verbessert dieses Grobgerüst des Prim/Dijkstra-Algorithmus, so sind Implementationen möglich (Vergleichen Sie [CLR91]), die in $O(|E| \cdot \log |V|)$, ja sogar in $O(|E| + |V| \cdot \log |V|)$ Schritten arbeiten. Andere Algorithmen für diese Problem kommen für lichte Graphen, d.h. solche mit dünn besetzten Adjazenzmatrizen, mit noch weniger Schritten aus. Die Suche nach dem besten Algorithmus wird also im Wesentlichen durch die nachweisbar untere Schranke eingeschränkt!

Bei selbstentwickelten Algorithmen muss immer auch ein Nachweis für die Korrektheit geliefert werden, was hier auch geschehen soll. Es gibt allerdings Datenbanken für bisher beste Implementationen, bzw. Umsetzungen von Algorithmen (z.B. LEDA, Univ. Saarbrücken) auf die weltweit zugegriffen werden kann. Bei deren Verwendung sind natürlich andere Probleme, wie die Anpassung an die bestehende Programmierumgebung und Hardware vorrangig zu Lösen.

Nachweis der Korrektheit des Algorithmus:

6.1 Theorem

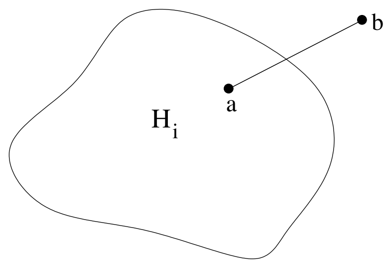
Der Algorithmus **MINGERÜST** liefert stets ein Minimal-Gerüst eines Graphen G, wenn G zusammenhängend ist.

$\diamond$ **Beweis:**

Im Verlaufe des Algorithmus **MINGERÜST** *werden* sukzessive Teil-Unter-Graphen (*TU-Graphen*) von G erzeugt:

$$H_1, H_2, \ldots, H_i, H_{i+1}, \ldots, H_n \text{ mit } \quad H_1 = (\{v\}, \emptyset) \text{ und } \quad H_n = (V, U)$$

- Alle TU-Graphen $H_1, \ldots, H_n$ sind Bäume, denn H_1 ist ein Baum, und wenn H_i ein Baum ist, dann auch H_{i+1}, denn es wird in den Schritten 3. und 4. stets eine neue "hängende" Kante hinzugefügt (zu einem Knoten außerhalb des Baumes H_i).



- Für $i = 1, 2, \ldots, n-1$ gilt: Wenn H_i TU-Graph eines Minimalgerüsts von G ist, so ist auch H_{i+1} ein TU-Graph eines Minimalgerüsts von G.

Angenommen, $H_i = (V_i, U_i)$ ist ein TU-Graph eines Minimalgerüsts T von G.

Da T ein Gerüst von G ist, gibt es eine Kante $e_1 = (a_1, b_1) \in U$ mit $a_1 \in V_i, b_1 \in V \setminus V_i$. Sei nun $\tilde{e}$ die Kante, die im i-ten Schritt (Erzeugung von H_{i+1} aus H_i) des Algorithmus ausgewählt wird.

Wenn $\tilde{e} = e_1$, dann fertig.

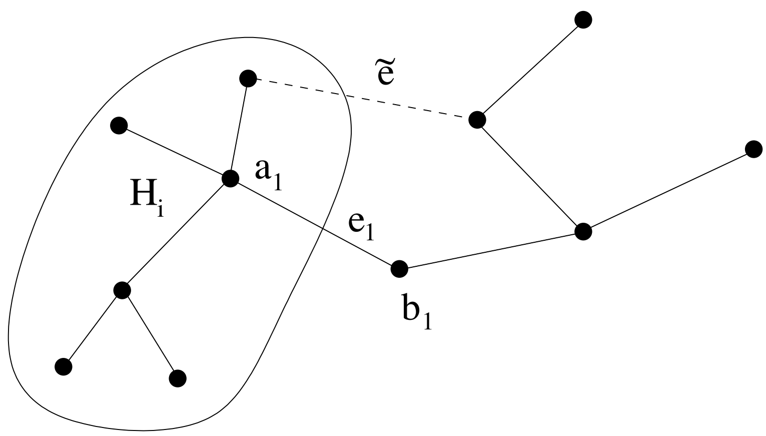


Abbildung 19: der TU-Graph H_i eines Graphen $T = (V, U)$

Wenn $\tilde{e} \neq e_1$, dann gilt $\tilde{e} = (\tilde{a}, \tilde{b}) \in E$ mit $\tilde{a} \in V_i$ und $\tilde{b} \in V - V_i$.

Wir fügen $\tilde{e}$ zu T hinzu. Dann entsteht genau ein Kreis C. Wegen Schritt 3. muss $l(\tilde{e}) \leq l(e_1)$ gelten. Bilden wir nun

$$\tilde{T} = (T \setminus \{e_1\}) \cup \{\tilde{e}\} \quad \rightarrow \quad l(\tilde{T}) \leq l(T).$$

Wegen der Minimalität von T muss auch $\tilde{T}$ ein Minimalgerüst sein. Damit ist H_{i+1} auch ein TU-Graph eines Minimalgerüsts $\tilde{T}$ von G. $\square$

Es gibt also Probleme, für die ein Greedy-Algorithmus eine optimale Lösung liefert (Minimalgerüst) und Probleme, für die ein Greedy-Algorithmus nicht immer eine optimale Lösung liefert (Münzrückgabe, siehe Seite 112).

Frage: Wie kann man diejenigen Probleme charakterisieren, für die ein Greedy-Algorithmus *stets* die optimale Lösung liefert?

- Eine Antwort darauf wird durch die Theorie der Matroide gegeben. (Siehe [CLR91]).

Ein "Greedy-Algorithmus" liefert immer dann eine optimale Lösung, wenn das betrachtete Problem eine *matroidale Struktur* hat. "Matroide" sind Verallgemeinerungen von Matrizen. Wir wollen auf die nötigen Definitionen hier jedoch nicht eingehen.

Beispiel 5: Vereinfachtes Rucksackproblem:

Das Vereinfachte Rucksackproblem ist folgendes: Ein Rucksack ist bis zu einem maximalen Gewicht *max* mit Dingen von beschränkter Verfügbarkeit so zu befüllen, dass der Inhalt des Rucksacks möglichst wertvoll ist. Die Anzahlen der zur Verfügung stehenden Objekte jeder Sorte, deren Wert und jeweiliges Gewicht sind natürlich vorher bekannt. Wenn wir der Einfachheit halber annehmen, dass alle Dinge etwa die gleiche Größe und Form haben und wie Zucker oder Sand beliebig portionierbar sind (daher die Bezeichnung "Vereinfachtes Rucksackproblem"), so können Größenprobleme keine zusätzliche Einschränkung bedeuten. Wir geben daher nur das von den Objekten eines Typs vorhandene und benutzbare Gesamtgewicht an, sowie den Preis pro Gewichtseinheit.

i	Objekttyp	g_i	p_i
1	Nägel	5,0 kg	17,00 DM/kg
2	Heftpflaster	0,1 kg	100,00 DM/kg
3	Cognac	1,5 kg	35,00 DM/kg
4	Toilettenpapier	0,5 kg	8,00 DM/kg
5	Käse	9,0 kg	18,00 DM/kg
6	Brot	1,0 kg	3,40 DM/kg

Tabelle 5: Maximales Rucksackgewicht $max = 15$ kg

In Tabelle 5 stellt g_i das nutzbare Gesamtgewicht der von Typ i verfügbaren Objekte und p_i deren Preis pro Gewichtseinheit dar.

Der Greedy-Algorithmus "schielt" immer nach dem lokalen maximalen Gewinn, wählt als jedesmal soviel von dem Element mit dem größten p_i, wie vorhanden ist. In diesem Beispiel wird also zunächst alles Pflaster, dann der Cognac, gefolgt vom Käse komplett verstaut. Das sind dann schon 10,6 kg und die verbleibenden 4,4 kg können wir mit dem nächst Wertvollen, den Nägeln, auffüllen.

Bei dem Einfachen Rucksack Problem liefert der Greedy-Algorithmus sogar die optimale Lösung. Wenn wir mit dem Rucksack jedoch nicht zu einem Pfandhaus gehen wollen, sondern auf eine Wanderung, so müssen wir neben den realen Gewichten auch den persönlichen Nutzen, beziehungsweise den jeweils unterschiedlichen Platzbedarf mit einbeziehen. Das allgemeine Rucksackproblem werden wir nicht mehr in allen Instanzen mit dem Greedy-Algorithmus optimal lösen können und kommen darauf später noch mehrfach zurück (vergl. Abschnitt 6.4, Def. 6.4 und Kapitel 8, Theorem 23). Wir sehen dies ähnlich auch an einem anderen Beispiel:

Beispiel 6: Münzrückgabe

Ein Kassen- oder Getränkeautomat soll so programmiert werden, dass er den Rückgabebetrag stets mit der kleinsten Anzahl von Münzen auswirft. Von jeder Münzsorte, hier 5 DM, 2 DM,1 DM, 50 Pf,10 Pf, 5 Pf, 2 Pf, und 1 Pf, seien ausreichend viele vorhanden. Ein Greedy-Algorithmus würde nun bei 63 Pf Wechselgeld zunächst das lokale Optimum mit einer 50 Pf Münze erreichen. Die noch fehlenden 13 Pf würden dann als je eine 10, 2, und 1 Pfennigmünze herausgegeben. Das sieht optimal aus, denn nur 4 Münzen wurden benötigt. Bei dieser Auswahl von Münzen ist das Verfahren tatsächlich optimal, aber das gilt nicht allgemein bei jeder Währung: Bei einer Währung mit 50, 25, 10 und 1 cent Münzen, würde ein Betrag von 30 cent mit 6 Münzen: ein 25 cent Stück und fünf 1 cent Münzen herausgegeben. Tatsächlich sind aber drei 10 cent Stücke die beste Lösung!

Für die Programmierung solcher Probleme bietet sich als Technik die sogenannte Dynamische Programmierung an, die wir am Beispiel des üblichen, also des Allgemeinen Rucksackproblems erläutern wollen.

6.4 Dynamische Programmierung

Die Idee der Dynamische Programmierung ist es, die bei der Zerlegung eines Problems in Teilprobleme und der Lösung des Ausgangsproblems durch Lösung der Teilprobleme (Rekursion) die Lösungen der Teilprobleme zur späteren Wiederverwendung aufzubewahren, d.h. zu tabellieren.

Ein Wanderer kann auf einer Wandertour in seinem Rucksack mit dem Fassungsvermögen L Gegenstände unterschiedlicher Größe l_i $(1 \leq i \leq m)$ und unterschiedlichen "Wertes" d_i $(1 \leq i \leq m)$ mitnehmen, von denen jeweils beliebig viele im Lager vorhanden sind.

Welche und wieviele Gegenstände sind einzupacken, wenn der Wert des Rucksackinhalts maximal sein soll?

Sei x_i die Anzahl der einzupackenden Gegenstände vom Typ $i, (1 \leq i \leq m)$.

Beispiel 7: Rucksackproblem

Allgemeines Rucksackproblem (RP):

Gegeben: Eine Menge von geordneten Paaren $(l_i, d_i) \in I\!N \times I\!N, 1 \leq i \leq m$, wobei l_i die "Größe" und d_i den "Wert" des i-ten Gegenstands bezeichnen. Ferner sei $L \in I\!N$, die "Rucksack-Kapazität" vorgegeben.

Gesucht: Ein Vektor $\mathbf{x} = (x_1, x_2, \ldots, x_m) \in I\!N^m$, der das folgende Optimierungsproblem löst:

1. $w = \sum\limits_{i=1}^{m} d_i x_i$ ist maximal

unter der Nebenbedingung:

2. $\sum\limits_{i=1}^{m} l_i x_i \leq L$.

Das Rucksackproblem ist das einfachste *ganzzahlige lineare Optimierungsproblem* (ILOP), nämlich mit nur einer Restriktion der Form (2) als Nebenbedingungen. Zur mathematischen Behandlung von linearen Optimierungsproblemen finden die Leser Hinweise in [Möl97].

Wir betrachten neben (RP) eine ganze Schar von Problemen als parametrisierte Rucksackprobleme, die das ursprüngliche Problem (RP) als Spezialfall enthält.

Für alle k mit $k \in I\!N$ und $1 \leq k \leq m$ und für alle r mit $r \in I\!N$ und $1 \leq r \leq L$ sei das diesen Parametern zugehörige Rucksackproblem:

Parametrisiertes Rucksackproblem ($\widetilde{RP}$):

1. $F(k,r) = \sum\limits_{i=1}^{k} d_i x_i$ ist maximal

unter den Nebenbedingungen:

2. $\sum\limits_{i=1}^{k} l_i x_i \leq r$,
3. $x_i \geq 0$ für i mit $1 \leq i \leq k$,
4. x_i ganzzahlig mit $1 \leq i \leq k$.

Für $k = m$ und $r = L$ erhalten wir das Problem (RP).

Wir treffen zusätzlich folgende Vereinbarungen: Es sei

$$\begin{aligned} F(0,r) &= 0 \qquad \mathit{f\ddot{u}ralle}\ r \geq 0, \\ F(k,0) &= 0 \qquad \mathit{f\ddot{u}ralle}\ 1 \leq k \leq m \end{aligned}$$

und setzen

$$F(k,r) = -\infty \quad \text{für } r < 0 \text{ und } 1 \leq k \leq m.$$

Wir lösen nun sukzessive die Probleme ($\widetilde{RP}$) für k und r.

$$F^*(k,r) \text{ sei maximaler Wert der Zielfunktion } F(k,r).$$

Es gibt 2 Möglichkeiten:

1. Fall: In der optimalen Lösung von $F(k,r)$ ist $x_k = 0$.
In diesem Fall ist $F^*(k,r) = F^*(k-1,r)$.

2. Fall: In der optimalen Lösung von $F(k,r)$ ist $x_k \geq 1$.
Hier können wir feststellen, dass für $x_k \geq 1$ gelten muss: $r \geq l_k$.

$$\begin{aligned} F^*(k,r) &= \max\left\{ d_k + \sum_{1=1}^{k} d_i x_i \mid \sum_{1=1}^{k} l_i x_i \leq r - l_k, x_i \geq 0, x_i \text{ ganz.} \right\} \\ &= d_k + \max\left\{ \sum_{1=1}^{k} d_i x_i \mid \sum_{1=1}^{k} l_i x_i \leq r - l_k, x_i \geq 0, x_i \text{ ganz.} \right\} \\ \rightarrow F^*(k,r) &= d_k + F^*(k, r - l_k). \end{aligned}$$

Insgesamt muss also sein:

$$F^*(k,r) = \max\{F^*(k-1,r), d_k + F^*(k, r-l_k)\} \quad \text{für } 1 \leq k \leq m, 0 \leq r \leq L.$$

Das nennt man eine **Rekursionsformel der dynamischen Programmierung**.

Mit dieser Rekursionsformel kann man sukzessive die optimalen Werte $F^*(k,r)$ berechnen, bis man schließlich $F^*(m,L)$ erhalten hat. Durch eine Rückwärts-Rechnung kann man dann schließlich eine optimale Lösung x^* erhalten.

Beispiel:

Gegeben: 3 Arten von Gegenständen. Der Gegenstand G_i habe die Größe l_i und den Wert d_i. Der Rucksack habe die Kapazität $L = 14$.

Frage: Wieviele Gegenstände x_i vom Typ i sind in den Rucksack zu packen, damit der Gesamtwert maximal ist?

Inputdaten:

i	1	2	3
l_i	4	7	9
d_i	5	10	12

mathematisches Modell:

$$F = 5x_1 + 10x_2 + 12x_3 \to \text{ Max!}$$

Nebenbedingungen:

$$\begin{aligned} 4x_1 + 7x_2 + 9x_3 &\leq 15, \\ x_1, x_2, x_3 &\in I\!N. \end{aligned}$$

Gesucht: eine optimale Lösung $x^* = (x_1, x_2, x_3)$.

Wir betrachten sukzessive die optimalen Werte $F^*(k,r)$ und stellen sie in einer Tabelle zusammen:

r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
$F^*(0,r)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
$F^*(1,r)$	0	0	0	0	5	5	5	5	10	10	10	10	15	15	15	15	
$F^*(2,r)$	0	0	0	0	5	5	5	10	10	10	10	15	15	15	20	20	
$F^*(3,r)$	0	0	0	0	5	5	5	10	10	12	12	15	15	17	20	20	$= F^*(3,15)$

Optimaler Wert der Zielfunktion ist also: $w = F^*(3,15) = 20$.

Zur Ermittlung der optimalen Lösung x^* wird im Verlaufe des Algorithmus (neben der Erstellung der obigen Taballen) zusätzlich eine *Indextabelle* mit Einträgen $I(k,r)$ erstellt.

$$I(k,r) = \begin{cases} I(k-1,r), \text{ falls } F^*(k-1,r) > d_k + F^*(k, r-l_k) \\ k, \text{ falls } F^*(k-1,r) < d_k + F^*(k, r-l_k) \\ I(k-1,r) \vee k, \text{ falls } F^*(k-1,r) = d_k + F^*(k, r-l_k) \end{cases}$$

Ferner sei $I(k,r) = 0$ falls $F(k,r) \leq 0$.

Dabei kann $I(k,r) = i$ so interpretiert werden, dass der i-te Gegenstand wenigstens einmal in der "Rucksackmenge"vorkommt, d.h. $x_i \geq 1$.

Indextabelle:

r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$I(1,r)$	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
$I(2,r)$	0	0	0	0	1	1	1	2	$1 \vee 2$	$1 \vee 2$	$1 \vee 2$	2	$1 \vee 2$	$1 \vee 2$	2	2
$I(3,r)$	0	0	0	0	1	1	1	2	$1 \vee 2$	3	3	2	$1 \vee 2$	3	2	2

Ermittlung der optimalen Lösung $x^* = (x_1, x_2, \ldots, x_m)$:

1. Initialisierung: Setze $x_i = 0$ für alle i mit $1 \leq i \leq m$.

   $i := I(m,l)$,

   $r := L$.

2. while $i > 0$ do

   $x_i := x_i + 1$,

$$r := r - l_i,$$

$$i := I(m, r).$$

Beispiel: Start mit $x_1 = 0, x_2 = 0, x_3 = 0$.

$$\left.\begin{array}{l} i{=}I(3,15) = 2 \\ r{=}15 \end{array}\right\} \rightarrow x_2 = 1, r = 15 - l_2 = 15 - 7 = 8.$$

$$\rightarrow i := I(3,8) = 1 \vee 2.$$

Fall a) $\left.\begin{array}{l} i{=}2 \\ r{=}8 \end{array}\right\} \rightarrow x_2 = x_2 + 1 := 2; r := 8 - 7 = 1.$

$i := I(3,1) = 0,$

$\rightarrow$ Lösung $x^* : x_1 = 0, x_2 = 2, x_3 = 3.$

Fall b1) $\left.\begin{array}{l} i{=}1 \\ r{=}8 \end{array}\right\} \rightarrow x_1 = 1; r := 8 - 4 = 4.$

$i := I(3,4) = 1.$

Fall b2) $\begin{array}{c} x_i{:=}x_1 + 1 = 2 \\ r{:=}4 - 4 = 0 \end{array}.$

$i := I(3,0) = 0.$

$\rightarrow$ Lösung $x^* : x_1 = 2, x_2 = 1, x_3 = 0.$

Komplexität: Das Aufstellen der Tabellen erfordert $O(m \cdot L)$ Vergleiche. Die Frage, ob das ein polynomialer Algorithmus ist, muss verneint werden. Als Input haben wir $L, l_1, \ldots, l_m$ und $d_1, \ldots, d_m$, das sind $2m + 1$ Größen, jede Inputgröße y hat die Inputlänge $\log(y)$, also ist die Inputlänge insgesamt $O(m \cdot \log(L))$ und $m \cdot L$ ist nicht als Polynom der Eingabegröße $m \cdot \log(L)$ darstellbar! Wir werden später sogar beweisen, dass bisher niemand das Rucksackproblem mit einem polynomialen Algorithmus deterministisch lösen kann, solange die Eingabe nicht unär codiert wird (vergleiche Kapitel 8, Theorem 23)!

Auch das folgende Problem erlaubt bis jetzt noch kein deterministisches Verfahren, welches alle Instanzen mit polynomialen Aufwand lösen kann.

Beispiel 8: Travelling Salesman Problem

Gegeben: Eine Menge von Orten (oder Knotenpunkten) $V = \{1, 2, \ldots, n\}$.
Eine $(n \times n)$-Matrix $D = (d_{i,k}) \in I\!N^{n \times n}$ - die Distanzmatrix, der durch Einbahnstraßen erreichbaren Orte.

Gesucht: Eine Rundreise (Hamiltonkreis) K durch mindestens (exakt) alle n Orte von V mit minimaler Gesamtlänge.

