BERTHOLD VÖCKING · HELMUT ALT
MARTIN DIETZFELBINGER
RÜDIGER REISCHUK · CHRISTIAN SCHEIDELER
HERIBERT VOLLMER · DOROTHEA WAGNER

# Taschenbuch der Algorithmen



eXamen.press

Springer

eXamen.press

**eXamen.press** ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Berthold Vöcking · Helmut Alt
Martin Dietzfelbinger · Rüdiger Reischuk
Christian Scheideler · Heribert Vollmer
Dorothea Wagner

# Taschenbuch der Algorithmen

Springer

*Herausgeber:*

Prof. Dr. rer. nat. Berthold Vöcking
Lehrstuhl für Informatik 1
(Algorithmen und Komplexität)
RWTH Aachen
Ahornstr. 55, 52074 Aachen

Prof. Dr. rer. nat. Helmut Alt
Institut für Informatik
Freie Universität Berlin
Takustr. 9, 14195 Berlin

Prof. Dr. rer. nat. (USA) Martin Dietzfelbinger
Institut für Theoretische Informatik
Fakultät für Informatik und Automatisierung
Technische Universität Ilmenau
Helmholtzplatz 1, 98693 Ilmenau

Prof. Dr. math. Rüdiger Reischuk
Institut für Theoretische Informatik
Universität zu Lübeck
Ratzeburger Allee 160, 23538 Lübeck

Prof. Dr. rer. nat. Christian Scheideler
Lehrstuhl für Informatik 14
(Effiziente Algorithmen)
Technische Universität München
Boltzmannstr. 3, 85748 Garching

Prof. Dr. rer. nat. Heribert Vollmer
Institut für Theoretische Informatik
Leibniz Universität Hannover
Appelstr. 4, 30167 Hannover

Prof. Dr. rer. nat. Dorothea Wagner
Institut für Theoretische Informatik
Universität Karlsruhe (TH)
Am Fasanengarten 5, 76131 Karlsruhe

ISBN 978-3-540-76393-2 e-ISBN 978-3-540-76394-9

DOI 10.1007/978-3-540-76394-9

ISSN 1614-5216

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

© 2008 Springer-Verlag Berlin Heidelberg

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

*Satz und Herstellung:* le-tex publishing services oHG, Leipzig
*Einbandgestaltung:* KünkelLopka Werbeagentur, Heidelberg

Gedruckt auf säurefreiem Papier

9 8 7 6 5 4 3 2 1

springer.com

# Vorwort

Die Informatik als Wissenschaft der Informationsverarbeitung gewinnt in fast allen Lebensbereichen mehr und mehr an Bedeutung. Die weltweite Vernetzung durch das Internet hat diese Entwicklung noch einmal entscheidend beschleunigt. So sind die Errungenschaften der Informatik heute allgegenwärtig in Produktion, Logistik, Verkehr, Medizin und in den Medien. Große wissenschaftliche Erfolge wie etwa die Sequenzierung des menschlichen Genoms wären ohne die Informatik nicht möglich gewesen. Auch im privaten Bereich scheint die Informatik omnipräsent zu sein. Ihre Errungenschaften stecken nicht nur im häuslichen Computer. So gut wie alle modernen Haushaltsgeräte – von der Waschmaschine über den Toaster bis zum Fernseher – werden durch eingebettete Systeme gesteuert. In fast jedem heute vom Band laufenden PKW finden sich gleich mehrere eingebettete Systeme, die nicht nur Motor und Bremssystem, sondern auch Klimaanlage und Fensterheber steuern – in Zukunft wohl auch das automatische Einparksystem. Selbst bei der zwischenmenschlichen Kommunikation vertrauen wir uns mehr und mehr der Informatik an, indem wir moderne Formen des Informationsaustausches wie E-Mails und SMS nutzen.

Die rasante Entwicklung der Informatik, die wir in den letzten Jahrzehnten beobachten konnten und die sich wahrscheinlich auch in der nächsten Zeit unvermindert fortsetzen wird, ist nicht nur auf die stetig fortschreitende Verbesserung der Hardware, sondern insbesondere auch auf die permanente Neu- und Weiterentwicklung der Software in Form immer effizienterer Programme zurückzuführen. Die größten Verbesserungen beruhen dabei auf neu entwickelten Algorithmen, hinter denen häufig clevere und kreative Ideen stecken. Algorithmen sind geschickte Verfahren, die vorgegebene Probleme auf unterschiedlichste Weise lösen. Dabei handelt es sich nicht nur um mathematische Aufgaben, die sich beispielsweise auf das Rechnen mit Zahlen beziehen, sondern auch um andere, ganz alltägliche Problemstellungen, bei denen räumli-

che Orientierung, logischer Spürsinn oder geschicktes Verhandeln gefragt sind, beispielsweise um Fragestellungen der folgenden Art:

- Wie lässt sich der kürzeste Weg zwischen zwei Orten ermitteln?
- Wie sollten Seeräuber eine Schatzkarte aufteilen, bzw. Bankangestellte den Geheimcode des Tresors?
- Wie können mehrere hungrige Partygäste einen Kuchen gerecht untereinander aufteilen?

Dieses Buch unternimmt einen umfangreichen Streifzug durch die faszinierende Welt der Algorithmen. Es verlangt keine besonderen Vorkenntnisse, so dass Schülerinnen und Schüler ab der Mittelstufe und auch Informatikinteressierte Laien neue und überraschende Einblicke gewinnen können. In 43 Artikeln von Informatikern, die an Universitäten im In- und Ausland lehren, werden wichtige und besonders elegante Algorithmen anschaulich und verständlich erklärt.

Die Initiative zur Erstellung einer derartigen Algorithmensammlung geht zurück auf Professor Dr. Volker Claus aus dem Vorstand des *Fakultätentags Informatik*, der Vereinigung der Informatikfakultäten an deutschen Universitäten. Seine Idee war es, im Informatikjahr 2006 wöchentlich einen Algorithmus herauszugreifen und diesen im Internet zu präsentieren mit dem Ziel, Schülerinnen und Schülern zu vermitteln, welche kreativen Ideen und Konzepte hinter Algorithmen stecken und welche Faszination von der Informatik ausgehen kann. Diese Idee wurde in der Initiative *Algorithmus der Woche* umgesetzt, in der Woche für Woche von März bis Dezember 2006 jeweils ein Artikel ins Netz gestellt wurde. Für dieses Buch wurde jeder dieser Artikel gründlich und umfassend überarbeitet. Die Autoren haben zahlreiche Verbesserungsvorschläge der Herausgeber aufgenommen und auch viele weitere Ergänzungen eingebracht. Insbesondere wurden Querbezüge zu anderen Artikeln der Sammlung eingearbeitet, und am Ende jedes Artikels gibt es Hinweise auf weiterführende Literatur.

An der umfangreichen redaktionellen Arbeit zur Erstellung dieses Buches haben insbesondere Marcel Ochel und Heiko Röglin mitgewirkt. An der redaktionellen Arbeit zum *Algorithmus der Woche*, die die Grundlage für dieses Buch gelegt hat, waren neben diesen beiden auch Dirk Bongartz, Torsten Sattler, Katrin Twickler und Melanie Winkler beteiligt. Wir möchten allen Autoren und dem Redaktionsteam für ihr Mitwirken danken. Durch das große Engagement aller Beteiligten ist unserer Meinung nach eine ausgesprochen facettenreiche Sammlung von interessanten und informativen Artikeln entstanden, die die besondere Faszination der Informatik spürbar werden lässt.

Die Herausgeber im März 2008

# Inhaltsverzeichnis

**Teil I Suchen und Sortieren**

**Übersicht**
*Martin Dietzfelbinger, Christian Scheideler* . . . . . 3

**1 Binäre Suche**
*Thomas Seidl, Jost Enderle* . . . . . 7

**2 Sortieren durch Einfügen**
*Wolfgang P. Kowalk* . . . . . 15

**3 Schnelle Sortieralgorithmen**
*Helmut Alt* . . . . . 21

**4 Paralleles Sortieren – Parallel geht schnell**
*Rolf Wanka* . . . . . 31

**5 Topologisches Sortieren –**
**Mit welcher Aufgabe meiner ToDo-Liste fange ich an?**
*Hagen Höpfner* . . . . . 43

**6 Texte durchsuchen – aber schnell!**
**Der Boyer-Moore-Horspool Algorithmus**
*Markus E. Nebel* . . . . . 51

**7 Tiefensuche (Ariadne und Co.)**
*Michael Dom, Falk Hüffner, Rolf Niedermeier* . . . . . 61

**8 Der Pledge-Algorithmus:**
**Wie man im Dunkeln aus einem Labyrinth entkommt**
*Rolf Klein, Tom Kamphans* . . . . . 75