Es ist ein kleiner Unterschied, aber kein gravierender, ob man erlaubt, dass bei einer Rundreise ein Ort mehrfach besucht werden darf oder nicht. In Hamiltonkreisen kommt jeder Knoten exakt einmal vor, aber es gibt kürzeste Rundreisen, besonders bei nicht vollständigen Graphen, bei denen ein Ort zweimal besucht werden sollte. Wenn die Distanzmatrix die eines ungerichteten, bewerteten Graphen (ohne Mehrfachkanten) ist, so ist diese immer symmetrisch! In diesem Fall stellen wir uns jede ungerichtete Kante durch zwei in gegengesetzte Richtung weisende, aber gerichtete Kanten mit gleicher Entfernung vor, müssen aber berücksichtigen, dass beim Entfernen einer dieser gerichten Kanten zugleich auch immer deren Pendent entfernt werden muss.

Für das Rundreiseproblem kann man die Methode der dynamischen Programmierung wie folgt anwenden:

Sei $S \subseteq \{2, 3, \ldots, n\}$ und $k \in S$. Setze:

$c(S, k) :=$ Länge eines kürzesten Weges, der in 1 beginnt, in k endet und alle Orte (Knotenpunkte) in S enthält.

Dann gilt (wenn $|S| = 1$):

$$c(\{k\}, k) = d_{1k} \quad \text{für alle } k = 2, \ldots, n.$$

Wenn $|S| > 1$, dann berechnen wir $c(S, k)$ für jede Teilmenge S und registrieren gleichzeitig den kürzesten Weg von 1 nach k über S.

Schließlich berechnen wir:

$$c_{min} := \min_{k \neq 1} \{c(\{2, \ldots, n\}, k) + d_{k1}\}.$$

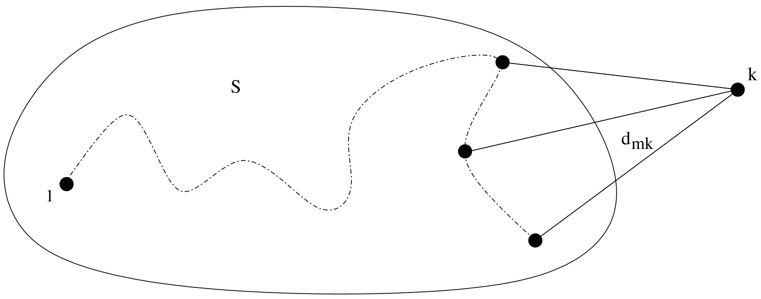


Abbildung 20: $c(S,k) := \min_{m \in S-\{k\}} \{c(S-k,m) + d_{mk}\}$

Wenn man jeweils denjenigen Ort m aufbewahrt, in dem das Minimum angenommen wird, dann kann man durch Backtracking eine kürzeste Rundreise ermitteln. Die Komplexität dieses Algorithmusses ist exponentiell $O(n^2 \cdot 2^n)$ und wesentlich bessere, also um Größenordnungen bessere, also polynomiale deterministische Verfahren sind bisher nicht bekannt!

Wir wollen den Abschnitt zur Dynamischen Programmierung mit einem Beispiel abschließen, das tatsächlich ein polynomiales Verfahren betrifft.

Beispiel 9: *Matrix Chain* Produkt

Betrachten Sie eine Kette von Matrizen, die multipliziert werden müssen. Da diese Multiplikation nicht kommutativ ist, kommt es also wesentlich auf die Reihenfolge an. Das alleine ist jedoch noch nicht ausreichend. Betrachten Sie folgendes zu berechnende Produkt:

$$\mathbf{A}\cdot\mathbf{B}\cdot\mathbf{C}\cdot\mathbf{D}\cdot\mathbf{E}\cdot\mathbf{F} = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{pmatrix} \cdot \begin{pmatrix} c_{1,1} \\ c_{2,1} \\ c_{3,1} \end{pmatrix} \cdot \begin{pmatrix} d_{1,1} & d_{1,2} \end{pmatrix} \cdot$$
$$\begin{pmatrix} e_{1,1} & e_{1,2} \\ e_{2,1} & e_{2,2} \end{pmatrix} \cdot \begin{pmatrix} f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,1} & f_{2,2} & f_{2,3} \end{pmatrix}.$$

Wenn Sie von links nach rechts vorgehen, benötigt zunächst die Multiplikation der 4×2 Matrix $\mathbf{A}$ mit der 2×3 Matrix $\mathbf{B}$ genau $4\cdot2\cdot3 = 24$ Multiplikationen mit dem Ergebnis der 4×3 Matrix $\mathbf{A}\cdot\mathbf{B}$. So weiter Vorgegangen erhalten wir insgesamt 84 Multiplikationen um das Produkt $((((\mathbf{A}\cdot\mathbf{B})\cdot\mathbf{C})\cdot\mathbf{D})\cdot\mathbf{E})\cdot\mathbf{F}$ zu bestimmen. Wenn wir allerdings von rechts aus beginnen, also $\mathbf{A}(\cdot\mathbf{B}(\cdot\mathbf{C}(\cdot\mathbf{D}(\cdot\mathbf{E}\cdot\mathbf{F}))))$ bestimmen, benötigen wir nur 69 der gegenüber der Addition wesentlich teureren Multiplikationen. Für jede der möglichen Klammerungen erhalten wir andere Ergebnisse für den zu leistenden Aufwand. Zum Beispiel erzielt folgende Klammerung - also Reihenfolge der Multiplikationen - das beste Ergebnis mit nur 36 Multiplikationen: $(\mathbf{A}\cdot(\mathbf{B}\cdot\mathbf{C}))\cdot((\mathbf{D}\cdot\mathbf{E})\cdot\mathbf{F})$. Sehr viel eklatanter werden die Unterschiede, wenn die Matrizen das hundertfache an Zeilen und Spalten besitzen, was eben erst die schnellen Verfahren erforderlich macht! Und selbst wenn wir dann, wie im Kapitel 8 beschrieben, eine schnelle Matrixmultiplikation verwenden, so ist die richtige Reihenfolge bei der Produktbildung immer noch wesentlich ausschlaggebender.

Was, wenn wir zunächst für alle Klammerungsmöglichkeiten die Anzahl der Produkte bestimmten und dann die beste heraussucthen? Nun, die Anzahl aller möglichen Klammerungen von n Matrizen zu Produkten ist $C(n-1)$, wobei $C(n) = \frac{1}{n+1}\binom{2n}{n}$ die n-te Catalan'sche Zahl ist, mit einer Größenordnung von $\Omega(\frac{4^n}{n^{3/2}})$. Dieses exponentielle Verfahren bietet sich also sicher *nicht* an.

Wir bemerken, dass in einer optimalen Klammerung von $\mathbf{A}_1\cdot\mathbf{A}_2\cdot\ldots\cdot\mathbf{A}_n$ jeder Teil $\mathbf{A}_i\cdot\ldots\cdot\mathbf{A}_j$, $1\le i<j\le n$, eine optimale Klammerung für sich selbst besitzt. Das bedeutet, dass sich die optimale Lösung des Gesamtproblems aus optimalen Teillösungen zusammensetzt. Daher eignet sich die Dynamische Programmierung hier besonders gut.

Wir vereinbaren, dass die n zu multiplizierenden Matrizen $\mathbf{M}_i$ jeweils r_i Zeilen und r_{i+1} Spalten besitzen. Die minimalen Kosten für das Produkt $\mathbf{M}_l\cdot\ldots\cdot\mathbf{M}_r$ werden in die Kostenmatrix $cost(l,r)$ eingetragen und zu Beginn für $l\neq r$ mit ∞, einem sehr hohen, niemals erreichbaren Wert, initialisiert. Nur die Werte von $cost(i,i)$ werden für alle i mit 0 eingesetzt. Während in $cost(l,r)$ die minimalen Kosten für das Produkt stehen, wird in der Matrix $best(l,r)$ der Index für die Aufteilungsstelle in zwei Teilprodukte stehen.

Um aus den bekannten Daten die richtige Klammerung des Produktes zu ermitteln, wird obige rekursive Prozedur $order(i,j)$ (Abbildung 22) benötigt. Als Tabellen-Ausgabe erhalten wir mit Angabe von $cost(i,j)$ auch die Klammerung (Überflüssiges ist dabei weggelassen):

```
WENN l ≠ r
DANN cost(l,r) ⟵ ∞;
FÜR i VON 1 BIS n WIEDERHOLE
  cost(i,i) ⟵ 0;
FÜR j VON 1 BIS n − 1 WIEDERHOLE
  FÜR i VON 1 BIS n − j WIEDERHOLE
    FÜR k VON i + 1 BIS i + j WIEDERHOLE
      t ⟵ cost(i,k − 1) + cost(k,i + j) + r_i · r_k · r_{i+j+1};
      WENN t < cost(i,i + j) DANN
        cost(i,i + j) ⟵ t; best(i,i + j) ⟵ k;
END.
```

Abbildung 21: Kostenermittlung beim Matrix Chain Algorithmus

```
PROCEDUR order(i,j)
      WENN i = j
      DANN SCHREIBE "M_i";
      SONST SCHREIBE "(";
            order(i,best(i,j) − 1); order(best(i,j),j);
            SCHREIBE ")";
      END.
```

Abbildung 22: Ermittlung der Klammerung für optimales Matrix Chain Produkt

	B	C	D	E	F
A	24 (A)(B)	14 (A)(BC)	22 (ABC)(D)	26 (ABC)(DE)	36 (ABC)(DEF)
B		6 (B)(C)	10 (BC)(D)	14 (BC)(DE)	22 (BC)(DEF)
C			6 (C)(D)	10 (C)(DE)	19 (C)(DEF)
D				4 (D)(E)	10 (DE)(F)
E					12 (E)(F)

Aus der Betrachtung des Programms folgt, dass das Matrix Chain Problem mit einem Zeitaufwand der Größenordnung $O(n^3)$ und Platzbedarf von $O(n^2)$ gelöst werden kann. Dabei ist zu berücksichtigen, dass der relativ große Zeit- und Platzbedarf dieses Verfahrens durch die erzielbaren Einsparungen praktisch vernachlässigbar ist.

Wer meint, ein rekursiver Algorithmus auf Grund der Gleichung

$$cost(i,j) = cost(i,k) + cost(k+1,j) + r_i \cdot r_k \cdot r_{j+1}$$

sei besser, bedenke zunächst, dass in dieser Rekursion so getan wird, als kennte man den Teilungsindex k, was aber nicht der Fall ist. Also muss dieser in der Rekursion gesucht, d.h. alle dafür möglichen Werte probiert werden. Man erhält so, z.B.:

$$cost(i,j) = \begin{cases} 0 & \text{, if } i = j \\ \min\limits_{i \leq k < j}(cost(i,k)) + cost(k+1,j) + r_i \cdot r_{k+1} \cdot r_{j+1} & \text{, } i < j \end{cases} .$$

Eine Analyse dieser Rekursion mit den Methden aus Kapitel 7 ergibt aber exponentielles Laufzeitverhalten für die rekursive Lösung!

6.5 Backtracking

Ein Backtracking Verfahren arbeitet nach dem Prinzip "Versuch und Irrtum" (*trial and error*) und wird meist dort angewendet, wo keine Strategien für lokale Verfahren zur Verfügung stehen. Man versucht eine bisher gefundene Teillösung durch eine der möglichen Erweiterungen zu der Gesamtlösung auszubauen. Erst wenn dies in eine Sackgasse führt, geht man einen Schritt zurück und wählt eine andere Fortsetzung. Alle (nicht optimierten) Prolog-Programme arbeiten so bei der Suche nach einer Antwort auf eine Anfrage an die Wissensbasis. Da das Verfahren exponentiell mit der Suchtiefe anwächst, eignet es sich für die Praxis nur, wenn durch Zusatzbedingungen schon vorab viele der sonst möglichen Sackgassen ausgeschlossen werden können (vergleiche Abschnitt 6.6 zu dem Branch-and-Bound-Verfahren).

Beispiel 10: 8 Königinnen-Problem

Man positioniere eine maximale Anzahl von Königinnen so auf dem Schachbrett, dass sich keine zwei bedrohen. Offensichtlich können dies niemals mehr als 8 sein, weshalb man meist von dem 8 Königinnen-Problem spricht.

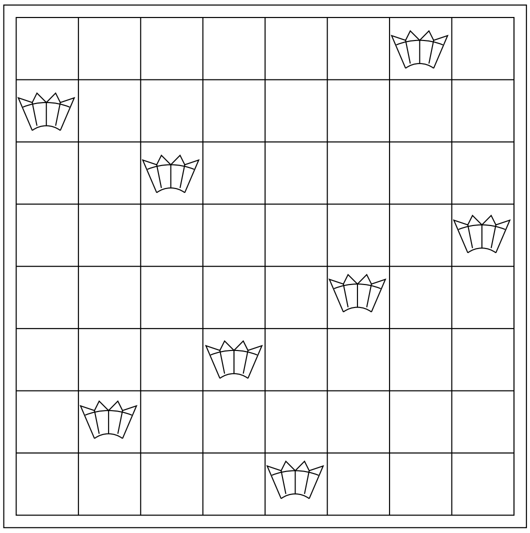


Abbildung 23: Das 8 Königinnen Problem

Die maximale Anzahl sich gegenseitig nicht bedrohender Königinnen auf einem Schachbrett ist 8. Interessant ist dann die

Frage: *Wieviele verschiedene Lösungen des Königinnen-Problems gibt es?*

6.6 Branch-and-Bound Verfahren

Branch-and-bound Verfahren sind eine Modifikation des reinen Backtracking, bei dem nun bei den Verzweigungsschritten die anzugehenden Subprobleme vor deren in Angriffnahme mit dem Optimum der zu erreichenden Zielfunktion bewertet werden. Die Auswahlentscheidung richtet sich dann nach jener Möglichkeit, die einen best möglichen Wert der Zielfunktion erwarten lässt. Die Zweige, deren Bewertung schlechter als der bisher beste erreichte Wert der Zielfunktion liegen, werden zunächst alle von einer weiteren Bearbeitung ausgeschlossen. An diesen wird erst dann weitergemacht (*back-tracking*), wenn ihre Zielfunktion von allen erreichbaren Zielfunktionen der Subprobleme überschritten wird! Das wohl bekannteste Beispiel für Programme, die nach dieser Methode arbeiten sind jene Spielprogramme, wie z.B. Schach, für die keine sichere Gewinnstrategie existiert.

Wir schauen uns dies am inzwischen bekannten Beispiel des Handlungsreisenden an.

Beispiel 11: Branch and Bound Verfahren zum Handlungsreisenden

Seien also n Städte $s_1, s_2, \ldots, s_n$ zusammen mit einer Entfernungstabelle $D \in I\!N^{n \times n}$ gegeben, in der $d_{i,j}$ die Entfernung der durch gerichtete Kanten (s_i, s_j) miteinander verbundenen Städte s_i und s_j steht. Wir setzen für jedes i, $1 \leq i \leq n$ den Wert $d_{i,i} := \infty$ ein, der auch immer dort steht, wo Städte nicht miteinander verbunden sind, also nicht mit dem zur Verfügung stehenden Verkehrsmittel erreicht werden können. Die Rundreise beginnt in der Stadt s_1 und die zu minimierende Zielfunktion ist die Länge des Rundweges, der hier jede Stadt auch nur einmal durchlaufen darf! Um das Branch-and-Bound Verfahren anwenden zu können, benötigen wir eine untere Schranke für die Zielfunktion. Diese ist zum Beispiel gegeben durch die halbe Summe der kürzesten An- und Abfahrtsswege aller Städte, da jede Stadt auf der optimalen Rundreise einmal betreten und einmal verlassen werden muss. Wenn auch die anfängliche Entfernungsmatrix symmetrisch ist, so brauchen die im weiteren Verlauf entstehenden Matrizen dies nicht mehr zu sein. Daher verwenden wir als untere Schranke den Wert

$$d := \frac{d_{in} + d_{out}}{2},$$

mit

$$d_{in} := \sum_{i=1}^{n} \min\{d_{j,i} | j \in \{1, \ldots, n\}\}$$

und

$$d_{out} := \sum_{i=1}^{n} \min\{d_{i,j} | j \in \{1, \ldots, n\}\}.$$

Wir beschreiben die Technik an einem Beispiel. Sei also das Rundreiseproblem gegeben durch die Entfernungsmatrix P_0 eines hier gerichteten Graphen, d.h. jede Verbindung kann je nach Richtung eine andere Länge (bzw. Kosten-Gewichtung) besitzen!

$$\begin{array}{c c}
P_0 & \begin{array}{cccc} s_1 & s_2 & s_3 & s_4 \end{array} \\
\begin{array}{c} s_1 \\ s_2 \\ s_3 \\ s_4 \end{array} & \begin{pmatrix} \infty & 3 & 2 & 7 \\ 4 & \infty & 3 & 6 \\ 1 & 1 & \infty & 3 \\ 1 & 6 & 6 & \infty \end{pmatrix}.
\end{array}$$

Ein branch-and-bound Verfahren beruht für dieses Problem auf dem Gedanken, dass eine Stadt, z.B. s_1 an der wir die Rundreise beginnen wollen, entweder nach s_2, nach s_3 oder nach s_4 verlassen werden kann. Den entstehenden Teilprobleme werden nun passende Entfernungsmatrizen zugeordnet, bei denen alle nicht mehr benutzbaren, gerichteten Kanten den Wert ∞ erhalten. Weil der Ort s_1 über s_2 verlassen wurde, kann diese Straße, die wie alle anderen keine Einbahnstraße ist, nicht mehr benutzt werden, und der Eintrag $d_{2,1}$ muss jetzt auf den Wert ∞ gesetzt werden. Da aber s_2 auch nur von s_1 aus erreicht wird, müssen auch die Einträge $d_{3,2}$ und $d_{4,2}$ nun ∞ lauten. Nun wurde s_1 ausschließlich nach s_2 verlassen, und die Straßen nach s_3 oder s_4 können nicht mehr von s_1 aus gewählt werden, so dass auch $d_{1,3}$ und $d_{1,4}$ den Eintrag ∞ erhalten müssen. Die untere Schranke für die kürzeste Rundreise im verbliebenen Graphen (Teilproblem) wird wieder auf die bekannte Weise bestimmt.

Wir erhalten beim Verlassen von s_1 nach s_2 die Matrix $P_{1,2}$:

$$\begin{array}{c} P_{1,2} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{array} \begin{array}{c} \begin{array}{cccc} s_1 & s_2 & s_3 & s_4 \end{array} \\ \begin{pmatrix} \infty & 3 & \infty & \infty \\ \infty & \infty & 3 & 6 \\ 1 & \infty & \infty & 3 \\ 1 & \infty & 6 & \infty \end{pmatrix} \end{array}$$

mit $d := \frac{d_{in}+d_{out}}{2} = \frac{10+8}{2} = 9$.

Beim Verlassen von s_1 nach s_3 erhalten wir die Matrix $P_{1,3}$:

$$\begin{array}{c} P_{1,3} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{array} \begin{array}{c} \begin{array}{cccc} s_1 & s_2 & s_3 & s_4 \end{array} \\ \begin{pmatrix} \infty & \infty & 2 & \infty \\ 4 & \infty & \infty & 6 \\ \infty & 1 & \infty & 3 \\ 1 & 6 & \infty & \infty \end{pmatrix} \end{array}$$

mit $d := \frac{d_{in}+d_{out}}{2} = \frac{7+8}{2} = 7,5$.

und beim Verlassen von s_1 nach s_4 erhalten wir die Matrix $P_{1,4}$:

$$\begin{array}{c} P_{1,4} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{array} \begin{array}{c} \begin{array}{cccc} s_1 & s_2 & s_3 & s_4 \end{array} \\ \begin{pmatrix} \infty & \infty & \infty & 7 \\ 4 & \infty & 3 & \infty \\ 1 & 1 & \infty & \infty \\ \infty & 6 & 6 & \infty \end{pmatrix} \end{array}$$

mit $d := \frac{d_{in}+d_{out}}{2} = \frac{12+17}{2} = 14,5$.

Da $P_{1,3}$ die kleinste untere Schranke besitzt, versprechen wir uns für den Fortgang den meisten Erfolg und setzten mit dieser Teillösung fort. Analog zu den eben gebildeten Alternativen entwickelt man die folgenden Entfernungstabellen mit ihren unteren Schranken:

Beim Verlassen von s_3 nach s_2 erhalten wir die Matrix $P_{1,3,2}$:

$$\begin{array}{cc} P_{1,3,2} & \begin{array}{cccc} s_1 & s_2 & s_3 & s_4 \end{array} \\ \begin{array}{c} s_1 \\ s_2 \\ s_3 \\ s_4 \end{array} & \begin{pmatrix} \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 6 \\ \infty & 1 & \infty & \infty \\ 1 & \infty & \infty & \infty \end{pmatrix} \end{array}$$

mit $d := \frac{d_{in}+d_{out}}{2} = \frac{10+10}{2} = 10$,

und beim Verlassen von s_3 nach s_4 erhalten wir die Matrix $P_{1,3,4}$:

$$\begin{array}{cc} P_{1,3,4} & \begin{array}{cccc} s_1 & s_2 & s_3 & s_4 \end{array} \\ \begin{array}{c} s_1 \\ s_2 \\ s_3 \\ s_4 \end{array} & \begin{pmatrix} \infty & \infty & 2 & \infty \\ 4 & \infty & \infty & \infty \\ \infty & \infty & \infty & 3 \\ \infty & 6 & \infty & \infty \end{pmatrix} \end{array}$$

mit $d := \frac{d_{in}+d_{out}}{2} = \frac{15+15}{2} = 15$.