**9 Zyklensuche in Graphen**
*Holger Schlingloff* . . . . . 83

**10 PageRank: Was ist wichtig im World Wide Web?**
*Ulrik Brandes, Gabi Dorfmüller* . . . . . 95

**Teil II Rechnen, Verschlüsseln und Codieren**

**Übersicht**
*Berthold Vöcking* . . . . . 105

**11 Multiplikation langer Zahlen (schneller als in der Schule)**
*Arno Eigenwillig, Kurt Mehlhorn* . . . . . 109

**12 Der Euklidische Algorithmus**
*Friedrich Eisenbrand* . . . . . 119

**13 Das Sieb des Eratosthenes:**
**Wie schnell kann man eine Primzahlentabelle berechnen?**
*Rolf Möhring, Martin Oellrich* . . . . . 127

**14 Einweg-Funktionen:**
**Vorsicht Falle – Rückweg nur für Eingeweihte!**
*Rüdiger Reischuk, Markus Hinkelmann* . . . . . 139

**15 Der One-Time-Pad-Algorithmus:**
**Der einfachste und sicherste Verschlüsselungsalgorithmus**
*Till Tantau* . . . . . 149

**16 Public-Key-Kryptographie**
*Dirk Bongartz, Walter Unger* . . . . . 157

**17 Teilen von Geheimnissen**
*Johannes Blömer* . . . . . 171

**18 Poker per E-Mail**
*Detlef Sieling* . . . . . 181

**19 Fingerprinting**
*Martin Dietzfelbinger* . . . . . 193

**20 Hashing**
*Christian Schindelhauer* . . . . . 205

**21 Fehlererkennende Codes: Was ist eigentlich EAN?**
*Alexander Souza, Angelika Steger* . . . . . 213

**Teil III Planen, strategisches Handeln und Computersimulationen**

**Übersicht**
*Helmut Alt, Rüdiger Reischuk* . . . . . 227

**22 Broadcasting: Wie verbreite ich schnell Informationen?**
*Christian Scheideler* . . . . . 229

**23 Zahlen auf Deutsch aussprechen**
*Lothar Schmitz* . . . . . 237

**24 Mehrheitsbestimmung – Wer wird Klassensprecher?**
*Thomas Erlebach* . . . . . 245

**25 Zufallszahlen: Wie kommt der Zufall in den Rechner?**
*Bruno Müller-Clostermann, Tim Jonischkat* . . . . . 255

**26 Gewinnstrategie für ein Streichholzspiel**
*Jochen Könemann* . . . . . 267

**27 Turnier- und Sportligaplanung**
*Sigrid Knust* . . . . . 275

**28 Der Alphabeta-Algorithmus für Spielbäume: Wie bringe ich meinen Computer zum Schachspielen?**
*Burkhard Monien, Ulf Lorenz, Daniel Warner* . . . . . 285

**29 Die Eulertour**
*Michael Behrisch, Amin Coja-Oghlan, Peter Liske* . . . . . 295

**30 Kreise zeichnen mit Turbo**
*Dominik Sibbing, Leif Kobbelt* . . . . . 303

**31 Gauß-Seidel Iteration zur Berechnung physikalischer Probleme**
*Christoph Freundl, Ulrich Rüde* . . . . . 313

**32 Dynamische Programmierung: Evolutionäre Distanz**
*Norbert Blum, Matthias Kretschmer* . . . . . 323

**33 Faires Teilen: Eine Weihnachtsstollengeschichte**
*Raimund Seidel* . . . . . 331

## Teil IV Optimieren

**Übersicht**
*Heribert Vollmer, Dorothea Wagner* . . . . . 343

**34 Kürzeste Wege**
*Peter Sanders, Johannes Singler* . . . . . 345

**35 Minimale aufspannende Bäume (Wenn das Naheliegende das Beste ist. . . )**
*Katharina Skutella, Martin Skutella* . . . . . 353

**36 Maximale Flüsse – Die ganze Stadt will zum Stadion**
*Robert Görke, Steffen Mecke, Dorothea Wagner* . . . . . 361

**37 Partnerschaftsvermittlung**
*Volker Claus, Volker Diekert, Holger Petersen* . . . . . 373

**38 Kleinster umschließender Kreis (Ein Demokratiebeitrag aus der Schweiz?)**
*Emo Welzl* . . . . . 385

**39 Online-Algorithmen: Was ist es wert, die Zukunft zu kennen?**
*Susanne Albers, Swen Schmelzer* . . . . . 389

**40 Bin Packing oder „Wie bekomme ich die Klamotten in die Kisten?“**
*Joachim Gehweiler, Friedhelm Meyer auf der Heide* . . . . . 395

**41 Das Rucksackproblem**
*Rene Beier, Berthold Vöcking* . . . . . 405

**42 Das Travelling Salesman Problem**
*Stefan Näher* . . . . . 413

**43 Simulated Annealing**
*Peter Rossmanith* . . . . . 423

**Die Autoren** . . . . . 433

# Teil I

## Suchen und Sortieren

# Übersicht

Martin Dietzfelbinger und Christian Scheideler

Technische Universität Ilmenau
Technische Universität München

Jedes Kind weiß, dass man – zumindest ab einer gewissen Anzahl – viel einfacher Dinge wiederfinden kann, wenn man Ordnung hält. Wir Menschen verstehen unter Ordnung halten, dass wir die Dinge, die wir besitzen, in Kategorien einteilen und diesen Kategorien dann feste Orte für deren Aufbewahrung zuweisen, die wir uns merken können. Strümpfe mögen wir dabei einfach ungeordnet in einer Schublade verschwinden lassen, aber für andere Dinge wie z. B. DVDs empfiehlt es sich, ab einer bestimmten Menge die Titel zu sortieren, damit man jede DVD schnell wiederfinden kann. Aber was genau heißt schnell, und wie schnell können wir Dinge sortieren oder wiederfinden? Mit diesem wichtigen Thema werden sich die ersten Kapitel in Teil 1 beschäftigen.

Das Kap. 1 macht den Anfang mit einer schnellen Suchstrategie namens *Binäre Suche*. Diese Suchstrategie setzt voraus, dass die Objekte, in denen gesucht werden soll (im Kapitel sind das CDs), bereits sortiert worden sind. Danach werden in Kap. 2 einige einfache Sortierstrategien vorgestellt. Diese führen eine Reihe von paarweisen Vergleichen und Umsortierungen der Objekte durch, bis am Ende alle Objekte sortiert sind. Allerdings sind die vorgestellten Verfahren nur für eine kleine Anzahl an Objekten zu empfehlen, da der Sortieraufwand für eine große Anzahl an Objekten im schlimmsten Fall beträchtlich sein kann. In Kap. 3 geht es dann um zwei Sortierstrategien, die auch für eine sehr große Anzahl an Objekten noch schnell arbeiten. Danach, in Kap. 4, geht es um ein „paralleles“ Sortierverfahren. Dabei bedeutet „parallel“, dass man viele der Vergleiche gleichzeitig durchführen kann und so erheblich weniger Zeit benötigt als bei einem Verfahren, in dem die Vergleiche hintereinander durchgeführt werden müssen. Solche Verfahren sind besonders dann interessant, wenn man einen Computer mit vielen Prozessoren hat, die gleichzeitig arbeiten können, oder gar spezielle Maschinen, die nichts anderes tun können als Sortieren. Die Liste der Sortierverfahren wird in Kap. 5 abgerundet mit einem Verfahren, das eine „Topologische Sortierung“ herstellt. Eine topologische Sortierung wird z. B. dann benötigt, wenn man eine Reihe

von Aufträgen hat, für die zeitliche Abhängigkeiten existieren, der Art, dass Auftrag A erledigt werden muss, bevor Auftrag B begonnen werden kann. Das Ziel der topologischen Sortierung ist es dann, eine Anordnung der Aufträge zu finden, so dass die Aufträge hintereinander abgearbeitet werden können, ohne dabei eine der zeitlichen Abhängigkeiten zwischen zwei Aufträgen zu verletzen.