Da das Minimum dieser unteren Schranken größer ist als eine der vorangegangenen, nicht weiter verfolgten Alternativen, wird nun zu der nächst besten Alternative, der Matrix $P_{1,2}$ mit der Gewichtung 9 zurückgesprungen (*back-tracking*), denn es wäre möglich, so eine bessere Lösung zu finden.

Wir erhalten nun beim Verlassen von s_2 nach s_3 die Matrix $P_{1,2,3}$:

$$\begin{array}{cc} P_{1,2,3} & \begin{array}{cccc} s_1 & s_2 & s_3 & s_4 \end{array} \\ \begin{array}{c} s_1 \\ s_2 \\ s_3 \\ s_4 \end{array} & \begin{pmatrix} \infty & 3 & \infty & \infty \\ \infty & \infty & 3 & \infty \\ \infty & \infty & \infty & 3 \\ 1 & \infty & \infty & \infty \end{pmatrix} \end{array}$$

mit $d := \frac{d_{in}+d_{out}}{2} = \frac{10+10}{2} = 10$,

und beim Verlassen von s_2 nach s_4 erhalten wir die Matrix $P_{1,2,4}$:

$$\begin{array}{c c} P_{1,2,4} & \begin{array}{cccc} s_1 & s_2 & s_3 & s_4 \end{array} \\ \begin{array}{c} s_1 \\ s_2 \\ s_3 \\ s_4 \end{array} & \begin{pmatrix} \infty & 3 & \infty & \infty \\ \infty & \infty & \infty & 6 \\ 1 & \infty & \infty & \infty \\ \infty & \infty & 6 & \infty \end{pmatrix} \end{array}$$

mit $d := \frac{d_{in}+d_{out}}{2} = \frac{16+16}{2} = 16$.

Aus der Distanzmatrix $P_{1,2,3}$ liest man leicht folgende Rundreise mit der Länge 10 ab: $s_1 \xrightarrow[3]{} s_2 \xrightarrow[3]{} s_3 \xrightarrow[3]{} s_4 \xrightarrow[1]{} s_1$. Da die unteren Schranken für $P_{1,2,4}$ (und $P_{1,3,4}$, sowie $P_{1,3,2}$) mit 16 (bzw. 15 und 10) nicht kleiner sind, können ausgehend von diesen keine besseren Rundreisen gefunden werden und obige Rundtour ist optimal. Jedoch bietet die Matrix $P_{1,3,2}$ die Chance für eine Rundreise gleicher Länge! Tatsächlich ist die alternative Tour $s_1 \xrightarrow[2]{} s_3 \xrightarrow[1]{} s_2 \xrightarrow[6]{} s_4 \xrightarrow[1]{} s_1$ möglich.

Branch-and-bound Verfahren sind im schlechtesten Falle exponentielle Verfahren, denn es ist denkbar, dass die optimale Lösung erst gefunden wird, wenn alle möglichen Verzweigungsmöglichkeiten abgesucht worden sind. Ein vollständiger binärer Verzweigungsbaum der Tiefe n hat aber 2^n Blätter. Auch bei der obigen branch-and-bound Lösung des Rundreiseproblems bringt die detaillierte Analyse nicht viel, denn wir wissen aus der NP-Vollständigkeit dieses Problems (siehe Kapitel 8), dass es zur Zeit kein deterministisches polynomiales Verfahren zu dessen Lösung gibt.

6.7 Heuristiken und Approximationsverfahren

Der Begriff "Heuristik" stammt von dem griechischen Mathematiker Archimedes ($\approx$ 285 - 212 v.Z.) und erinnert an seinen berühmten Ausspruch:

> $E\upsilon\rho\eta\kappa\alpha$ - Heureka! ("Ich hab's gefunden")
> (heurisko (griech.): ich finde).

Eine *Heuristik* (ein *heuristisches Verfahren*) ist ein Algorithmus, der (im allgemeinen recht gute) Lösungen für Problem-Beispiele erzeugt, aber einer exakten Analyse nicht (oder nur sehr schwer) zugänglich ist. ("Die Verfahren funktionieren, aber man weiß nicht genau warum!").

Viele lokale Suchverfahren sind Heuristiken. Weitere Heuristiken sind:

- Simulated Annealing,
- Tabu Search,
- Genetische Algorithmen.

Approximationsverfahren sind Algorithmen zur Erzeugung von Lösungen, die i.a. nur suboptimal sind, aber der optimalen Lösung (nachweislich) nahekommen.

Wir erinnern noch einmal an die Definition eines diskreten Optimierungsproblems (DOP):

Ein ein diskretes Optimierungsproblem P ist gegeben durch:

$\mathcal{I}$ = Menge der Instanzen (Beispiele),

$\mathcal{L}$ = Menge der Lösungen,

w = Wertfunktion (Zielfunktion): $w : \mathcal{L} \to I\!R$.

$s(I)$ sei eine Lösung für $I \in \mathcal{I}$.

$s^*(I)$ heißt *optimale Lösung* für I : $\Leftrightarrow$

$$\text{Minimum-Problem: } w(s^*(I)) \leq w(s(I)) \text{ für alle Lösungen } s(I),$$

$$\text{Maximum-Problem: } w(s^*(I)) \geq w(s(I)) \text{ für alle Lösungen } s(I).$$

Kurz: $OPT(I) := w(s^*(I))$.

Ein Algorithmus $\mathcal{A}$ heißt *Polynomial-Zeit- Approximationsalgorithmus* (PZ-Algorithmus) für P genau dann, wenn $\mathcal{A}$ für jedes $I \in \mathcal{I}$ in polynomialer Zeit eine Lösung $s_A(I)$ von I erzeugt:

$$\mathcal{A}(I) := w(s_A(I)).$$

Gütemaß für einen Approximationsalgorithmus:

<u>P Min-Problem:</u>

$$R_{\mathcal{A}}(I) = \frac{\mathcal{A}(I)}{OPT(I)}$$

<u>P Max-Problem:</u>

$$R_{\mathcal{A}}(I) = \frac{OPT(I)}{\mathcal{A}(I)}$$

Es gilt stets:

$$1 \leq R_{\mathcal{A}}(I) \leq +\infty$$

für alle $I \in \mathcal{I}$ und alle Algorithmen für P. $\mathcal{A}$ heißt ein *Optimierungsalgorithmus* für P genau dann, wenn gilt:

$$\mathcal{A}(I) = OPT(I) \text{ für alle } I \in \mathcal{I}.$$

Wir betrachten:

$$R_{\mathcal{A}} := \inf\{c \mid R_{\mathcal{A}}(I) \leq c \text{ für alle } I \in \mathcal{I}\}$$

$$\rightarrow 1 \leq R_{\mathcal{A}} \leq +\infty.$$

$R_{\mathcal{A}}$ kann nur in seltenen Fällen exakt bestimmt werden. Aber man kann $R_{\mathcal{A}}$ zuweilen abschätzen.

Man kann nun diskrete Optimierungsprobleme (DOP) einteilen in:

- *gut-approximierbare DOP P*: es gibt für P Algorithmen $\mathcal{A}$ mit $R_{\mathcal{A}}$ "nahe" bei 1,
- *schlecht-approximierbare DOP P*: das sind solche Probleme, für die es *nachweislich* keine guten Approximationsalgorithmen gibt.

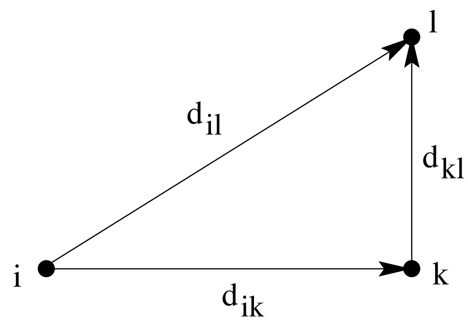
Beispiel: ΔTSP (D symmetrisch)

ΔTSP ist ein Spezialfall des TSP, und zwar soll für D die Dreiecksungleichung gelten:

$$D = (d_{ik}),$$

und für alle i,k,l mit $1 \leq i, k, l \leq n$ sei:

$$d_{ik} + d_{kl} \geq d_{il}. \tag{14}$$

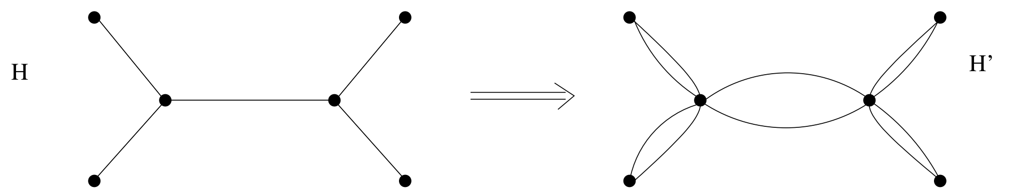


Algorithmus für ΔTSP:

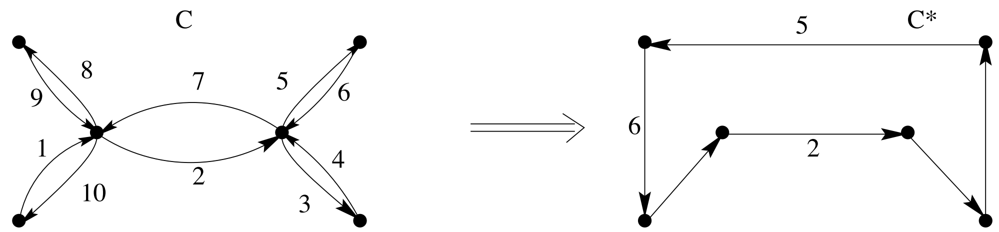
Gegeben: $I \in \Delta TSP$,
D = Entfernungsmatrix (n Städte),
$G = (V, D)$ zugehöriger Graph,
$V = \{s_1, s_2, \ldots, s_n\}$,
$E = \mathcal{P}_2(V)$
$D = E \to \mathbb{R}^+$ erfülle die Δ-Ungleichung (3.1).

1. Bestimme ein Minimalgerüst H von G.

2. Verdoppelung der Kanten von H:
   zu jeder Kante in H wird ein zweite (parallele) Kante hinzugefügt (es entsteht ein **Multigraph** - mit Mehrfachkanten):

   H' ist ein Eulergraph (d.h. alle Knotenpunkte haben eine gerade Valenz)[8]. Daher besitzt H' einen Eulerkreis C'.
   (Ein **Eulerkreis** ist eine geschlossene Kantenfolge, die jede Kante des Graphen genau einmal enthält. - Analog zu den Knoten bei Hamiltonkreisen!.
   Bestimmung des Eulerkreises C' in einem Eulergraphen H' (kann in $0(m)$ Zeit geschehen).

3. Erzeuge aus C' durch Anwendung der Dreiecksungleichung (3.1) einen Hamiltonkreis C^* von G, indem bereits erfaßte Knotenpunkte übersprungen werden.

[8]Anzahl der inzidenten Kanten, vergl. [BLa96]).

4. STOP: Output: C^* mit $l(C^*) = \mathcal{A}(I)$.

Es gilt:

$$l(H) \leq OPT(I) \leq l(C^*) \leq l(C') = 2 \cdot l(H).$$

Folglich ist

$$OPT(I) \leq \mathcal{A}(I) \leq 2 \cdot OPT(I),$$

$$\text{also } R_A(I) = \frac{\mathcal{A}(I)}{OPT(I)} \leq 2 \text{ und somit } R_A \leq 2.$$

Dieses Ergebnis kann auch wohl kaum verbessert werden, denn es gelten die folgenden Ergebnisse.

So gilt zum Beispiel (vergleichen Sie die Definition 8.12 von NP-Vollständigkeit in Kapitel 8) der folgende Satz:

6.2 Theorem

Das ΔTSP-Problem ist NP-vollständig.

$\diamond$

Es gilt weiterhin der Satz:

6.3 Theorem

Angenommen, für jedes $\epsilon > 0$ gibt es einen PZ-Approximationsalgorithmus $\mathcal{A}$ für TSP, der für jede symmetrische Distanzmatrix $D \in I\!N^{n \times n}$ in polynomialer Zeit eine Tour T_A liefert mit der Länge $\mathcal{A}_{T_A}(I)$, für die $R_{\mathcal{A}} \leq 1 + \epsilon$ gilt, dann ist HAMILTONKREIS $\in \mathcal{P}$, d.h. dann ist $\mathcal{P} = \mathcal{NP}$.

$\diamond$

Also ist TSP vermutlich (d.h. unter der Annahme, dass $\mathcal{P} \neq \mathcal{NP}$ gilt) ein schlecht-approximierbares Problem.

Beispiel für ein gut-approximierbares Problem:

Für das Rucksackproblem gibt es einen PZ-Algorithmus $\mathcal{A}$ mit $R_{\mathcal{A}} \leq 1 + \epsilon$ für beliebige $\epsilon \geq 0$.

7 Methoden zur Analyse von Algorithmen

Wenn wir konkrete Algorithmen, wie Sortier-, Such- oder Graphenalgorithmen, nach ihrer „Güte" beurteilen sollen, so gibt es dafür keine einfachen Antworten, denn die Beurteilungskriterien sind nicht immer klar und oft subjektiv. Einigen ist zunächst nur schnelles Arbeiten wichtig, anderen das auf lange Sicht hin Bedeutsamere: leichte Änderbarkeit, offensichtliche Klarheit und bewiesene Korrektheit.

Am objektivsten ist wohl die Angabe der Laufzeit und die des benötigten Speicherplatzes, auch wenn dies nur zwei von vielen wichtigen Kriterien sind. Natürlich ist exaktes Zeitverhalten, ausgedrückt in Sekunden und Minuten, kaum aussagekräftig, denn die Laufzeit hängt immer auch von dem Computer und dessen Befehlssatz ab, auf dem ein Algorithmus implementiert ist, von der Programmiersprache und dem verwendeten Compiler. Damit ein Algorithmus nicht nur für spezielle Eingaben effizient implementiert wird, muss die Analyse alle möglichen Eingaben als gleichverteilt annehmen und dann die mittlerer Laufzeit (average case) bestimmen, aber auch die schlechteste Laufzeit bei einer möglichen Eingabe (*worst case*). Die Analyse der mittleren Laufzeit ist nicht immer leicht, und letztendlich auch nicht vollständig befriedigend, denn die mathematische Gleichverteilung der Eingaben hat mit der in den Anwendungen vorliegenden (meist eben leider unbekannten) Verteilung oft nicht viel zu tun!

Wie schon vorher auch, wird eine abstrakte Maschine, der die üblicherweise benutzten Datenstrukturen zugänglich sind, als Vergleichsbasis herangezogen und die Laufzeit der Algorithmen als Funktion der Eingabegröße mit der Landau'schen Notation (siehe Kapitel 1, Abschnitt 1.7) beschrieben. Andere formale Modelle, wie die Turing-Maschine treten eher in den Hintergrund, da sie die realen Implementationen nicht gut genug modellieren. Allenfalls die in Kapitel 4 vorgestellte Register-Maschine wäre ein Kandidat für einige Algorithmenimplementationen. Ziel dabei ist es in der Regel, verschiedene Verfahren für den selben Zweck zu vergleichen und zwischen diesen auszuwählen.

Bei der asymptotischen Analyse von Algorithmen und Netzen ist die Hauptaufgabe, die Zeitkomplexität (oder andere Komplexität) $T(n)$ – wobei n die Größe der Eingaben ist – in der Form

$$T(n) \in f(n) + \Theta(g(n))$$

auszudrücken, wobei die beiden Funktionen $f(n)$ und $g(n)$ einfach sind.

Zum Beispiel weiß man, dass *Mergesort* eine *„worst-case-“* Zeitkomplexität von $O(n \log n)$ besitzt und *Insertsort* eine solche von $\Theta(n^2)$ – das heißt, Mergesort ist asymptotisch schneller als Insertsort.

Manchmal, beispielsweise für Insertsort, ist es möglich, die exakte Zeit für einen gegeben Computer festzustellen, aber das ist meistens nicht die Mühe wert.

Asymptotische Analyse von Algorithmen hat ihre positiven Seiten, aber auch Schwierigkeiten.

1. Für die asymptotische Analyse eines Algorithmus braucht man oft nur die Hauptidee des Algorithmus kennenzulernen.

   Zum Besipiel reicht es aus zu wissen, dass Mergesort ein „*divide-and-conquer*“-Algorithmus ist, der ein Problem auf zwei Probleme der halben Größe reduziert (mit linearer Zeit für *decomposition* und *composition*), um zu wissen, dass Mergesort ein $\Theta(n \log n)$-Algorithmus ist.

   Ähnlich genügt es zu wissen, dass **Insertsort** ein solcher Sortieralgorithmus ist, für den

   > nach dem $i - ten$ Zyklus die ersten i Elemente der zu sortierenden Folge schon sortiert sind und im $i - ten$ Schritt das $i - te$ Element der Folge in die richtige Position gebracht wird.

   $$\underbrace{a_1 \ldots\ldots a_{i-1}}_{sortiert}\; a_i \ldots\ldots a_n \qquad\qquad i - ter \text{ Zyklus}$$

2. Man muss mit der asymptotischen Analyse vorsichtig sein – besonders, wenn die Komplexität nicht nur vom Umfang der Eingabedaten abhängt.

   Für die Zeitkomplexität des Insertsort gilt zum Beispiel:

   $$\begin{array}{rcl} T(n) & \in & \Omega(n) \\ T(n) & \in & O(n^2) \end{array}$$

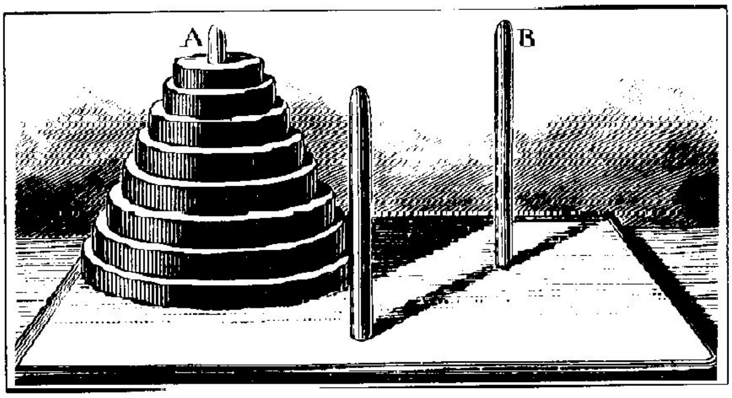
   Kann man nun schreiben „$T(n) \in \Omega(n^2)$“ ?
   Nein, denn es gibt Eingabedaten, für die Insertsort $\Theta(n)$ Zeit braucht. Aber man kann sagen, dass für die *worst-case*-Zeitkomplexität des Insertsort gilt:

   $$T_{worst}(n) \in \Omega(n^2)$$

7.1 Rekurrenzgleichungen

Wir werden Methoden kennenlernen, wie man einige Rekurrenzgleichungen lösen kann, um geschlossene Formeln für die n-ten Elemente zu erhalten. Manchmal kann man die exakte Lösung nicht finden; aber oft braucht man diese auch gar nicht! Meistens genügt eine gute Annäherung.

Beispiel 1: [Türme-von-Hanoi-Problem (Lucas 1883)]



Aufgabe: Seien drei Stangen gegeben und n Scheiben, die der Größe nach sortiert auf Stange A liegen. Die Aufgabe ist es, den ganzen Turm von Scheiben auf Stange C zu verschieben, dabei nur eine Scheibe in einem Zeitschritt zu bewegen und nie eine größere Scheibe auf eine kleinere zu legen.

Es gibt einen einfachen rekursiven Algorithmus:

1. Verschiebe die $n-1$ oberen Scheiben von A auf B.
2. Verschiebe die größte Scheibe von A auf C.
3. Verschiebe alle $n-1$ Scheiben von B auf C.

Verschiebungs-Analyse: Sei $T(n)$ die Anzahl der Verschiebungen, die obiger Algorithmus braucht, um n Scheiben zu verschieben. Es gilt

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n-1)+1 & \text{falls } n > 1 \end{cases}$$

Wichtige Frage: Kann man das schneller tun? *Antwort:* Nein.

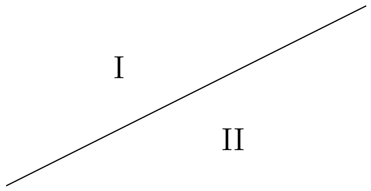
Beispiel 2:

Wir wollen die Frage beantworten, in wieviele Gebiete n Geraden die euklidsche Ebene maximal zerteilen können.