In Kap. 6 kehren wir wieder zurück zum Suchproblem. Diesmal beschäftigen wir uns mit der Textsuche. Konkret geht es darum zu überprüfen, ob ein Text ein bestimmtes Wort enthält. Ein Mensch kann das natürlich effizient testen (zumindest für kurze Suchworte und nicht allzu lange Texte), aber ein effizientes Suchverfahren für einen Computer zu entwickeln ist gar nicht so einfach. Im Kapitel wird ein Suchverfahren vorgestellt, das in der Praxis sehr schnell arbeitet, auch wenn es für pathologische Fälle eventuell viel Zeit benötigt.

Im weiteren Verlauf von Teil 1 geht es um Suchaufgaben in Welten, die man nicht als Ganzes überblicken kann. Wie kann man ein Labyrinth aus Gängen und Kreuzungen vollständig durchsuchen, ohne im Kreis zu laufen und ohne Wege mehrfach zu betreten? Kap. 7 zeigt, wie man dieses Problem mit einem grundlegenden, vielseitig verwendbaren Algorithmus lösen kann, der *Tiefensuche* heißt und die Möglichkeit nutzt, Markierungen (Kreidestriche an der Wand) zu setzen. Interessanterweise hilft Tiefensuche auch, wenn man einen Teil des World Wide Web systematisch durchsuchen möchte, ohne sich zu wiederholen, oder wenn man ein Labyrinth erzeugen möchte. Im anschließenden Kap. 8 geht es noch einmal um Labyrinthe, aber dieses Mal sucht man von irgendeiner Stelle in einem Labyrinth den Ausgang und hat dazu keine anderen Hilfsmittel als einen Kompass (also einen guten Richtungssinn). Diesmal kann man also keine Markierungen hinterlassen. Der Pledge-Algorithmus, der diese Problem löst, ist sehr einfach zu formulieren und auszuführen – nur der Nachweis, dass er wirklich funktioniert, braucht ein bisschen Überlegung. Dieser Algorithmus kann auch von Robotern verwendet werden, die den Ausgang aus einem mit großen und kleinen Hindernissen vollgestellten Raum finden sollen. In Kap. 9 geht es um eine spezielle Anwendung der Tiefensuche, nämlich um das Finden von Kreisen in Labyrinthen oder Wegenetzen oder in Beziehungsnetzen, die Abhängigkeiten darstellen sollen. Manchmal ist es sehr wichtig, Kreise zu finden, um „Verklemmungen“ aufzulösen, wo Leute oder Vorgänge so aufeinander warten, dass man nie fertig werden kann. Sogar die Struktur aller Kreise in so einem Beziehungsgeflecht kann man auf überraschend einfache und effiziente Weise ermitteln.

Den ersten Teil schließt Kap. 10 ab, in dem es eigentlich nicht um ein Suchproblem geht, sondern um ein Problem, mit dem Suchmaschinen im World Wide Web konfrontiert sind. Die Benutzerin gibt ein paar Suchbegriffe ein und erwartet, dass die Suchmaschine ihr eine Liste von Seiten präsentiert, aus der sie die richtigen auswählen kann. Nun passen oft Tausende oder Hunderttausende von Webseiten „irgendwie“ zu den Suchbegriffen, aber man wünscht

sich, dass vorne in der Liste möglichst „wichtige“, „interessante“ oder „relevante“ Seiten angezeigt werden. Die Suchmaschinen tun dies auch, aber wie? In Kap. 10 wird das Prinzip erklärt. Dabei erfährt man auch, was Trampelpfade im World Wide Web sind und wie man mit ihnen rechnen kann, ohne sie abzulaufen.