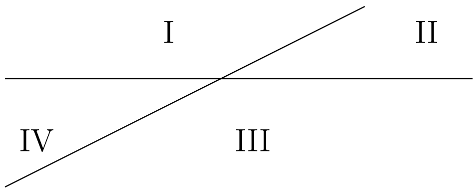
Wir beginnen mit Antwort für die leicht zu ermittelnden Fälle mit wenig Geraden:

Ist keine Gerade vorhanden, $n = 0$, so bleibt das Gebiet unverändert. Wenn wir die Zahl der jeweils maximal möglichen Gebiete mit L_n bezeichnen, so gilt also $L_0 = 1$.

Bei einer Geraden sind offensichtlich zwei Gebiete zu erwarten, d.h. $L_1 = 2$.

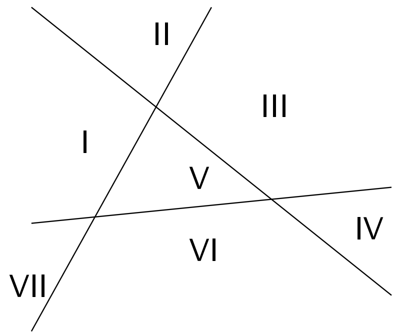


Für $n = 2$ ergibt sich folgendes Bild:



also $L_2 = 4$.

Die erste Vermutung $L_n = 2^n$, die für $n = 0, 1, 2$ stimmt, wird durch $L_3 = 7$ sofort widerlegt:



Da jede neue Gerade, die zu keiner anderen parallel liegt, alle vorherigen Geraden einmal schneidet, kann die n-te Gerade die bisherigen $n-1$ Geraden in höchstens $n-1$ verschiedenen Punkten schneiden und dann die bisherigen L_{n-1} Gebiete um höchstens n erweitern (Alle Gebiete auf einer Seite der neuen Geraden und dabei dann vor, zwischen und hinter den Schnittpunkten). Wenn die n-te Gerade nicht durch einen früheren Schnittpunkt geht, so wird die Maximalzahl erreicht, und es ergibt sich die Rekurrenz $L_n := L_{n-1} + n$ für die gesuchte Zahl L_n mit dem Anfangswert $L_0 := 1$.

Um nun eine geschlossene Formel für L_n zu gewinnen, wickeln wir die ersten Rekurrenzen ab:

$$\begin{array}{rcl} L_0 & = & 1 \\ L_1 & = & L_0 + 1 = 1 + 1 = 2 \\ L_2 & = & L_1 + 2 = (L_0 + 1) + 2 = 1 + 1 + 2 = 4 \\ L_3 & = & (L_1 + 2) + 3 = ((L_0 + 1) + 2) + 3 \\ & = & 1 + 1 + 2 + 3 = 7 \end{array}$$

Die offensichtliche Vermutung ist also

$$L_n \quad = \quad 1 + \sum_{i=1}^{n} i \tag{15}$$

Diese Vermutung muss aber noch formal bewiesen werden, was wir durch vollständige Induktion tun werden.

Die **Verankerung** für $n = 0$ ergibt sich wie gewünscht:

$$L_0 = 1 + \sum_{i=1}^{0} i = 1$$

Die **Induktionsannahme** lautet

$$L_m = 1 + \sum_{i=1}^{m} i \quad \text{für ein festes} \quad m \in I\!N$$

Der **Induktionsschritt** ergibt

$$\begin{array}{rcll} L_{m+1} & = & L_m + (m+1) & \text{entsprechend der Rekurrenz} \\ & = & 1 + \sum_{i=1}^{m} i + (m+1) & \text{nach Induktionsannahme} \\ & = & 1 + \sum_{i=1}^{m+1} i & \text{trivial.} \end{array}$$

Damit ist die Formel $L_n = 1 + \sum_{i=1}^{n} i$ bewiesen. Eine richtig geschlossene Formel ist diese Summation jedoch noch nicht. Mit Hilfe von $\sum_{i=1}^{m} i = \frac{n(n+1)}{2}$ (vergl. Kapitel 1, Theorem 1.2) erhält jede(r) nun die wirklich geschlossene Formel für L_n:

$$L_n = 1 + \frac{n(n+1)}{2} \tag{16}$$

Wichtige Empfehlung: Immer wenn man eine Rekurrenzgleichung lösen muss, ist es nützlich, zuerst die Werte der gesuchten Funktion für kleine Argumente zu berechnen. Diese Werte können helfen

1. die Lösung zu finden,
2. die Lösung zu verifizieren.

Wir zeigen dies zunächst an Beispiel 2, der Rekurrenzgleichung für die Türme von Hanoi:

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n-1) + 1 & \text{falls } n \geq 2 \end{cases}$$

Tabellarische Darstellung:

n	1	2	3	4	5	6	7	8	9	10
$T(n)$	1	3	7	15	31	63	127	255	511	1023

Schätzung: $T(n) = 2^n - 1$? (das soll noch bewiesen werden!)

7.2 Vollständige Induktion

Nachdem man eine Lösung einer Rekurrenzgleichung geschätzt hat, ist vollständige Induktion oft eine gute Methode, die Richtigkeit der Schätzung zu beweisen.

Für die Türme von Hanoi mit der Rekurrenzgleichung

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n-1) + 1 & \text{falls } n > 1 \end{cases} \tag{17}$$

war die Lösung $T(n) = 2^n - 1$ geschätzt worden.

Nun beweisen wir, dass $T(n) = 2^n - 1$ wirklich eine Lösung ist.

Beweis

Induktionsanfang: Für $n = 1$ gilt $T(1) = 2^1 - 1 = 1$.

Induktionsschritt: Nach der *Induktionsvoraussetzung* sei für ein gegebenes $n \geq 1 \quad T(n) = 2^n - 1$ eine Lösung von Gleichung (17).
Induktionsbehauptung:

$$T(n+1) = 2T(n) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 1$$

Das bedeutet, dass für alle $n \geq 1$ gilt: $T(n) = 2^n - 1$.

$\square$

7.3 Reduzierung der Rekurrenzgleichungen auf Summen

Dies ist eine Methode, eine Lösung zu finden, wenn man aus den endlich vielen ersten Werten keine Vermutung hatte ableiten können, die beweisbar eine Lösung darstellte.

Beispiel 1: [Lösung]
Für die Rekurrenzgleichung (des Türme-von-Hanoi-Problems)

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T(n-1) + 1 & \text{falls } n > 1 \end{cases}$$

verfährt man schrittweise (sogenanntes „Abwickeln der Rekursion”).

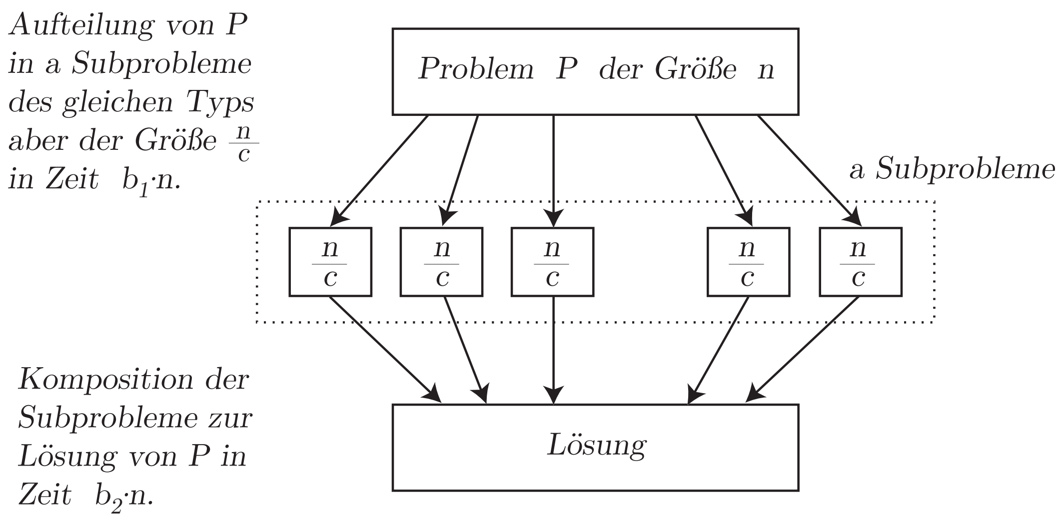
$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2T(n-2) + 1) + 1 = 4T(n-2) + 2 + 1 \\ &= 4(2T(n-3) + 1) + 2 + 1 \\ &= 2^3 T(n-3) + 2^2 + 2^1 + 2^0 \\ &= 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i \\ &= 2^{n-1} T(1) + \sum_{i=0}^{n-2} 2^i \\ &= \sum_{i=0}^{n-1} 2^i \quad \text{(geometrische Reihe)} \\ &= \frac{2^n - 1}{2 - 1} = 2^n - 1 \end{aligned}$$

Beispiel 3: [*divide-and-conquer*]

Die „*divide-and-conquer*“–Methode (Teile und Herrsche) ist vielleicht die wichtigste Methode, um algorithmische Probleme zu lösen. In Kapitel 6 haben wir den Mergesort angesprochen (siehe Abschnitt 6.2).

Sehr oft kann man ein algorithmisches Problem P der Größe n mit folgender Methode lösen:

1. Man zerteilt das Problem P in der Zeit $b_1 \cdot n$ in a gleiche Subprobleme der Größe $\frac{n}{c}$.
2. Man löst die Subprobleme rekursiv mit derselben Methode.
3. In der Zeit $b_2 \cdot n$ werden die Lösungen der Subprobleme zur Lösung des Originalproblems zusammengesetzt.



Frage: Wie ist die Zeitkomplexität $T(n)$ dieses Algorithmus?

$$T(n) = \begin{cases} d & \text{falls } n = 1 \\ aT(\frac{n}{c}) + b_1 n + b_2 n & \text{falls } n > 1 \end{cases}$$

Zur Analyse des „*divide-and-conquer*“–Algorithmus vereinfachen wir die Rekurrenz zunächst zu

$$T(n) = \begin{cases} b & \text{falls} \quad n = 1 \\ aT(\frac{n}{c}) + bn & \text{falls} \quad n > 1. \end{cases}$$

Dies ist möglich, da b_1 und b_2 für jedes Problem bekannt sind und konstant bleiben. Es gilt zudem $d \leq b_1 + b_2$, so dass $b := b_1 + b_2$ eine brauchbare Abschätzung ist.

Durch rekursives Einsetzen („Abwickeln“) erhalten wir:

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{c}\right) + bn = a\left(aT\left(\frac{n}{c^2}\right) + b\frac{n}{c}\right) + bn \\
&= a^2T\left(\frac{n}{c^2}\right) + bn\frac{a}{c} + bn \\
&= a^2\left(aT\left(\frac{n}{c^3}\right) + b\frac{n}{c^2}\right) + bn\frac{a}{c} + bn \\
&= a^3T\left(\frac{n}{c^3}\right) + bn\left(\frac{a}{c}\right)^2 + bn\frac{a}{c} + bn \\
&= a^kT\left(\frac{n}{c^k}\right) + bn \cdot \sum_{i=0}^{k-1}\left(\frac{a}{c}\right)^i
\end{aligned}$$

Wenn $n = c^k$ ist, so bricht das Verfahren bei $T(n) = a^k b + bn\sum_{i=0}^{k-1}(\frac{a}{c})^i$ ab, denn es ist $T(1) = b$.

Mit $n = c^k$ folgt $k = \log_c(n)$ und somit

$$\begin{aligned}
T(n) &= a^k b + bn\sum_{i=0}^{k-1}\left(\frac{a}{c}\right)^i \\
&= a^k b + bn\sum_{i=0}^{k}\left(\frac{a}{c}\right)^i - bn\left(\frac{a}{c}\right)^k \\
&= a^k b\left(1 - \frac{n}{c^k}\right) + bn\sum_{i=0}^{k}\left(\frac{a}{c}\right)^i \\
&= bn\sum_{i=0}^{\log_c(n)}\left(\frac{a}{c}\right)^i
\end{aligned}$$

Je nach dem Verhältnis von a zu c ergeben sich unterschiedliche Lösungen:

Fall 1, $a < c$: Die Reihe $\sum\limits_{i=0}^{\infty}\left(\frac{a}{c}\right)^i$ konvergiert, so dass $T(n)$ proportional zu n ist.

Fall 2, $a = c$: $T(n) = bn(\log_c(n) + 1)$, d.h. $T(n)$ ist „proportional“ zu $n\log(n)$.

Fall 3, $a > c$: $T(n)$ ist „proportional“ zu $n^{\log_c(a)}$.

Dann folgt

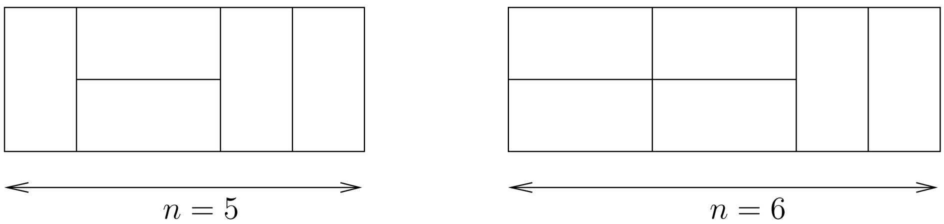
$$
\begin{aligned}
T(n) &= bn \sum_{i=0}^{\log_c(n)} \left(\frac{a}{c}\right)^i \\
&= bn \frac{\left(\frac{a}{c}\right)^{\log_c(n)+1} - 1}{\frac{a}{c} - 1} \\
&\approx bn \left(\frac{a}{c}\right)^{\log_c(n)} \\
&= bn \frac{a^{\log_c(n)}}{n} \\
&= ba^{\log_c(n)} = bn^{\log_c(a)}
\end{aligned}
$$

weil $a^{\log_c(n)} = n^{\log_c(a)}$ stets gilt.

Schlußfolgerung:
Die Zeitkomplexität eines *divide-and-conquer*–Algorithmus hängt nur von dem Verhältnis $\frac{a}{c}$ ab – und nicht von der Art des Problems oder dem Lösungsweg –, wenn der Zeitbedarf für die Zerlegung in Teilprobleme und die Zusammenfassung proportional zur Größe des Problems ist.

7.4 Reduktion auf algebraische Gleichungen

Für Platinenlayout sind Anordnungen von Chips der Maße $1*2$ cm auf einer „Bahn“ von 2 cm Höhe und bisher unbestimmter Länge n nötig. Mögliche Anordnungen sind z.B.:



Bezeichnen wir mit T_n die Anzahl der verschiedenen Anordnungen (*tilings*, Parkettierungen, Kachelungen) einer Fläche der Größe $2 * n$, so sehen wir, dass offensichtlich aus jeder Parkettierung einer Bahn der Fläche $2*(n-1)$ durch Voraussetzen eines Chips der Höhe 2 cm und Breite 1 cm eine Parkettierung der Bahn $2 * n$ entsteht. Durch Voraussetzen zweier waagerecht übereinander angeordneter Chips der Maße 1 cm $*$ 2 cm entsteht eine Parkettierung der Bahn $2*(n+1)$, die nicht durch Voraussetzen nur eines Chips aus einer Parkettierung der Bahn $2*n$ entstanden sein kann.

Es gilt also $T_n = T_{n-1} + T_{n-2}$ mit den Anfangswerten $T_0 := 1$ und $T_1 := 1$.

Auf gleiche Weise sind die *Fibonacci-Zahlen* (Leonardo Fibonacci, 1202) durch die folgenden Rekurrenzgleichungen definiert:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad \text{für } n > 1 \tag{18}$$

Wir suchen eine Lösung in der Form $F_n = r^n$, wobei r eine unbekannte Konstante ist. (Dass dieser Ansatz zum Erfolg führen muss, folgt aus allgemeineren Resultaten, auf die wir hier nicht eingehen wollen.)

Würde eine solche Lösung existieren, dann müßte gelten:

$$r^n = r^{n-1} + r^{n-2} \quad \text{für jedes } n > 1$$

und daraus folgt, dass entweder $r = 0$ oder $r^2 = r+1$. Diese Gleichung hat zwei Lösungen:

$$r_1 = \frac{1+\sqrt{5}}{2}, \qquad r_2 = \frac{1-\sqrt{5}}{2}$$

Eine allgemeine Lösung der ganzen Gleichung (18), die auch die Anfangsbedingungen erfüllt, hat dann die Form $\lambda r_1^n + \mu r_2^n$ wobei λ und μ Konstanten sind. Daraus folgt

$$\lambda + \mu = F_0 = 0, \qquad \lambda r_1 + \mu r_2 = F_1 = 1$$

Also gilt:

$$\lambda = -\mu = \frac{1}{\sqrt{5}}$$

und

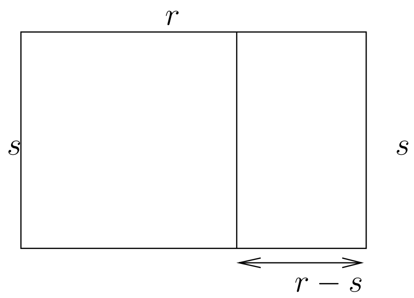
$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Wegen $\lim_{n\to\infty}\left(\frac{1-\sqrt{5}}{2}\right)^n = 0$ gilt:

$$F_n \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n \qquad \text{für } n \to \infty$$

Die Nullstellen $r_1 = \frac{1+\sqrt{5}}{2}$ und $r_2 = \frac{1-\sqrt{5}}{2}$ werden im Zusammenhang mit dem „goldenen Schnitt“ meist als $\Phi := r_1$ und $\hat{\Phi} := r_2$ bezeichnet.

Der goldene Schnitt ist die Zerteilung eines Rechtecks der Höhe s und der Länge r so, dass das übrigbleibende Rechteck der Höhe s und der Breite $r - s$ das gleiche Seitenverhältnis $x := \frac{r}{s}$ hat.



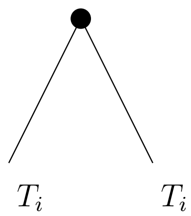
Damit dies gelingen kann, müssen r und s so gewählt sein, dass $\frac{r}{s} = \frac{s}{r-s}$ gilt. Daraus folgt $x = \frac{r}{s} = \frac{s}{r-s} = \frac{1}{x-1}$ oder $x^2 - x - 1 = 0$. $x = \Phi = \frac{1+\sqrt{5}}{2}$ wie auch $x = \hat{\Phi} = \frac{1-\sqrt{5}}{2}$ sind die Lösungen dieser Gleichung!. Eine Größe r wird nach dem goldenen Schnitt geteilt, wenn der Teil s das geometrische Mittel von r und $r - s$ ist: $s = \sqrt{r(r-s)}$.

Fast immer führen rekursive Definitionen auf Rekurrenzgleichungen und erfordern ähnliche Analysemethoden. Daher ist es gut zu wissen, dass es eine Unmenge von Beispielen von Rekursionen in Definitionen gibt:

7.1 Definition

Ausgeglichene binäre Bäume sind anschaulich durch folgende Rekursion erklärt:

$T_0 := \bullet$ und $T_{i+1} :=$

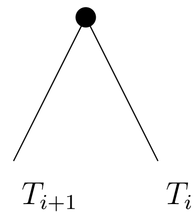


□

7.2 Definition

Fibonacci-Bäume sind rekursiv definiert durch

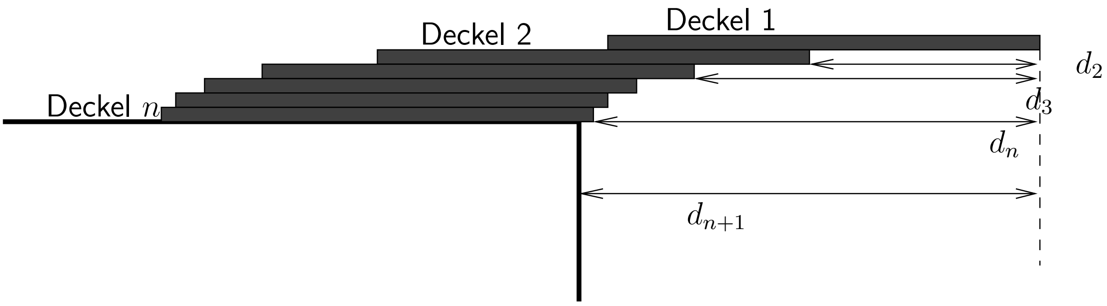
$T_0 := \bullet$, $T_1 := \bullet$ und $T_{i+2} :=$



□

Als letztes Beispiel einer Rekursion beginnen wir mit einer Fragestellung, die wohl gut beim gemeinschaftlichen Bier von Architekturstudierenden entstanden sein könnte:

Die Aufgabe besteht darin, Bierdeckel des Durchmessers 2 (in einer passenden Maßeinheit) an der Tischkante so übereinander zu stapeln, dass jeweils der größtmögliche Überhang d_i ensteht:



Es ist klar, dass der Schwerpunkt des Stapels der obersten k Deckel direkt über der Außenkante von Deckel $k+1$ liegen muss.

Offensichtlich gilt $d_1 = 0$ und $d_2 = 1 = \frac{d_1+1}{1}$, sowie in der Folge $d_3 = \frac{(d_1+1)+(d_2+1)}{2}$, $d_{k+1} = \frac{1}{k} \cdot \sum\limits_{i=1}^{k} (d_i + 1) = \frac{1}{k} \left(k + \sum\limits_{i=1}^{k} d_i \right) = 1 + \frac{1}{k} \cdot \sum\limits_{i=1}^{k} d_i$, also $k \cdot d_{k+1} = k + \sum\limits_{i=1}^{k} d_i$. Setzt man hier $k-1$ statt k ein, so gilt ebenso: $(k-1)\, d_k = k - 1 + \sum\limits_{i=1}^{k-1} d_i$ und Subtraktion beider Gleichungen ergibt die Auflösung der Summe: $kd_{k+1} - (k-1)\, d_k = 1 + d_k$ oder umgerechnet die einfache Rekurrenz

$$d_{k+1} = d_k + \frac{1}{k}$$

mit der offensichtlichen Lösung $d_{k+1} = \sum\limits_{i=1}^{k} \frac{1}{i}$.

Gibt es für die Größen d_k eine bessere Darstellung als die Summe, oder kann man diese durch eine geschlossene Form mit Funktionen nach unten und oben eingrenzen?

Sehr oft kann man exakte Antworten/Lösungen in einer geeigneten Form nicht finden oder sie sind zu kompliziert, um verwendbar zu sein. Häufig sogar, gerade bei der Analyse von Algorithmen und Programmen, sind sie nicht wichtig. Gute und verständliche Approximationen sind das, was man eher bräuchte und manchmal sind diese sogar gar nicht so schwer zu erhalten.

Im folgenden Abschnitt wird die Integralabschätzung als eine nützliche Methode zur sogenannten *asymptotischen Analyse* eingeführt.

Bei der asymptotischen Approximation einer Funktion $T(n)$ $(n \in \mathbb{Z})$ oder $A(x)$ $(x \in \mathbb{R})$ sucht man eine "Approximation im Limit" von $T(n)$ (oder $A(x)$) für $n \to \infty$ (oder $x \to a$, mit $a \in \mathbb{R}$).

Die Asymptotische Analyse ist sehr wichtig für *Komplexitätsanalyse* von Algorithmen, Programmen, Netzwerken.

In diesen Fällen steht n für die Größe der Eingaben, d.h die Länge ihrer Darstellung über einem festen Alphabet, und man sucht ein Verständnis für das Wachstum der Zeitkomplexität (bzw. Speicherkomplexität) wenn $n \to \infty$.

Beispiel 4: [Harmonische Zahlen]

Bei der Analyse von Algorithmen trifft man oft sogenannte *Harmonische Zahlen*

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i} \tag{19}$$

H_n: n-te Harmonische Zahl

Nach (19) kann man H_n für jedes einzelne n exakt berechnen! Das hilft jedoch noch nicht viel. Es gibt keine geschlossene Formel für H_n. Deshalb braucht man eine gute Approximation für H_n.

7.5 Integralabschätzung

Bei Summenformeln können wir diese oft nach oben und unten durch Integrale abschätzen. Wir illustrieren dies zunächst am Beispiel der Summe der Kubikzahlen $q_n := \sum\limits_{i=0}^{n} i^3$.

Wenn wir eine Abschätzung für das Wachstum von q_n für $n \to \infty$ suchen, so ist einfach aber unbefriedigend die folgende Abschätzung:

Jeder Summand i^3 der Summe q_n wird trivialer Weise durch n^3 dominiert.
Da es n von Null verschiedene Summanden gibt, folgt $q_n \leq n^4$ für $n \geq 1$.

Natürlich kann $q_n = \frac{n^2(n+1)^2}{4}$ fast jeder Formelsammlung entnommen werden, aber nicht für jede Summe muss dieses Vorgehen erfolgreich sein. Außerdem bekommt man so meist nur sehr große und unhandliche Formeln.

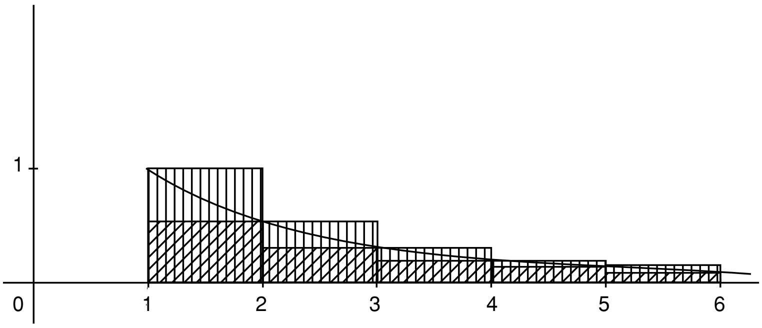


Abbildung 24: Skizze der Funktion H_n

Einer entsprechenden Grafik für die Funktion $f(x) := x^3$, die die Leser bitte selber zeichnen mögen, entnehmen wir dann leicht, dass folgendes gilt:
Die Fläche der über der Funktion $f(x) := x^3$ liegenden (bei der Funktion H_n senkrecht gestreiften) Rechtecke entspricht dann der Summe $q_n = \sum_{i=1}^{n} i^3$ und $q_n \geq \int_0^n x^3 dx$. Die Fläche der unter der Funktion liegenden schräggestreiften Rechtecke entspricht dann die Summe

$$\begin{aligned} q_{n-1} &= \sum_{i=1}^{n-1} i^3 \quad \text{mit} \\ q_{n-1} &\leq \int_0^n x^3 dx \end{aligned}$$

mit $\int_a^b f(x)dx = F(b) - F(a)$ für die Stammfunktion $F(x)$ mit $F'(x) = f(x)$ folgt dann

$$\int_0^n x^3 dx = \frac{n^4}{4} \leq q_n \leq n^3 + \int_1^n x^3 dx = n^3 + \frac{n^4 - 1}{4}$$

und

$$\frac{n^4}{4} \leq q_n \leq n^3 + \frac{n^4 - 1}{4}$$

ist eine sehr viel bessere Abschätzung von q_n.

Es gilt sogar allgemeiner für monotone Funktionen:

7.3 Theorem

Für $G(n) := \sum_{i=1}^{n} g(i)$ mit $g(i) \leq g(i+1)$ gilt:

$$\int_0^n g(x)dx \leq G(n) \leq \int_1^{n+1} g(x)dx$$

$\diamond$

Ist eine Funktion $g(x)$ monoton fallend, aber stets gilt $g(x) > 0$, so läßt sich das Verfahren der Abschätzung von $G(n) := \sum_{i=1}^{n} g(i)$ mit Hilfe von Integralen immer noch gut anwenden.

Die Harmonischen Zahlen, zum Beispiel, haben die Form $H_n = \sum_{i=1}^{n} \frac{1}{i}$, so dass $g(x) = \frac{1}{x}$ monoton fallend ist. Wegen $\int \frac{1}{x}dx = \ln(x)$ können wir mit dieser Methode eine Abschätzung von H_n vornehmen:
Betrachten wir die Funktion $g(x) = \frac{1}{x}$, so erhalten wir ein Diagramm, in dem die Rechtecke der entsprechenden Flächen $\frac{1}{x}$ eingezeichnet sind (vgl. Abbildung 24).

Es folgt nun $\int_1^n \frac{1}{x}dx < H_n < 1 + \int_1^n \frac{1}{x}dx$ und somit

$$\ln n < H_n < \ln n + 1 \qquad \text{falls } n > 1$$

und diese Approximation ist schon nicht schlecht. Viel besser aber ist die Approximation

$$H_n = \ln n + 0.5772156649 + \frac{1}{2n} - \frac{1}{12n^2} + \underbrace{\Theta(n^{-4})}_{\text{„wächst wie } n^{-4}\text{“}}$$

Beispiel 5:[Fakultät]

Die Fakultät $n!$ ist eine andere Funktion, die sehr wichtig für die Analyse von Algorithmen ist. Es gibt eine exakte Antwort, wieviel $n!$ ist:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

Aber das ist oft nicht gut genug. Es gibt keine geschlossene Formel für $n!$. Aber es gibt eine gute (Stirling, 1692 – 1700) Approximation

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Daraus folgt zum Beispiel:

$$\log(n!) \in \Theta(n \log n)$$

7.6 Schnelle Multiplikation

Wieviel Zeit braucht ein Mensch (ein Computer), um zwei n-stellige ganze Zahlen zu multiplizieren, wenn er den gewöhnlichen Schulalgorithmus benutzt?

$$\begin{array}{cccccccccccc}
 & & & & a_1 & a_2 & \dots & a_n & \times & b_1 \; b_2 \; \dots \; b_n & \\
 & & & & \bullet & \bullet & \dots & \bullet & & & \\
 & & & \bullet & \bullet & \dots & \bullet & & & & \\
 & & \vdots & & & \vdots & & & & n \text{ Zeilen} & \\
\bullet & \bullet & \bullet & & & \dots & \bullet & \bullet & & 2n \text{ Stellen} &
\end{array}$$

Die exakte Antwort kann für jeden Menschen/Computer unterschiedlich sein, aber für alle gilt: *Man braucht k^2-mal mehr Zeit, wenn man k-mal grössere Zahlen multiplizieren will.* Das bedeutet, dass man $O(n^2)$ Zeit braucht, um n-stellige ganze Zahlen mit dem Schulalgorithmus zu multiplizieren.

Wir haben in Kapitel 6 die rekursive Technik zur genaueren Multiplikation in Beispiel 3 auf Seite 104 kennengelernt. Dort wurde das Problem der Multiplikation der n-stelligen Zahlen in $O(n)$ Zeit auf das Problem von drei Multiplikationen der $\frac{n}{2}$-stelligen Zahlen reduziert.

Aus der Analyse der Zeitkomplexität der „divide-and-conquer"-Methode folgt dann, dass die Zeitkomplexität dieses Multiplikations-Algorithmus gerade $O\left(n^{1.585}\right)$ ist. (Bitte rechnen Sie das nach!) Ähnliche "Tricks" erlaubten V. Strassen beim Matrizenprodukt die Anzahl der Multiplikationen gegenüber der Standardmethode mit Hilfe einer geschickten Zerlegung zu reduzieren.

Um zwei Matrizen $A = \{a_{ij}\}$ und $B = \{b_{ij}\}$ des Grades n zu multiplizieren, $A \times B = C = \{c_{ij}\}$, braucht man mit dem (Schul-)Algorithmus

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$T(n) = 2n^3 - n^2$ arithmetische Operationen. Einfacher und oft hinreichend ist es, zu sagen, dass $T(n) \in \Theta(n^3)$ („$T(n)$ wächst etwa so wie n^3").

Auf Strassen (1969) geht ein Verfahren zurück, was ähnlich der Rechnung im Beispiel zur multi-precision-Multiplikation zweier Zahlen, durch geschickte Verwendung von Zwischenergebnissen die Matrixmultiplikation von $2 \times 2-$Matrizen mit nur 7 (statt 8) Multiplikationen und dafür 18 (wesentlich kostengünstigeren) Addionen auskommt. In der Verwendung eines *divide-and-conquer*-Verfahrens wird so ein viel besseres Ergebnis erziehlt, welches wir hier zum Abschluss vorstellen werden.

Zur Berechnung der 4 Zahlen $c_{1,1}, c_{1,2}, c_{2,1}, c_{2,2}$ im Produkt

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix} = \begin{pmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{pmatrix}$$

werden zunächst die 7 Größen $d_i, 1 \leq i \leq 7$ berechnet:

$$\begin{aligned} d_1 &:= (a_{1,2} - a_{2,2}) * (b_{2,1} + b_{2,2}) \\ d_2 &:= (a_{1,1} + a_{2,2}) * (b_{1,1} + b_{2,2}) \\ d_3 &:= (a_{1,1} - a_{2,1}) * (b_{1,1} + b_{1,2}) \\ d_4 &:= (a_{1,1} + a_{1,2}) * b_{2,2} \\ d_5 &:= a_{1,1} * (b_{1,2} - b_{2,2}) \\ d_6 &:= a_{2,2} * (b_{2,1} - b_{1,1}) \\ d_7 &:= (a_{2,1} + a_{2,2}) * b_{1,1} \end{aligned}$$

Das geschieht mit 7 Multiplikationen und 10 Additionen (bzw. Subtraktionen). Mit 8 weiteren Additionen (bzw. Subtraktionen) erhält man nun das gesuchte Ergebnis:

$$\begin{aligned} c_{1,1} &:= d - 1 + d_2 - d_4 + d_6 \\ c_{1,2} &:= d_4 + d_5 \\ c_{2,1} &:= d_6 + d_7 \\ c_{2,2} &:= d_2 - d_3 + d_5 - d_7 \end{aligned}$$

Ein Algorithmus, der dieses Verfahren mit minimalem Aufwand in der *divide-and-conquer*-Methode anwendet, benötigt für Matrizen der Größe 2^n

$$f(n) := \begin{cases} 7 \cdot f(n-1) & \text{falls} \quad n \geq 1 \\ 1 & \text{falls} \quad n = 0. \end{cases}$$

Multiplikationen. Es folgt sofort $f(n) = 7^n$. Mit der Größe der Matrizen von $N := 2^n$ ergibt sich dann $\log(N) = n \log(2)$ und somit $f(n) = 7^n = 7^{(\frac{\log N}{\log 2})} = N^{(\frac{\log 7}{\log 2})} < N^{2,81}$.

Kann dieser Gewinn an Zeit gegenüber dem Schulverfahren von der ebenfalls rekursiv wachsenden Zahl von Additionen/Subtraktionen wieder zunichte gemacht werden? Bezeichnen wir die Zahl der Additionen/Subtraktionen bei jeder Multiplikation zweier (Unter-)Matrizen der Größe 2^n mit g(n), dann gilt die Rekurrenz:

$$g(n) := \begin{cases} 7 \cdot g(n-1) + 18 \cdot 4^{n-1} & \text{falls} \quad n \geq 1 \\ 0 & \text{falls} \quad n = 0. \end{cases}$$

Die Zahl 4^{n-1} kommt daher, dass in jedem der sieben rekursiven Aufrufe 18 mal jeweils zwei Untermatrizen der Größe 2^{n-1}, also mit $2^{n-1} \times 2^{n-1} = (2^{n-1})^2 = 2^{2(n-1)} = 4^{n-1}$ vielen Elementen paarweise addiert werden. Diese Rekurrenz ist zwar nicht linear, aber Abwickeln ergibt (auch ohne eine allgemeinere Methode) die gesuchte Lösung:

$$\begin{aligned}
g(n) &= 7 \cdot g(n-1) + 18 \cdot 4^{n-1} \\
&= 7 \cdot g(n-1) + \frac{9}{2} \cdot 4^n \\
&= 7 \cdot (7 \cdot g(n-2) + \frac{9}{2} \cdot 4^{n-1}) + \frac{9}{2} \cdot 4^n \\
&= 7^2 \cdot g(n-2) + \frac{9}{2} \cdot 4^n \frac{7}{4} + \frac{9}{2} \cdot 4^n \\
&\vdots \quad \vdots \\
&= 7^k \cdot g(n-k) + \frac{9}{2} \cdot 4^n \cdot \sum_{i=0}^{k-1} (\frac{7}{4})^i \\
&\vdots \quad \vdots \\
&= 7^n \cdot g(0) + \frac{9}{2} \cdot 4^n \cdot \sum_{i=0}^{n-1} (\frac{7}{4})^i \\
&= \frac{9}{2} \cdot 4^n \cdot \frac{(\frac{7}{4})^n - 1}{\frac{7}{4} - 1} \\
&= \frac{9}{2} \cdot \frac{7^n - 4^n}{\frac{7}{4} - 1} = \frac{9}{2} \cdot 4 \frac{7^n - 4^n}{7 - 4} \\
&= 6 \cdot (7^n - 4^n) \leq 6 \cdot 7^n
\end{aligned}$$

Die Zahl der Additionen ist ebenfalls beschränkt durch eine Funktion von der gleichen Ordnung wie die Zahl $f(n)$ der Multiplikationen. Also ist der Multiplikationsalgorithmus für Matrizen der Dimension $N = 2^n$ mit maximaler Laufzeit der Größenordnung $O(N^{(\frac{\log 7}{\log 2})}) \subseteq O(N^{2,81})$ implementierbar.

8 Strukturelle Komplexitätstheorie

Die in Kapitel 1.1 angesprochene Komplexitätstheorie ist ein Teilgebiet der Theoretischen Informatik, in dem insbesondere die quantitativen Aspekte der Berechenbarkeit untersucht werden. Es wird hier also nicht nur danach gefragt, ob ein Problem entscheidbar ist, sondern darüber hinaus noch gefragt, wie schwer das Problem zu entscheiden ist. Das Ziel ist dabei, eine Einstufung und Einordnung für den Mindestaufwand an Rechenzeit oder Speicherplatz zu erhalten, der notwendig ist, um algorithmische Probleme zu lösen. Dabei sind die unentscheidbaren Probleme in der Komplexitätstheorie aus naheliegenden Gründen uninteressant. Als Maschinenmodell wird dabei die Turing-Maschine verwendet. Die Probleme, die untersucht werden, sind entweder *Optimierungsprobleme* oder reine *Ja-Nein-Entscheidungsprobleme*. Beispiele für die erste Klasse sind Fragen wie „Finde in einem beliebig vorgegebenen kantengewichteten Graphen den kürzesten Weg vom Knoten A zum Knoten B“. Entscheidungsprobleme erlauben grundsätzlich nur zwei Antworten: *Ja* oder *Nein*. Ein Beispiel dafür wäre z.B. die Frage, ob eine gegebene kontextfreie Grammatik ein bestimmtes Wort generieren kann, oder ob es in einem beliebigen ungerichteten Graphen einen Hamilton-Kreis gibt, d.h. einen Kreis, der jede Kante genau einmal verwendet. Wir werden solche Entscheidungsprobleme stets mit der Menge aller derjenigen kodierten Probleminstanzen identifizieren, für die die gestellte Frage mit *Ja* beantwortet wird.

8.1 Darstellung von Problemen für TM-Eingabe

Ein Problem ist in diesem Zusammenhang immer eine allgemeine Fragestellung, die für jede spezielle vorgegebene Problem-Instanz beantwortet werden soll, die sich aus der allgemeinen Fragestellung durch Ersetzen der frei wählbaren Parameter ergibt. Damit ein spezielles Problem von einer Turing-Maschine bearbeitet werden kann, muss das Problem der Maschine in kodierter Form vorgelegt werden. Dies kann auf die unterschiedlichste Weise geschehen, wir wollen aber hier die gleiche Form der Kodierung benutzen, wie wir sie bei Turing-Maschinen verwendet haben, nämlich durch Darstellung mit ausreichend großem Alphabet, welches eine binäre Kodierung gestattet. Die Länge dieser Darstellung wird dann als die Größe des speziellen Problems bezeichnet.

Als Beispiel sei hier das Hamilton-Kreis Problem mit seiner Notation angeführt:

8.1 Definition

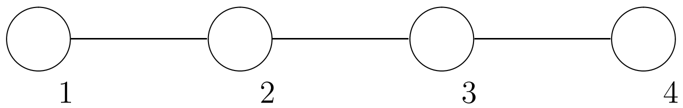
Hamilton-Kreis (H_c) (*Hamilton-Circuit*)

> **Eingabe:** Ein ungerichteter Graph $G = (V, E)$
>
> **Frage:** Gibt es einen geschlossenen Kreis in G, bei dem jeder Knoten genau einmal durchlaufen wird, d.h. existiert eine Folge von Knoten $v_1, v_2, \ldots, v_n$ mit $n = |V|, V = \{v_1, v_2, \ldots, v_n\}$ und $\{v_i, v_{i+1}\} \in E$ und $\{v_n, v_1\} \in E$ für alle $i \in \mathbb{N}$ mit $1 \leq i < n$?

□

Als Kodierung für einen Graphen $G = (V, E)$ könnte man die Liste der Kanten zusammen mit der der Knoten verwenden, was höchstens eine Länge von $|V|+3\cdot|E|-1+\lceil\log_k(|V|)\rceil\cdot(|V|+2\cdot|E|)$ ergibt, wenn die Knoten als Zahlen zur Basis k mit Trennsymbolen in der Knotenliste und der Kantenliste als Zeichenkette notiert werden.

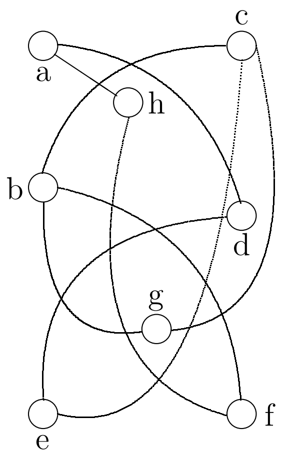
G sei der folgende, sehr einfache Graph:



Für diese Kodierung von G ist $\langle G \rangle = 1, 2, 3, 4\{1, 2\}\{2, 3\}\{3, 4\}$ mit der Länge: $12 + \lceil log_{10}(4)\rceil \cdot 10 = 22$.