# 1

# Binäre Suche

Thomas Seidl und Jost Enderle

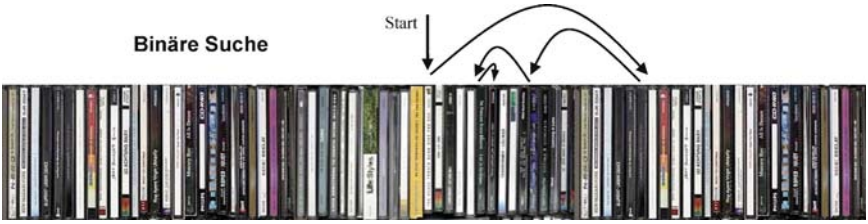
RWTH Aachen

Wo ist nur die neue Nelly-CD? Meine große Schwester Linda mit ihrem Ordnungsfimmel hat sie bestimmt schon wieder ins CD-Regal gepackt. Ich habe ihr schon tausend Mal gesagt, dass sie meine neuen CDs draußen liegen lassen soll. Jetzt darf ich wieder alle 500 CDs im Regal einzeln anschauen. Das dauert ewig, bis ich die alle durchsucht habe!

OK, wenn ich Glück habe, finde ich die CD vielleicht schon früher und muss doch nicht auf alle Cover draufschauen. Im schlimmsten Fall hat aber Linda die CD mal wieder ihrer Freundin ausgeliehen: Dann darf ich alle durchsuchen und am Ende wieder Radio hören.

Aaliyah, AC/DC, Alicia Keys, ... hmmm, Linda scheint die CDs nach Künstlern sortiert zu haben. Damit sollte meine Nelly-CD wohl leichter zu finden sein. Ich greife einfach mal mitten rein. „Kelly Family“, das war wohl zu weit vorne, ich muss weiter hinten suchen. „Rachmaninov“, das war wieder zu weit hinten, mal ein bisschen weiter links nachschauen“ ... „Lionel Hampton“. Noch ein bisschen nach rechts ... „Nancy Sinatra“ ... „Nelly“!

Das ging ja jetzt doch ganz flott! Mit der Sortierung reicht es, ein paar mal hin- und herzuspringen, bis man die CD gefunden hat. Auch wenn die CD nicht im Regal gewesen wäre, hätte man das schnell gemerkt. Wenn wir jetzt aber mal 10.000 CDs haben, muss ich dann wahrscheinlich doch wieder ein paar hundert Mal hin- und herspringen und die CDs anschauen. Ob man das irgendwie ausrechnen kann?




## Sequenzielle Suche

Linda studiert ja seit letztem Jahr Informatik, die hat sicher irgendwo Unterlagen herumliegen, in denen auch etwas Informatives drinsteht. Mal sehen ... unter „Suchalgorithmen“ bin ich wohl richtig. Hier wird beschrieben, wie man einen Eintrag in einer gegebenen Menge (hier: CDs) anhand eines Schlüsselwerts (hier: Künstler) sucht. Was ich zuerst versucht habe, nennt sich wohl „sequenzielle“ oder „lineare Suche“. Wie schon vermutet, muss man hier durchschnittlich die Hälfte der Einträge durchsehen, bis man den gesuchten Schlüsselwert gefunden hat. Die Anzahl der Suchschritte steigt proportional mit der Anzahl der Einträge, d. h., wenn man doppelt so viele Einträge hat, muss man auch doppelt solange suchen.

## Binäre Suche

Mein zweites Suchverfahren scheint auch einen extra Namen zu haben, „binäre Suche“. Bei einem gegebenen Suchschlüssel und einer sortierten Liste von Einträgen fängt man bei dem mittleren Eintrag an und vergleicht dessen Schlüssel mit dem Suchschlüssel. Hat man den gesuchten Eintrag dann schon gefunden, ist die Suche beendet. Ansonsten wird dasselbe entweder für die linke oder die rechte Hälfte der Einträge gemacht, je nachdem, ob der gelesene Schlüssel größer oder kleiner war als der Suchschlüssel, und zwar so oft, bis man entweder den Eintrag gefunden hat oder bis keine Halbierung des Suchraums mehr möglich ist (d. h., man ist an der Stelle angekommen, an der der Eintrag eigentlich stehen müsste). In den Unterlagen meiner Schwester ist auch ein entsprechender Programm-Code abgedruckt.