Eine Darstellung mit den Zeilen der Adjazenzmatrix ist stets $|V|^2 + |V| - 1$ Zeichen lang: 0100#1010#0101#0010 hat die Länge 19. Da in ungerichteten Graphen $|E| \leq |V|^2$ gilt, unterscheiden sich diese Größen im schlechtesten Fall nur unwesentlich – d.h. polynomiell – voneinander.

In der Notation für ungerichtete Graphen verwenden wir für die Kantenbezeichnung statt der Tupel (v_1, v_2) bei gerichteten Graphen stets Teilmengen von V mit höchstens zwei Elementen. $\{a, b\}$ ist also eine Kante zwischen den Knoten a und b. Eine Schleife am Knoten c wird dann mit $\{c\}$ bezeichnet. Ansonsten entspricht die Definition der eines Graphen zu einer Relation.



Zu Ja-Nein-Problemen, wie z.B. diesem, gehören immer jene Sprachen, die die Kodierungen von solchen Problem-Instanzen enthalten, für die die Antwort *Ja* gegeben wird. In obigem Beispiel also $L_{H_c} := \{\langle G \rangle \mid G$ ist Graph mit einem Hamilton-Kreis $\}$. Das spezielle Problem für den gegebenen Graphen G' ist dann die Frage: Gilt $\langle G' \rangle \in L_{H_c}$? Üblicherweise werden in solchen Fällen die Klammern $\langle$ und $\rangle$ weggelassen, da diese standardmäßig ergänzt werden können. Wir sehen hier leicht, dass $G' \in L_{H_c}$ ist.

Aber ob eine Turing-Maschine dies nicht nur in diesem speziellen Fall, sondern für alle möglichen Problem-Instanzen mit vertretbarem Aufwand lösen kann, ist noch zu untersuchen. Woran wir interessiert sind, ist zunächst ein Algorithmus, der dieses Problem löst, der also die Menge L_{H_c} akzeptiert und dabei stets hält, d.h. die Menge L_{H_c} entscheidet.

Im Folgenden werden wir die Abkürzungen für die Probleme immer auch als Abkürzungen für die ihnen zugeordneten Sprachen verwenden. Statt L_{H_c} schreiben wir also, wie überall üblich, nur H_c und benutzen die Abkürzung des Problems immer auch als Bezeichnung für die ihm zugeordnete Sprache. Ist ein Problem entscheidbar, dann soll die dazu verwendete TM möglichst wenig Aufwand treiben, also wenig Schritte machen bzw. mit wenig Feldern auf dem Arbeitsband auskommen. Auch wollen wir dieses Problem mit anderen vergleichen, die in den unterschiedlichsten Anwendungen vorkommen. Die Mengen, die alle diejenigen Sprachen enthalten, die mit dem gleichen Aufwand an Rechenzeit oder Speicherplatz analysiert werden können, bilden natürlich wieder Sprachfamilien, die wir zu diesem Zweck mit denen vergleichen wollen, die uns aus anderer Blickrichtung schon vertraut sind.

8.2 Zeit- und Platzbedarf

Um die Größe des Aufwands von Berechnungen auf Turing-Maschinen zu definieren, verwenden wir die k-Band off-line TM aus Definition 2.10.

Eine NTM hat für das Akzeptieren eines Wortes i.A. mehrere verschiedene Erfolgsrechnungen – d.h. Folgen von Konfigurationen – zur Verfügung, was insbesondere bedeutet, dass es für ein und dieselbe Eingabe auch verschieden lange (Erfolgs-)Rechnungen geben kann.

Zur Betrachtung der Komplexität beginnen wir mit einzelnen Rechnungen, dann betrachten wir alle Rechnungen für ein festes Wort und sodann die Rechnungen für alle Wörter einer Länge. Eine Rechnung ist dabei eine endliche Folge von Konfigurationen, die mit der Anfangskonfiguration beginnt und in einer Konfiguration endet, in der die Maschine hält. Eine solche Rechnung muss also nicht mit dem Akzeptieren der Eingabe enden, es kann auch abgelehnt werden.

8.2 Definition

In einer fest vorgegebenen Rechnung benötigt eine NTM soviel Zeit, wie darin einzelne Konfigurationen durchlaufen werden.

Die NTM benötigt in dieser Rechnung soviel Platz, wie maximal Felder auf einem der Arbeitsbänder besucht werden.

□

Zum Verständnis dieser Definition der Platzbeschränkung stelle man sich alle Arbeitsbänder der NTM als Spuren eines einzigen Bandes vor.

8.3 Definition

Eine NTM A verarbeitet ein Wort w mit der Zeitbeschränkung t genau dann, wenn die kürzeste Rechnung für w nur $\lceil t \rceil$ Zeit benötigt bzw. Schritte hat.

Eine NTM A verarbeitet w mit der Platzbeschränkung $s \in \mathbb{R}$, wenn die Rechnung mit dem geringsten Platzverbrauch für w nur $\lceil s \rceil$ Platz benötigt.

□

8.4 Definition

Seien s und t Funktionen, $s : \mathbb{R} \to \mathbb{R}$, $t : \mathbb{R} \to \mathbb{R}$.

Eine NTM A ist $t(n)$-zeitbeschränkt genau dann, wenn sie für *jedes akzeptierte* Eingabewort der Länge n mit der Zeitbeschränkung $t(n)$ arbeitet.

Eine NTM A ist <u>$s(n)$-platzbeschränkt</u> genau dann, wenn sie für *jedes akzeptierte* Eingabewort der Länge n mit der Platzbeschränkung $s(n)$ arbeitet.

□

Wenn es für die Funktionen $t : I\!R \to I\!R$ eine DTM gibt, die für jedes $n \in I\!N$ nach $\lceil n \rceil$ Schritten anhält, so kann für die NTM A leicht eine äquivalente gefunden werden, die auf *allen* Eingaben der Länge n innerhalb der Zeitschranke anhält! Etwas entsprechendes gilt für die Platzbeschränkung. Jedoch haben nicht alle Funktionen diese Eigenschaft.

Normalerweise ist es absolut ausreichend, wenn man die Beschränkungsfunktionen auf den Bereich der natürlichen Zahlen einschränkt, da ja Eingabezeichen, Bandsymbole und Schritte einer Turing-Maschine nur in ganzen Einheiten verwendet werden.

Wenn eine TM mit der Platzbeschränkung $s(n)$ arbeitet, dann sollte man davon ausgehen, dass $s(n) \geq 1$ für jedes $n \in I\!N$ ist, denn jede TM benutzt wenigstens ein Feld zum Ansehen auf dem Arbeitsband, selbst wenn nur # darauf steht. Wir sagen also, die TM arbeitet mit Platzbeschränkung $s(n)$, wenn sie in Wirklichkeit $\max\{1, s(n)\}$-platzbeschränkt ist.

Es ist ebenfalls nützlich anzunehmen, dass eine TM ihre Eingabe stets vollständig liest und ein weiteres Feld vorrückt, um deren Ende festzustellen. Also sagen wir, dass eine TM mit Zeitbeschränkung $t(n)$ arbeitet, wenn sie tatsächlich $\max\{n+1, t(n)\}$- zeitbeschränkt ist.

Beispiel:
Die Sprache $\{w\$w^{rev} \mid w \in \{a, b\}^*\}$ ist kontextfrei, und kann daher mit dem Verfahren von Cocke/Younger/Kasami (siehe Veranstaltung F2) in Polynomzeit analysiert werden. Der Speicherplatz, der dabei benötigt wird, ist entsprechend groß. Diese spezielle Sprache kann aber auch leicht mit der Platzkomplexität $s(n) = n$ akzeptiert werden, wobei hier n die Länge des jeweiligen Eingabewortes bezeichnet. Wenden wir das Kellerprinzip an, so genügt uns ein Speicherbedarf von n, bei Wörtern, die tatsächlich in der Sprache enthalten sind, sogar $\lceil \frac{n+1}{2} \rceil$. Es ist aber auch möglich, nur noch mit $\log(n)$ Platzbedarf auszukommen. Dabei wird die Anzahl der schon bearbeiteten Symbole von w bzw. w^{rev} in $w\$w^{rev}$ als Binärzahl kodiert, so dass das nächste zu bearbeitende Zeichen leicht ermittelt werden kann. Die Leser(innen) sind aufgerufen, sich an einem Verfahrensvorschlag zu versuchen.

Auf der Basis von Beschränkungsfunktionen kann man nun leicht neue (vielleicht schon bekannte?) Sprachklassen definieren. Beispiel: $DTIME(f(n)) := \{L \mid L = L(A)$ für eine $f(n)$-zeitbeschränkte deterministische Turing-Maschine $A\}$. Der Nachteil einer solchen Definition *an dieser Stelle* ist offensichtlich: Wir müssten diese Klasse von allen denen unterscheiden, bei denen statt $f(n)$ eine Funktion $g(n) := c \cdot f(n)$ mit $c \neq 1$ verwendet würde. Daher untersuchen wir zunächst den Einfluß von konstanten Faktoren und Summanden auf Beschränkungsfunktionen.

8.3 Bandkompression und Beschleunigung

8.5 Theorem (Bandkompression)

Zu jeder $s(n)$-platzbeschränkten TM und jeder reellen Zahl $c \in \mathbb{R}$ mit $c > 0$ gibt es eine äquivalente TM die $c \cdot s(n)$-platzbeschränkt ist.

$\diamond$

Beweis

Sei $c < 1$, denn für $c \geq 1$ ist nichts zu beweisen. Die Konstruktion des $|w|$-platzbeschränkten LBA B aus einem beliebigen LBA A aus Theorem 5.12 kann für jede TM durchgeführt werden. Wenn A $s(n)$-platzbeschränkt ist, so ist B dann nur noch $c \cdot s(n)$-platzbeschränkt.

$\square$

Damit sind dann für die Platzbeschränkung z.B. die Funktionen $f(n) := 3n^2 - 70n + 5$ und $g(n) := n^2$ nicht von einander zu unterscheiden, denn $f(n) \leq 4n^2$ und nach Theorem 8.5 kann die mit $f(n)$-Platzbeschränkung akzeptierte Sprache auch von einer $g(n)$-platzbeschränkten TM akzeptiert werden.

Ein fast identisches Ergebnis erhalten wir für Zeitbeschränkungen:

8.6 Theorem (lineare Beschleunigung, *speed-up*)

Zu jeder $t(n)$-zeitbeschränkten TM mit $\inf\limits_{n \to \infty}(\frac{t(n)}{n}) = \infty$ und jedem $c \in \mathbb{R}$ mit $c > 0$ gibt es eine äquivalente $c \cdot t(n)$-zeitbeschränkte TM.

$\diamond$

Beweis

(Genauere Angaben in der Literatur, z.B. [HUl79])

Wieder werden jeweils r benachbarte Felder des Arbeitsbandes der zu simulierenden TM A zu einem neuen Symbol kodiert. Dies geschieht auch mit der Eingabe, die beim Lesen von dem read-only Eingabe-Band der Simulations-TM B zusammen mit dieser Blockbildung auf eines der Arbeitsbänder kopiert wird. Kopfbewegungen der zu simulierenden TM in diesem Bereich von r benachbarten Symbolen kosten die neue TM B keine einzige Kopfbewegung.

Ein ständiger Wechsel zwischen benachbarten Blöcken jedoch kostet genauso viel Zeit wie in der simulierten TM A. Um das zu sparen, werden mit dem aktuell wichtigen Block (das ist der, in dessen r Symbolen sich die TM A gerade befindet) auch gleich noch seine beiden Nachbarn in der endlichen Kontrolle gespeichert.

Dies geschieht mit vier Bewegungen des LSK von B (links-rechts-rechts-links). Danach steht der LSK von B wieder auf den ursprünglichen Block, allerdings sind jetzt die Inhalte der beiden benachbarten Blöcke bekannt. Somit kann B jetzt auch bestimmen, wie sich A innerhalb dieser drei Blöcke verhalten hätte. (Dazu braucht B keine Zeit. Diese Information ist fest in die endliche Kontrolle von B eingebaut. Wie sich A innerhalb von drei beliebigen Blöcken verhält, kann ja vor Beginn der Konstruktion von B schon bestimmt werden.) Falls A hält, so tut das auch B. Hält A nicht innerhalb der drei Blöcke, so muss sie irgendwann ja den Bereich dieser drei benachbarten Blöcke nach links oder rechts verlassen. B ist bekannt, wie der Inhalt der drei Blöcke aussehen muss, wenn A diesen Bereich verlassen will. In weiteren vier Schritten aktualisiert B nun den Inhalt der drei Blöcke und verläßt diesen Bereich entsprechend der Bewegung von A. Dann wird der ganze Prozeß wiederholt. Das Verlassen des Bereiches der drei benachbarten Blöcke kostet A mindestens r Schritte, B benötigt dafür 8 Schritte. $t(n)$ Schritte von A werden also von B in höchstens $\lceil \frac{8t(n)}{r} \rceil$ Schritten simuliert.

Dazu kommen n Schritte, um die Eingabe zu lesen und in Symbolen von Blöcken aufeinanderfolgender Zeichen zu schreiben, sowie weitere $\lceil \frac{n}{r} \rceil$ Schritte, um den LSK von B auf den Anfang der Folge von kodierten Blöcken zu bewegen. Da stets $\lceil x \rceil < x + 1$ ist, sind insgesamt höchstens $n + 1 + \frac{n}{r} + 1 + 8 \cdot (\frac{t(n)}{r})$ Schritte von B nötig. Da $\inf\limits_{n \to \infty} (\frac{t(n)}{n}) = \infty$ ist, gibt es für jede Konstante d eine Zahl n_d, so dass für alle $n \geq n_d$ dann $(\frac{t(n)}{n}) \geq d$ ist (d.h. auch $n \leq (\frac{t(n)}{d})$). Immer wenn jetzt $n \geq 2$ (und somit $n + 2 \leq 2n$) und $n \geq n_d$ gilt, ist dann also die Schrittzahl von B nicht größer als $t(n)\left(\frac{2}{d}+\frac{1}{dr}+\frac{8}{r}\right)$.

Wählen wir nun $r := \lceil\frac{16}{c}\rceil$ und $d := \lceil\frac{4}{c}\rceil + \frac{1}{8}$ dann gilt $r \cdot c \geq 16$ und $d \geq \frac{32+c}{8c}$.

Setzt man diese Werte in die obige Formel für die maximale Laufzeit von B ein, erhält man

$$\begin{aligned}
\frac{2}{d}+\frac{1}{dr}+\frac{8}{r} &= \frac{2}{d}+\frac{c}{drc}+\frac{8c}{rc}\\
&\leq \frac{2}{d}+\frac{c}{16d}+\frac{8c}{16}\\
&= \frac{32+c}{16d}+\frac{c}{2}\\
&= \frac{(32+c)c}{16cd}+\frac{c}{2}\\
&= \frac{(32+c)c}{8c\cdot 2d}+\frac{c}{2}\\
&\leq \frac{cd}{2d}+\frac{c}{2}\\
&= c
\end{aligned}$$

und die Laufzeit bleibt unterhalb von $c \cdot t(n)$ für alle $n \geq n_d$. Um die Wörter, die kürzer als $\max\{2, n_d\}$ sind, zu verarbeiten (das ist nur eine endliche Menge), braucht B nur die endliche Kontrolle und $n+1$ Bewegungen, um die Eingabe zu lesen. Damit ist B wie gewünscht eine $c \cdot t(n)$-zeitbeschränkte TM.

□

Nachdem nun klar wurde, dass konstante Faktoren oder additive Glieder bei den Beschränkungsfunktionen keine entscheidende Rolle spielen, können wir uns mit Aussagen über deren asymptotisches Verhalten begnügen und Sprachfamilien auf der Basis der Akzeptierungskomplexität besser definieren.

8.4 Komplexitätsklassen

Zur Klassifizierung von Funktionen verwendet man natürlich die bekannten Landau-Schreibweisen (vergleiche Kapitel 1, Abschnitt 1.7)

Wichtige Bemerkung:
Bei der asymptotischen Analyse gilt:

THINK BIG

das heißt, es kommt nur darauf an, wie sich die Funktionen für *große Argumente* verhalten.

Beispiel: (extrem drastisch!)

1. Es gilt $\log(n) \in o(n^{0.0001})$. Das heißt, $\lim_{n\to\infty} \frac{\log(n)}{n^{0.0001}} = 0$, kurz: $n^{0.0001}$ wächst viel schneller als $\log n$.

$$\begin{array}{rrcl} \text{Für } n = 10^{100} \text{ gilt:} & \log 10^{100} = 100 & > & (10^{100})^{0.0001} = 10^{0.01} = 1,023 \\ \text{Aber für } n = 10^{10^{100}}\text{:} & \log 10^{10^{100}} = 10^{100} & < & \left(10^{10^{100}}\right)^{0.0001} = 10^{10^{96}} \\ \text{Also:} & \forall\, n > 10^{10^{100}} : \log(n) & < & n^{0.0001}. \end{array}$$

2. Sei $\epsilon = 1/10^{10^{100}}$. Es gilt

$$\log(n) \in o(n^{\epsilon}).$$

Seien k, n so dass

$$\epsilon > 10^{-k}, \qquad n = 10^{10^{2k}}$$

$$\begin{array}{rl} \text{Dann gilt:} & \log n = 10^{2k}, \\ & n^{\varepsilon} > \left(10^{10^{2k}}\right)^{10^{-k}} = 10^{10^{k}} \\ \text{Also:} & \forall\, n > 10^{10^{2k}} : \log(n) < n^{\varepsilon}. \end{array}$$

In der Praxis werden so drastische Beziehungen wahrscheinlich nicht vorkommen, die Hauptidee der asymptotischen Analyse, THINK BIG, ist davon aber nicht betroffen.

8.7 Definition

Für Funktionen $s, t : \mathbb{N} \to \mathbb{R}$ für die stets $t(n) \geq n + 1$ und $s(n) \geq 1$ ist, bezeichne:

$$\begin{array}{rcl} \mathcal{D}\mathit{Time}(t(n)) & := & \{L \mid L = L(A) \text{ für eine } t(n)\text{-zeitbeschränkte DTM } A\} \\ \mathcal{N}\mathit{Time}(t(n)) & := & \{L \mid L = L(A) \text{ für eine } t(n)\text{-zeitbeschränkte NTM } A\} \\ \mathcal{D}\mathit{Space}(s(n)) & := & \{L \mid L = L(A) \text{ für eine } s(n)\text{-platzbeschränkte DTM } A\} \\ \mathcal{N}\mathit{Space}(s(n)) & := & \{L \mid L = L(A) \text{ für eine } s(n)\text{-platzbeschränkte NTM } A\} \\ \mathcal{P} & := & \ L \mid \text{ Es gibt ein Polynom } p : \mathbb{N} \to \mathbb{R} \text{ und eine} \\ & & p(n)\text{-zeitbeschränkte DTM } A \text{ mit } L = L(A)\} \\ \mathcal{NP} & := & \{L \mid \text{ Es gibt ein Polynom } p : \mathbb{N} \to \mathbb{R} \text{ und eine} \\ & & p(n)\text{-zeitbeschränkte NTM } A \text{ mit } L = L(A)\} \end{array}$$

Damit ist wegen des *speed-up* Satzes (Theorem 8.6):

$$\mathcal{P} = \bigcup_{i \geq 1} \mathcal{D}\mathit{Time}(n^i) \quad \text{und} \quad \mathcal{NP} = \bigcup_{i \geq 1} \mathcal{N}\mathit{Time}(n^i)$$

Ähnlich definiert man polynomiale Platz-Klassen:

$$\begin{aligned} \mathcal{P}\mathit{Space} &:= \bigcup_{i \geq 1} \mathcal{D}\mathit{Space}(n^i) \\ \mathcal{NP}\mathit{Space} &:= \bigcup_{i \geq 1} \mathcal{N}\mathit{Space}(n^i) \end{aligned}$$

□

8.8 Korollar

Die sich direkt aus den Definitionen ergebenden trivialen Beziehungen zwischen diesen Komplexitätsklassen sind offensichtlich die folgenden:

$$\begin{aligned} \mathcal{D}\mathit{Space}(f) &\subseteq \mathcal{N}\mathit{Space}(f) \\ \mathcal{D}\mathit{Time}(f) &\subseteq \mathcal{N}\mathit{Time}(f) \\ \mathcal{D}\mathit{Time}(f) &\subseteq \mathcal{D}\mathit{Space}(f) \\ \mathcal{N}\mathit{Time}(f) &\subseteq \mathcal{N}\mathit{Space}(f) \\ \mathcal{P} &\subseteq \mathcal{NP} \\ \mathcal{P}\mathit{Space} &\subseteq \mathcal{NP}\mathit{Space} \end{aligned}$$

◇

8.5 NP-Vollständige Probleme

Von Problemen, die in der Komplexitätsklasse $\mathcal{P}$ liegen, die also deterministisch in Polynomzeit gelöst werden können, sagt man häufig, dass sie effizient lösbar seien. Eine weithin verbreitete Ansicht ist, dass alle Probleme in $\mathcal{P}$ auch in der Praxis noch mit vertretbarem Aufwand bearbeitet werden können. Nehmen wir also einmal an, dass alle Probleme in $\mathcal{P}$ in der Praxis vernünftig gelöst werden können. Was ist dann mit $\mathcal{NP}$?

Bekannt ist nur, dass alle nicht-deterministisch in Polynomzeit lösbaren Probleme $L \in \mathcal{NP}$, stets mit exponentiellem Zeitaufwand, also in $\mathcal{D}\mathit{Time}(2^{p(n)})$ für ein Polynom p, deterministisch gelöst werden können. Bessere Aussagen sind in den meisten Fällen nicht bekannt.

Viele der *praktisch wichtigen* Fragestellungen haben Lösungen mit Algorithmen, die nur dann in Polynomzeit arbeiten, wenn sie nicht-deterministisch sind. Für eine Implementierung sind aber stets deterministische Verfahren nötig. Die Beantwortung der Frage ob die Sprachklassen $\mathcal{P}$ und $\mathcal{NP}$ verschieden sind, ist also – nicht nur in der Informatik – durchaus bedeutsam.

In der Klasse $\mathcal{NP}$ sind natürlich auch solche Sprachen enthalten, die in sehr kurzer Zeit, z.B. in Linearzeit, erkannt werden können, also in $\mathcal{D}Time(n)$ liegen, wie auch andere, für die in ihrer bisher besten deterministischen Implementation exponentielle Zeit verbraucht wird.

Wir sehen uns weitere Beispiele von ähnlich aussehenden aber dennoch unterschiedlich lösbaren Problemen an:

<table>
<tr><th>$\mathcal{P}$</th><th>$\mathcal{NP}$</th></tr>
<tr><td colspan="2">Gegeben:
Ungerichteter, kantenbewerteter Graph $G = (V, E)$,
eine Gewichtsfunktion $g : E \to I\!N$,
zwei Knoten $s, t \in V$ und eine Schranke $k \in I\!N$</td></tr>
<tr><td>Frage:
Gibt es einen einfachen Pfad p
von s nach t mit $g(p) \leq k$?
„Kürzester Weg zwischen zwei Knoten“</td><td>Frage:
Gibt es einen einfachen Pfad p
von s nach t mit $g(p) \geq k$?
„Längster Weg zwischen zwei Knoten“</td></tr>
</table>

<table>
<tr><th>$\mathcal{P}$</th><th>$\mathcal{NP}$</th></tr>
<tr><td colspan="2">Gegeben:
Matrix $A \in \mathbb{Z}^{m \times n}$ und ein Vektor $b \in \mathbb{Z}^m$</td></tr>
<tr><td>Frage:
Gibt es $x \in \mathbb{Z}^n$ mit $Ax = b$</td><td>Frage:
Gibt es $x \in I\!N^n$ mit $Ax = b$</td></tr>
</table>

<table>
<tr><th>$\mathcal{P}$</th><th>$\mathcal{NP}$</th></tr>
<tr><td colspan="2">Gegeben:
Ungerichteter Graph $G = (V, E)$</td></tr>
<tr><td>Frage:
Gibt es einen geschlossenen Kreis, in dem jede Kante genau einmal auftritt?
„Euler-Kreis“</td><td>Frage:
Gibt es einen geschlossenen Kreis, in dem jeder Knoten genau einmal auftritt?
„Hamilton-Kreis“</td></tr>
</table>

Wir kennen für alle der oben angegebenen Beispiele Turing-Maschinen, die zum Erkennen der dem jeweiligen Problem zugeordneten Sprachen nichtdeterministisch in Polynomzeit arbeiten. Für obige Probleme, die der Familie $\mathcal{NP}$ zugeordnet wurden, kennt man bisher nichts Besseres. Kein bekanntes, deterministisches Verfahren für diese Probleme arbeitet in polynomieller Zeit.

8.9 Definition

Sei $L_w := \{\langle G, s, t, k\,\rangle \mid G$ besitzt einen Pfad p von S nach t mit $g(p) \geq k\}$ die dem Problem „Längster Weg zwischen zwei Knoten“ zugeordnete Sprache.
Und $K_w := \{\langle G, s, t, k\,\rangle \mid G$ besitzt einen Pfad p von s nach t mit $g(p) \leq k\}$ sei die Sprache, die zu dem Problem „Kürzester Weg zwischen zwei Knoten“ gehört.

$\square$

Wie sieht eine NTM zur Erkennung der „Längster Weg zwischen zwei Knoten"-Sprache L_w mit polynomieller Zeitbeschränkung nun eigentlich aus?

Zu einem kodierten Graphen, der als Eingabe vorgelegt wird (die Kodierung des Graphen habe die Länge n), schreibt die NTM (durch nichtdeterministisches „Raten") eine beliebige Kantenfolge p von s nach t auf ihr Band, indem sie die Eingabe oft genug nach passenden Paaren $\{v_1, v_2\}$ und $\{v_3, v_4\}$ mit $v_2 = v_3$ durchsucht. Da jede Kante bei einem einfachen Pfad nur einmal durchlaufen werden darf, kann die NTM die ausgewählten Paare markieren, und muss so höchstens $|E|$-mal die Eingabe durchlaufen. Dieser Zeitaufwand $t(n)$ ist dann von der Ordnung $O(n^2)$. Die Prüfung, ob $g(p) \geq k$ ist, kann durch Addieren der in der Liste für g notierten Gewichte in etwa der gleichen Zeit geschehen. Das selbe Verfahren können wir anwenden um zu zeigen, dass das Problem K_w des kürzesten Weges ebenfalls in polynomieller Zeit nichtdeterministisch gelöst werden kann.

Damit gilt nun für beide Probleme $L_w \in \mathcal{NP}$ und $K_w \in \mathcal{NP}$.

Die Frage ist nun, ob es nicht in beiden Fällen möglich ist, eine deterministische Turing-Maschine zu finden, die die gleiche Sprache erkennt und dabei in Polynomzeit arbeitet.

Für K_w gelingt dies tatsächlich:

8.10 Theorem

Das Problem „kürzester einfacher Weg von s nach t" ist deterministisch in Polynomzeit zu lösen, d.h. $K_w \in$ P.

◇

Beweis (-Skizze)

Statt die DTM in allen Details anzugeben, skizzieren wir das Verfahren hier nur und geben eine Begründung dafür, dass nur polynomielle Zeit verbraucht wird:

Es kann vorkommen, dass es zwischen zwei Knoten mehrere Kanten mit verschiedenen Gewichten gibt. Da nach dem kürzesten Weg gesucht wird, brauchen wir aber in solchen Fällen nur jeweils die Kante mit dem kleinsten Gewicht zu betrachten. Die überzähligen Kanten können einfach aus dem Graphen entfernt werden. Wir haben jetzt also einen Graphen, der zwischen zwei Knoten höchstens eine Kante besitzt

Zuerst konstruiert nun die DTM aus der kodierten Eingabe $\langle G, s, t \rangle$ zu $G = (V, E)$, mit s und t als ausgezeichnete Knoten sowie der Angabe von $g(e)$ für jede Kante $e \in E$, eine Matrix $W \in I\!N^{n \times n}$, für $n := |V|$. In der Matrix W gibt das Element $W_{i,j}$ in der i-ten Zeile und j-ten Spalte die Länge des kürzesten, einfachen Weges von $s := v_1$ zum Knoten v_j mit bis zu i Kanten an.

Die erste Zeile kann schon eingetragen werden: es ist $W_{1,j} := g(e)$ mit $e = \{v_1, v_j\}$ falls $e \in E$ oder ω, falls $e \notin E$ Dabei ist ω ein Platzhalter mit der Bedeutung „es gibt keine Kante bzw. keinen Weg“. Formal: $\forall n \in I\!N : \omega > n$. Für $\{v_k, v_j\} \notin E$ können wir an dieser Stelle auch gleich $g(\{v_k, v_j\}) = \omega$ vereinbaren.

Nun kann $W_{i,j}$ zeilenweise durch Erhöhen des Index i von den Anfangswerten aus bestimmt werden.

$W_{i+1,j}$ wird induktiv bestimmt durch:

$$W_{i+1,j} := \min_{1 \leq k \leq |V|} \{W_{i,j}, W_{i,k} + g(\{v_k, v_j\})\}\,.$$

Da die Zeile $W_{i,j}$ für $1 \leq j \leq |V|$ bekannt ist und nur beschränkt viele, jedoch höchstens $|E|$ Kanten aus jedem Knoten heraus führen, berechnet sich der Aufwand zu $O(|V|^2 \cdot |E|)$ Schritten. Zu berücksichtigen ist $\min\{n, \omega\} := n$ sowie $n + \omega = \omega$ sowie der Aufwand für die Vergleiche und Addition. Ist v_r die Bezeichnung des zu erreichenden Zielknotens t, so kann man zum Schluß die Länge des kürzesten Pfades von s nach s in der Matrix als $W_{n,r}$ ablesen. Ist der Wert $W_{n,r} \leq k$, so akzeptiert die DTM, anderenfalls verwirft sie die Eingabe.

Für das Problem des kürzesten Weges von einem Knoten zu einem anderen in einem Graphen (gerichtet oder nicht) gibt es viele andere Algorithmen. Man erinnere sich an das Verfahren von Kleene, das benutzt wird, um aus einem endlichen Automaten den äquivalenten Rationalen Ausdruck zu erzeugen. Auch dies Verfahren läßt sich so abwandeln, dass anstelle der Ausdrücke nur die minimalen Wege notiert werden. Wir wollen hier nicht näher darauf eingehen und empfehlen die in dieser Hinsicht sehr ausführliche Literatur (z.B. [Meh84]).

□

Wenn wir nach einem Algorithmus für das „Längster Weg von s nach t“-Problem suchen, so werden wir in der Regel nicht so erfolgreich sein.

Natürlich stellt sich sofort die Frage, ob das nun an unserem *Können* liegt oder vielleicht *in dem Problem selbst* begründet ist?

Ähnlich wie bei der Reduktion eines bisher unbekannten Entscheidungs-Problems A auf ein anderes – nennen wir es B – dessen Entscheidbarkeit uns bekannt ist, so gibt es auch Reduktionen der Sprachen einer Komplexitätsklasse auf die einer anderen.

In beiden Fällen geschieht das durch eine Abbildung $f : X^* \to Y^*$ für die $f(w) \in L_B$ für ein beliebiges Wort $w \in X^*$ genau dann gilt, wenn $w \in L_A$ ist. Wird nur nach der Entscheidbarkeit des Problems A gefragt, so reicht es, wenn f berechenbar ist. Zur Entscheidung, ob ein Wort $w \in L_A$ ist, wird zunächst $f(w)$ berechnet und dann entschieden, ob $f(w) \in L_B$ gilt.

Wollen wir aber erreichen, dass auch Aussagen über die Berechnungskomplexität des Problems A bei solch einer Reduktion möglich sind, so müssen wir sicherstellen, dass diese Funktion f mit wenig Speicherplatz oder in kurzer Zeit berechnet werden kann.

Diese Überlegungen führten zuerst zum Begriff der Reduktionen, die in polynomieller Zeit durchgeführt werden können.

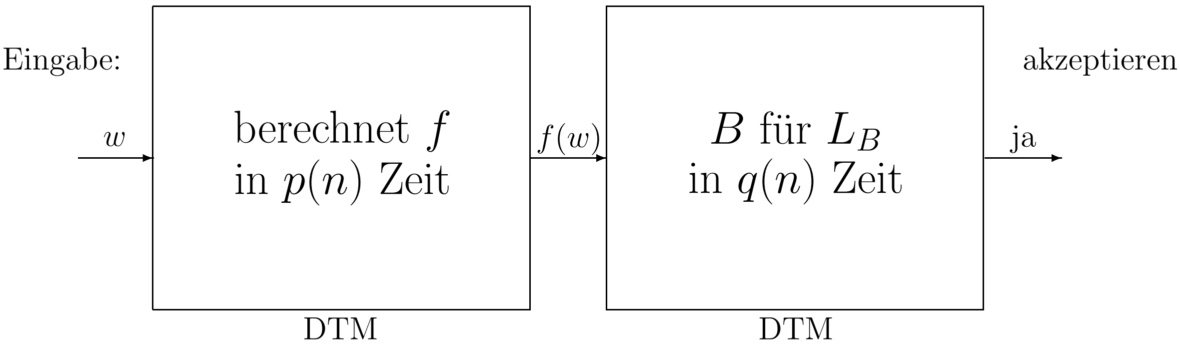
8.11 Definition

Ein Problem A ist polynomiell reduzierbar auf ein Problem B, notiert als $A \leq_{pol} B$ genau dann, wenn es eine Polynomzeit beschränkte deterministische Turing-Maschine gibt, die für die zu den Problemen gehörenden Sprachen $L_A \subseteq X^*$ und $L_B \subseteq Y^*$ eine Funktion $f : X^* \to Y^*$ berechnet mit $w \in L_A$ gdw. $f(w) \in L_B$.

□

Gilt $A \leq_{pol} B$, so kann aus jedem Verfahren für das Problem B eines für das Problem A gewonnen werden. Ist das Problem B deterministisch in Polynomzeit lösbar, so besitzt nun auch das Problem A diese Eigenschaft.

Sei dazu $n := |w|$:



Die Zeit, die nun für die Akzeptierung der Sprache L_A benötigt wird, berechnet sich zu $p(n) + q(|f(w)|)$, wenn $n = |w|$ ist, und dies ist wieder ein Polynom, wenn q ein solches war, da ja $|f(w)|$ auch durch $p(n)$ beschränkt ist.

Mit anderen Worten bedeutet das:

$$A \leq_{pol} B \text{ und } B \in \mathcal{P} \rightarrow A \in \mathcal{P}$$

.

Es gibt Sprachen, die für die Klasse $\mathcal{NP}$ typisch sind, weil entweder *alle* oder *keine* davon auch deterministisch in Polynomzeit zu erkennen sind. Diese Sprachen heißen *NP-vollständig* und sind alle paarweise aufeinander in Polynomzeit reduzierbar.

8.12 Definition

Eine Sprache L heißt NP-vollständig genau dann, wenn sie selbst in $\mathcal{NP}$ liegt und jede beliebige Sprache M aus $\mathcal{NP}$ sich auf L polynomiell reduzieren läßt, d.h. wenn gilt:

$$L \in \text{NP und } \forall M \in \mathcal{NP} : M \leq_{pol} L$$

□

Aus der Definition und dem oben Gesagten über die gesamte Dauer der Rechnung aus Reduktion gefolgt von Erkennung mit der Turing-Maschine für eine NP-vollständige Sprache, folgt leicht das folgende Korollar:

8.13 Korollar

Sei L NP-vollständig, dann gilt $L \in \mathcal{P}$ gdw. $\mathcal{P} = \mathcal{NP}$.

◇

Beweis

$$\begin{aligned} L \text{ NP-vollständig und } L \in \mathcal{P} &\rightarrow \forall M \in \mathcal{NP} : M \leq_{pol} L \text{ und } L \in \mathcal{P} \\ &\rightarrow \forall M \in \mathcal{NP} : M \in \mathcal{P} \\ &\rightarrow \mathcal{NP} = \mathcal{P} \end{aligned}$$

$\mathcal{NP} = \mathcal{P}$ impliziert trivialerweise, dass jede NP-vollständige Sprache auch in $\mathcal{P}$ liegt.

□

Die NP-vollständigen Sprachen, bzw. die dazu gehörenden Probleme, werden zu recht als schwierig angesehen, kennt man bisher doch stets nur deterministische Verfahren, die in exponentieller Zeit arbeiten. Das Problem „$\mathcal{P} = \mathcal{NP}$?“ ist seit seiner Formulierung durch S. Cook (1971) ungelöst. Die meisten Wissenschaftler vermuten die Ungleichheit dieser Klassen und vielleicht kann es dafür sogar mit unseren Mitteln niemals einen formalen Beweis geben.

Selbst wenn wir nicht sagen können, dass NP-vollständige Probleme beweisbar schwierig sind, so ist die Evidenz dafür deutlich genug. Mit den zur Zeit zur Verfügung stehenden Mitteln ist bislang keine Verbesserung möglich oder in Sichtweite.

Das erste Problem, das als NP-vollständig erkannt wurde, ist das *Erfüllbarkeitsproblem für boolesche Formeln* (abgekürzt SAT für *satisfiability*).

8.14 Definition

SAT:= $\{w \in X^* \mid w$ ist ein erfüllbarer boolescher Ausdruck $\}$ ist bzw. beschreibt das Erfüllbarkeitsproblem, das wie folgt definiert ist:

Erfüllbarkeitsproblem:

> **Eingabe:** Eine Menge V von Variablen und eine boolesche Formel $B \in X^*$ darüber mit den üblichen Operatoren $\vee, \wedge, \neg$ sowie Klammern.
>
> **Frage:** Gibt es eine Belegung der Variablen mit TRUE und FALSE derart, dass der Wahrheitswert der Formel B TRUE wird?

□

8.15 Theorem

SAT ist NP-vollständig.

◇

Beweis -Idee

(Den exakten Beweis entnehme man [GJo79] oder [Meh84])

Es ist einfach zu sehen, dass $SAT \in \mathcal{NP}$ ist. Zu einer gegebenen Formel B mit den Variablen $x_1, x_2, \ldots x_n$ wird nichtdeterministisch eine Belegung der Variablen mit TRUE oder FALSE geraten. Danach wird geprüft, ob B mit dieser Belegung den Wert TRUE bekommt. Dies ist in Polynomzeit möglich.

Um zu zeigen, dass *jedes* Problem aus $\mathcal{NP}$ deterministisch in Polynomzeit auf SAT reduziert werden kann, stellen wir zunächst einmal fest, dass es ausreicht, wenn wir 1-Band TM betrachten. Gemäß Theorem 2.11 kann jede Sprache, die von einer k-Band TM erkannt wird, auch von einer 1-Band TM erkannt werden. Eine genauere Analyse zeigt, dass die Simulation einer $t(n)$ zeitbeschränkten k-Band TM auf einer 1-Band TM maximal $t^2(n)$ Zeit erfordert. Für $\mathcal{NP}$ macht es also keinen Unterschied, ob wir 1-Band oder k-Band Maschinen betrachten, da das Quadrat eines Polynoms wieder ein Polynom ist.

Es werden nun zu jeder polynomzeitbeschränkten 1-Band Turing-Maschine boolesche Formeln aufgestellt, deren Konjunktion genau dann wahr ist, wenn die TM eine Erfolgsrechnung besitzt. Für jedes Polynom p wird es eine Reduktions-DTM geben, die dies leistet.

Sei $L \in \mathcal{NP}$ beliebig und $A_L := (Z, X, Y, K, q_0, Z_{\text{end}})$ eine $p(n)$-zeitbeschränkte 1-Band-NTM mit $L = L(A_L)$, wobei p ein Polynom ist. A_L hat in jeder ihrer Erfolgsrechnungen wegen der Zeitbeschränkung bei Eingabe von w mit $|w| = n$ höchstens $p(n)$ verschiedene Konfigurationen k_i, wobei die Länge $|k_i|$ einer jeden höchstens $p(n)+1$ ist. Eine Rechnung von A_L läßt sich also eindeutig durch eine Folge von Konfigurationen $k_0\$k_1\$\ldots k_{p(n)}$ beschreiben. In jedem der maximal $p(n)$ Schritte kann sich der LSK von A_L jeweils nur an genau einer von $p(n)$ verschiedenen Positionen befinden. Weiterhin kann sich zu jedem Zeitpunkt A_L in nur genau einem Zustand befinden. Wir können o.B.d.A. annehmen, dass sich bei vorzeitigem Akzeptieren die Konfiguration in allen weiteren Schritten nicht mehr verändert und die aktuelle Bandinschrift stets alle $p(n)$ Felder beschriftet hat, notfalls mit aufgefüllten #.

Nun können wir Boole'sche Variablen definieren, mit deren Hilfe eine Formel F_w konstruiert wird, so dass „$w \in L$“ gdw. „Es gibt eine Erfolgsrechnung für w mit $p(n)$ Schritten“ gdw. „Es gibt eine F_w erfüllende Variablenbelegung“.

Die Variablen mit Ihrer Bedeutung sind:

FELD$(i, j, t) \hat{=}$ In Konfiguration k_t steht das Zeichen x_j in Feld i.

ZUSTAND$(r, t) \hat{=}$ In der Konfiguration k_t befindet sich die TM A_L im Zustand z_r.

KOPF$(i, t) \hat{=}$ In der Konfiguration k_t steht der LSK auf dem Feld i.

Die Anzahl dieser Variablen berechnet sich zu einer Größe der Ordnung $O(p(n)^2)$. Wir definieren die Formel $\oplus(a_1, a_2, \ldots, a_r) := (a_1 \vee a_2 \vee \ldots \vee a_r) \bigwedge_{i \neq j}(\neg a_i \vee \neg a_j)$ mit der Bedeutung: $\oplus(a_1, a_2, \ldots, a_r) = 1$ gdw. genau ein $a_i = 1$. Die Länge dieser Formel ist von der Ordnung $O(r^2)$.

Die Formel F_w hat die Form $F_w := A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G$.

Die einzelnen Teilformeln bedeuten:

$A \hat{=}$ In jeder Konfiguration k_t steht der LSK von A_L auf genau einem Feld.

$B \hat{=}$ In jeder Konfiguration k_t enthält jedes Feld genau ein Zeichen.

$C \hat{=}$ In jeder Konfiguration k_t befindet sich A_L in genau einem Zustand.

$D \hat{=}$ Bei jedem Übergang wird genau das Feld verändert, auf das der LSK zeigt.

$E \hat{=}$ Jeder Übergang im Zustand z_k mit x_j unter dem LSK entspricht der Turing-Tafel.

$F \hat{=}$ Die erste Konfiguration ist $k_0 = q_0 w\# \ldots \#$.

$G \hat{=}$ Der Zustand in der letzten Konfiguration $k_{p(n)}$ ist Endzustand aus Z_{end}.

Als Beispiel für die Teilformeln und deren Größe geben wir hier nur die ersten an, die anderen werden auf die selbe Art gebildet und sind alle in $O(p^3(n))$ Zeit konstruierbar. Die Details entnehme man bitte der Literatur.

Es ist $A := A_1 \wedge A_2 \wedge \ldots \wedge A_{p(n)}$ und

$$\forall t \leq p(n) \text{ sei } A_t := \oplus(\text{KOPF}(1,t), \text{KOPF}(2,t), \ldots, \text{KOPF}(p(n),t))$$

Auch B und C sind zusammengesetzte Formeln:

$$\begin{aligned}
B &:= \bigwedge_{1 \leq i,t \leq p(n)} B(i,t) \text{ mit} \\
B(i,t) &:= \oplus(\text{FELD}(i,1,t), \ldots, \text{FELD}(i,m,t)),\ m := |Y| \\
\\
C &:= \bigwedge_{1 \leq t \leq p(n)} C_t \text{ mit} \\
C_t &:= \oplus(\text{ZUSTAND}(1,T), \ldots, \text{ZUSTAND}(s,t)),\ s := |Z|
\end{aligned}$$

An diesen Formeln kann man beispielhaft sehen, wie jedes einzelne Detail der Erfolgsrechnungen ausgedrückt werden kann.

□

Wenn wir einem neuen Problem begegnen dessen Komplexität wir nicht kennen, so hilft – oft noch vor der Suche nach einem effizienten Verfahren – die Reduktion auf ein schon bekanntes, so dass dann dessen einfache Methode mitverwendet werden kann. Oder man findet eine Reduktion von einem schwerem, z.B. NP-vollständigen Problem, auf das eigene. Im letzteren Fall kann man nicht mehr mit einem in allen Fällen effizienten Algorithmus rechnen.

8.16 Lemma

L NP-vollständig, $L \leq_{pol} M$ und $M \in \mathcal{NP} \rightarrow M$ NP-vollständig

◇

Beweis

Das Lemma folgt direkt aus der Definition von NP-Vollständigkeit und der Transitivität der Polynomzeitreduktionen.

□

Dies kann zum Beispiel mit der Variante *KNF* des Problems *SAT* durchgeführt werden, bei der sich die booleschen Formeln alle in konjunktiver Normalform befinden.

8.17 Definition

Das Erfüllbarkeitsproblem boolescher Formeln in konjunktiver Normalform wird durch die Sprache $KNF \subsetneq SAT$ gegeben, wobei:

$\underline{KNF} := \{w \in X^* \mid w$ ist eine erfüllbare boolesche Formel in konjunktiver Normalform$\}$

□

8.18 Theorem

KNF ist NP-vollständig.

◇

8.19 Definition

Die Sprache $3\text{-}SAT \subsetneq KNF \subsetneq SAT$ ist gegeben durch:

$\underline{3\text{-}SAT}:= \{w \in X^* \mid w$ ist erfüllbare boolesche Formel in konjunktive Normalform mit genau 3 Literalen in jeder Klausel $\}$.

□

8.20 Theorem

$3\text{-}SAT$ ist NP-vollständig.

◇

Wir hatten vorher gesehen, dass das Problem L_w des „längsten Weges zwischen zwei Knoten“ zwar in Polynomzeit aber dies bisher noch nicht deterministisch lösbar ist. Wie schon angedeutet, ist dieses Problem tatsächlich NP-vollständig, was eine leichte Reduktion von einem anderen NP-vollständigen Problem auf L_w zeigen wird. Das Problem, das wir dazu benutzen ist die Frage nach der Existenz eines Hamilton-Kreises, was im Unterschied zum Euler-Kreis sicher dann schwerer ist, wenn $\mathcal{P} \neq \mathcal{NP}$ ist. Das Problem H_c war ganz zu Anfang von Kapitel 8 definiert worden.

Wir geben auch das nächste Theorem ohne Beweis.

8.21 Theorem

Das Hamilton-Kreis Problem H_c ist NP-vollständig.

◇

8.22 Theorem

Das Problem L_w („Längster Weg zwischen zwei Knoten“) ist NP-vollständig.

◇

Beweis

Wir können Lemma 8.16 benutzen. $L_w \in \mathcal{NP}$ ist schon bekannt. Und mit Theorem 8.21 wissen wir, dass H_c NP-vollständig ist. Wir brauchen also nur noch eine Polynomzeit-Reduktion, die $H_c \leq_{pol} L_w$ leistet.

Sei G ein Graph, dessen Kantenbewertungen alle gleich sind und den Wert 1 haben, dann ist jeder Hamilton-Kreis ein einfacher Pfad vom Knoten v_0 zum gleichen Knoten v_0 zurück, dessen Länge mit $|V|$ maximal in G ist. Umgekehrt ist jeder einfache Pfad der Mindestlänge $|V|$ immer auch ein Hamilton-Kreis. Also löst die Frage nach der Existenz solch eines einfachen Pfades immer auch das Problem H_c.

□

NP-vollständige Probleme gibt es in allen Bereichen, einige weitere seien hier zusätzlich angegeben:

8.23 Theorem

Folgende Probleme sind NP-vollständig:

Allgemeines Rucksackproblem(*knappsack problem*)

> **Eingabe:** Eine maximale Tragfähigkeit $G \in I\!N$, m Gegenstände unterschiedlichen Gewichts $g_i \in I\!N, 1 \leq i \leq m$ und unterschiedlichen Wertes $w_i \in I\!N, 1 \leq i \leq m$, von denen beliebig viele zur Verfügung stehen.

Gesucht: Vektor $\mathbf{x} := (x_1, x_2, \ldots, x_m) \in I\!N^m$ mit:

$$\textbf{1.} \sum_{i=1}^{m} x_i \cdot w_i \text{ ist maximal} \qquad \text{und} \qquad \textbf{2.} \sum_{i=1}^{m} x_i \cdot g_i \leq G.$$

Variante des Rucksackproblems

Eingabe: Eine maximale Tragfähigkeit $G \in I\!N$, m Gegenstände unterschiedlichen Gewichts $g_i \in I\!N, 1 \leq i \leq m$ und unterschiedlichen Wertes $w_i \in I\!N, 1 \leq i \leq m$, von denen jeder nur einmal zur Verfügung steht.

Gesucht: Indexmenge $I \subseteq \{1, 2, \ldots, m\}$ mit:

$$\textbf{1.} \sum_{i \in I} w_i \text{ ist maximal} \qquad \text{und} \qquad \textbf{2.} \sum_{i \in I} g_i \leq G.$$

Partition (*partition*)

Eingabe: Eine endliche Menge A und eine Funktion $g : A \to I\!N$

Frage: Gibt es eine Teilmenge $B \subsetneq A$ derart, dass

$$\sum_{a \in B} g(a) = \sum_{a \in A \setminus B} g(a)$$

Clique (*clique*)

Eingabe: Ungerichteter Graph $G := (V, E)$ und $k \in I\!N$ mit $k \leq |V|$

Frage: Enthält G eine k-Clique, d.h. eine Teilmenge V_K mit k Knoten von V derart, dass $V_K \times V_K \setminus Id_V \subseteq E$?

Handlungsreisender (*traveling salesman*)

Eingabe: Ein ungerichteter Graph $G := (V, E)$, eine Gewichtsfunktion $g : E \to I\!N$ und eine Beschränkung $k \in I\!N$

Frage: Gibt es einen Hamilton-Kreis p in G mit $g(p) \leq k$?

$\diamond$

Wenn wir ein Problem in die Komplexitätsklasse P einordnen können, so meistens deshalb, weil wir einen konkreten Algorithmus angeben können, der es deterministisch „relativ schnell“ löst. Solche Einordnungen sind uns leider nicht immer effektiv möglich. Auf jeden Fall aber wissen wir, dass dieses Problem dann nicht so schwierig zu lösen ist wie viele andere wichtige Fragestellungen.

Was wir bisher nicht untersucht haben ist der Zusammenhang zwischen Zeit- und Platz-Komplexitätsklassen sowie zu den bekannten Sprachklassen der Chomsky-Hierarchie.

Oft werden die Zeit-Komplexitätsklassen als besonders bedeutungsvoll angesehen, aber auch nach dem Platzbedarf können viele Probleme eingestuft und verglichen werden. Für die Klasse PSPACE aller Probleme, die mit polynomiellem Platzbedarf gelöst werden können, z.B., existieren sogar vollständige Sprachen. Möchte man allerdings Reduktionen für diesen Vollständigkeitsbegriff verwenden, so muss sichergestellt werden, dass diese auch mit der erforderlichen Platzbeschränkung arbeiten. Eine Reduktion mit polynomieller Zeitbeschränkung ist dazu in der Regel nicht mehr zu verwenden, da diese viel zu viel Platz benutzen könnte.

Das gilt zwar nicht für die Klasse PSPACE, jedoch bei Platzklassen mit geringerer Platzbeschränkung würde dies leicht auftreten. Man definierte daher schon frühzeitig Reduktionen, die mit wenig (genauer: logarithmischem) Platzbedarf auskommen müssen.

Für diesen Teil der Komplexitätstheorie, und die offen stehenden Fragen, verweisen wir auf die Literatur und die weiterführenden Vorlesungen im Hauptstudium.

Literatur

[Aig93] M. Aigner: „Diskrete Mathematik“, Vieweg, Braunschweig (1993).

[And01] J.A. Anderson: „Discrete Mathematics with Combinatorics“, Prentice Hall (2001).

[BvG88] S. Baase & A.v. Gelder: „Computer Algorithms – Introduction to Design and Analysis“, Addison Wesley, Reading Mass. (2000).

[BDi88] J.L. Balcázar & J. Diáz & J. Gabárro: „Structural Complexity I“, EATCS-Monographs 11, Springer-Verlag (1988).

[BGo91] F.L. Bauer & G. Goos: „Informatik, Eine einführende Übersicht, Bde. I,II“, Springer-Verlag, Heidelberg (1991).

[Big99] W.L. Biggs: „Discrete Nathematics“, Oxford Univ. Press (1985/1999).

[BLa96] R. Bodendiek & R. Lang: „Lehrbuch der Graphentheorie, Bde. I, II“, Spektrum Akademischer Verlag, Heidelberg (1995/1996).

[Boe85] E. Börger: „Berechenbarkeit, Komplexität, Logik”, Vieweg, Braunschweig (1985).

[BBr96] G. Brassard & P. Bradley: Fundamentals of Algorithmics, Prentice Hall, Englewood Cliffs, NJ (1996).

[Bro95] I.N. Bronstein & K.A. Semedjajew & G. Musiol & H. Mühlig: „Taschenbuch der Mathematik“ Verlag Harri Deutsch, Frankfurth a.M. (1995).

[CLR91] T.H. Corman, Ch.E. Leiserson & R.L. Rivest: “Introduction to Algorithms“, MIT-Press McGraw-Hill Book Comp. (1991).

[Dit96] I. Dittmer: „Konstruktion guter Algorithmen“, B.G. Teubner, Stuttgart (1996).

[Dud93] „Duden Informatik“, Dudenverlag, Mannheim (1993).

[GKP89] R. Graham and O. Knuth, D. and Patashnik: „Concrete Mathematics“, Addison-Wesley Publ. Comp., Reading, Ma. (1989).

[GJo79] M.R. Garey & D.S. Johnson: „Computers and Intractability“, Freeman (1979).

[Gru97] J. Gruska: „foundations of computing“, Int. Thomson Computer Press, London, Boston, Bonn (1997).

[Hil00] D. Hilbert: „Mathematische Probleme”, Vortrag auf dem Pariser Mathematiker-Kongress 1900, in: F.S. Alexandrow, „Die Hilbert’schen Probleme”, Leipzig (1971).

[HUl79] J.E. Hopcroft & J.D. Ullman: „Introduction to Automata Theory, Languages, and Computation“, Addison-Wesley Publ. Comp., Reading Mass. (1979).

[HUl86] J.E. Hopcroft & J.D. Ullman: „Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie”, Addison-Wesley Publ. Comp., Bonn (1986).

[HMU01] J.E. Hopcroft & R. Motwani& J.D. Ullman: „Introduction to Automata Theory, Languages, and Computation“, Addison-Wesley Publ. Comp., Reading Mass. (2001).

[HSa81] E. Horowitz & S. Sahni: „Algorithmen, Entwurf und Analyse“, Springer-Verlag Heidelberg (1981).

[Ihr99] T. Ihringer: „Diskrete Mathematik“, B.G. Teubner, Stuttgart (1999).

[Joh82] D.S. Johnson: „NP-Completeness – An Ongoing Guide“, Journal of Algorithms (Seit 1982 mind. 22 Folgen).

[KEM68] „Kleine Enzyklopädie der Mathematik“, Bibl. Institut, Leipzig (1968).

[Koz92] D.E. Kozen: „The Design and Analysis of Algorithms“, Springer-Verlag, Heidelberg (1992).

[Lin65] S. Lin: „Computer Solutions of the TSP“, in: Technical Journal **44**, Bell Systems (1965) 2245–2269.

[Loe76] J. Loeckx: „Algorithmentheorie“, Springer-Verlag Heidelberg (1976).

[Mau69] H. Maurer: „Theoretische Grundlagen der Programmiersprachen“, B.I. Wissenschaftsverlag, Stuttgart (1969).

[Meh84] K. Mehlhorn: „Graph Algorithms and NP-Completeness“, EATCS-Monographs 2, Springer-Verlag (1984).

[Möl97] H. Möller: „Algorithmische Lineare Algebra“, Vieweg & Sohn, Braunschweig, Wiesbaden (1997).

[OWi90] T. Ottmann & P. Widmayer: „Algorithmen und Datenstrukturen“, Reihe Informatik Band 70, B.I. Wissenschaftsverlag, Mannheim (1990).

[Pau78] W.J. Paul: „Komplexitätstheorie“, Teubner, Stuttgart, (1978).

[PSt98] C.H. Papadimitriou & K. Steiglitz: Combinatorial Optimization: Algorithms and Complexity“, Dover Publications Inc., Mineola, New York (1998).

[Rei90] K.R. Reischuk: „Einführung in die Komplexitätstheorie“, Teubner, Stuttgart (1990).

[Sav98] J. Savage: „Models of Computation: Exploring the Power of Computing “, Addison Wesley (1998).

[Sed92] R. Sedgewick: „Algorithmen“, Addison Wesley (1992).

[Tra77] B.A. Trachtenbrodt: „Algorithmen und Rechenautomaten“, Verlag der Wissenschaften, Berlin (1977).

[Tur43] A.M. Turing: „On computable numbers with an application to the Entscheidungsproblem“, Proc. London Math. Soc. 42 (1936) 230-265; 43 (1937) 544-546.

[Weg93] I. Wegener: „Theoretische Informatik“, B.G. Teubener, Stuttgart (1993).

[Zob87] R. Zobel: „Diskrete Strukturen – eine angewandte Algebra für Informatiker“, Reihe Informatik Band 49, B.I. Wissenschaftsverlag, Mannheim (1987).

Index

Abschlußeigenschaft, 85
8 Königinnen-Problem, 122
Ackermann-Funktion, 68
ägyptische Multiplikation, 3
AFL-Theorie, 84
Algorithmus, 5, 7, 12, 27
Allgemeines Rucksackproblem, 113
arithmetische Funktion, 17
aufzählbare Menge, 44

Bandkompression, 156
Baum-Adressen, 43
Beschleunigung
  lineare , 156
Binäre Suche, 100
Branch and Bound, 123

Clique, 173

Darstellung von Problemen, 151
dichtestes Punktepaar, 102, 103
diophantische Gleichung, 13
Divide and Conquer, 100, 139
Division
  ganzzahlige, 3
3-SAT, 171
DTM, *siehe* Turing-Maschine
Dyck-Wörter, 15
Dynamische Programmierung, 112

entscheidbare Menge, 44
Erfüllbarkeitsproblem, 167, 171
Euklidischer Algorithmus, 23
euklidsche Ebene, 135

Fakultätsfunktion, 99, 147
Fermatscher Satz, 54
Flussdiagramm, 16
Funktion
  μ-rekursive, 70
  arithmetische, 17
  partiell rekursive, 37
  partiell-rekursive, *siehe* Funktion
  primitiv-rekursive, 65
  totale, 60
  Turing-berechenbare, 37

ganzzahlige Division, 3
Gebiete, 135
Gesamtalphabet, 87
ggT, *siehe* größter gemeinsamer Teiler
Grammatik
  monotone, 90
  Typ-0, 86
  Typ-1, 89
Greedy-Algorithmen, 105
größter gemeinsamer Teiler, 4
Gödelisierung, 48

Halteproblem, 51
Hamilton-Kreis, 152, 172
Handlungsreisender, 11, 118, 123, 173
Harmonische Zahlen, 145

Integralabschätzung, 145

k-Band off-line Turing-Maschine, 41
k-Keller-Automat, 62
k-Zähler-Automat, 63
Kachelproblem, 57
Knappsack, 172
Knappsack Problem, *siehe* Allgemeines Rucksackproblem
KNF, 171
Komplexitätsklassen, 158
Konfiguration, 35
kontextsensitive Grammatik, 89

Landau-Schreibweise, *siehe* O-Notation
LBA, *siehe* linear beschränkter Automat
linear beschränkter Automat, 93
$\log(n) = \lceil \log_2(n) \rceil$, 17
Längster Weg, 172

Matrix Chain Produkt, 119
Menge
  entscheidbare, 44
  rekursiv aufzählbare, 44
Mergesort, 101
monotone Grammatik, 90
Multi Precision Multiplication, 104, 105
Multiplikation
  multi precision, 104, 105
  ägyptische, 3
Münzrückgabe, 112

Nonterminal, 86
NP-vollständig, 166
NTM, *siehe* Turing-Maschine

O-Notation, 17

partition, 173
Phrasenstruktur-Grammatik, *siehe* Grammatik
Platzbedarf, 154
Platzbeschränkung, 154
Platzkomplexität
  logarithmische, 79
  uniforme, 79
Polynomzeitreduktion, 170
Primfaktorzerlegung, 4
Primzahltest, 4, 54
Produktion, 87
Programmlänge, 22

RAM, *siehe* Registermaschine
Rechenzeit, 22
Reduktion, 56
Reduktion, polynomielle, 165
Regel, 87
Registermaschine, 72
Rekurrenzgleichungen, 134
rekursive Menge, *siehe* Menge
Rucksackproblem, 173
  allgemeines, 113
  vereinfachtes, 111
Rundreise, 11, *siehe* Handlungsreisender

SAT, *siehe* Erfüllbarkeitsproblem
Schnelle Multiplikation, 148
Schrittrelation $\vdash_{\overline{A}}$, 36
semi-Thue System, 86
speed-up, 156
Speicherplatzbedarf, 22
Sprache
  akzeptierte, 36

entscheidbare, *siehe* entscheidbare Menge
rekursiv aufzählbare, 44
rekursive, *siehe* entscheidbare Menge
Sprachfamilie, 84
STS, *siehe* semi-Thue System

Terminalsymbol, 87
TM, *siehe* k-Band off-line Turing-Maschine
traveling salesman, *siehe* Handlungsreisender
travelling salesman, 11, 118, 123
Turing-berechenbar, 37
Turing-Maschine, 27
deterministische, 27
k-Band off-line Turing-Maschine, 41
nichtdeterministische, 42
universelle, 50
Turing-Tabelle, 29
Turing-Tafel, 30
Türme-von-Hanoi, 134, 137

Universelle Turing-Maschine, 50

zehntes Hilbertsches Problem, 13
Zeitbedarf, 154
Zeitbeschränkung, 154