Dabei ist A eine „Reihe“; das ist eine Liste von Daten, in dem die Einträge durchnummeriert sind, so wie im Regal die CD-Positionen. Der fünfte Eintrag in einer solchen Reihe wird dann z. B. mit A[5] bezeichnet. Wenn unser Regal also 500 CDs enthält und wir nach dem Schlüssel „Nelly“ suchen, müssen wir BinaereSuche(Regal, „Nelly“, 1, 500) aufrufen, um die Position der gesuchten CD zu finden. Bei der Ausführung des Programms wird dann zunächst „links“ auf 251 gesetzt, danach „rechts“ auf 375 usw.

Die Funktion BinaereSuche gibt die Position von „Schluessel“ in Reihe „A“ zwischen „links“ und „rechts“ aus.

```
1  function BinaereSuche (A, Schluessel, links, rechts)
2  while links ≤ rechts do
3      mitte := (links + rechts)/2     {Mitte bestimmen, Ergebnis runden}
4      if A[mitte] = Schluessel then return mitte
5      if A[mitte] > Schluessel then rechts := mitte − 1
6      if A[mitte] < Schluessel then links := mitte + 1
7  endwhile
8  return „nicht gefunden“
```

## Rekursive Implementierung

In Lindas Unterlagen ist noch ein zweiter Algorithmus für die binäre Suche angegeben. Seltsam, wieso benötigt man für dieselbe Funktion unterschiedliche Algorithmen? Hier steht, dass der zweite Algorithmus *Rekursion* benutzt; was ist nun das schon wieder?

Gleich mal nachschlagen…: „Eine rekursive Funktion ist eine Funktion, die durch sich selbst definiert ist bzw. sich selbst aufruft.“ Als Beispiel ist die *Summenfunktion* genannt, die folgendermaßen definiert ist:

$$\text{sum}(n) = 1 + 2 + \ldots + n$$

Es werden also alle natürlichen Zahlen bis zu der Obergrenze $n$ aufaddiert, für $n = 5$ erhalten wir also:

$$\text{sum}(4) = 1 + 2 + 3 + 4 = 10$$

Möchte man das Ergebnis der Summenfunktion für ein bestimmtes $n$ berechnen und kennt schon das Ergebnis für $n-1$, muss man einfach $n$ zu diesem Ergebnis hinzuaddieren:

$$\text{sum}(n) = \text{sum}(n-1) + n$$

Eine solche Definition bezeichnet man als *Rekursionsschritt*. Um auf diese Weise die Summenfunktion für ein $n$ berechnen zu können, benötigt man noch den *Rekursionsanfang* für das kleinste $n$:

$$\text{sum}(1) = 1$$

Mit diesen Definitionen lässt sich nun die Summenfunktion für ein beliebiges $n$ berechnen:

$$\begin{aligned}\text{sum}(4) &= \text{sum}(3) + 4\\ &= (\text{sum}(2) + 3) + 4\\ &= ((\text{sum}(1) + 2) + 3) + 4\\ &= ((1 + 2) + 3) + 4\\ &= 10\end{aligned}$$

Ähnlich verhält es sich mit der rekursiven Definition für die binäre Suche: Anstatt wiederholt die Schleife zu durchlaufen (*iterative* Implementierung), ruft sich hier die Funktion im Funktionsrumpf selbst auf:

Die Funktion BinSucheRekursiv gibt die Position von „Schluessel“ in Reihe „A“ zwischen „links“ und „rechts“ aus.

```
1  function BinSucheRekursiv (A, Schluessel, links, rechts)
2  if links > rechts return „nicht gefunden“
3  mitte := (links + rechts)/2      {Mitte bestimmen, Ergebnis runden}
4  if A[mitte] = Schluessel then return mitte
5  if A[mitte] > Schluessel then
6      return BinSucheRekursiv (A, Schluessel, links, mitte − 1)
7  if A[mitte] < Schluessel then
8      return BinSucheRekursiv (A, Schluessel, mitte + 1, rechts)
```

Dabei ist A wieder die zu durchsuchende Reihe, „Schluessel“ der gesuchte Schlüssel und „links“ und „rechts“ die linke und rechte Grenze des zu durchsuchenden Bereichs in A. Soll der Eintrag „Nelly“ in der Reihe „Regal“ mit 500 Einträgen gesucht werden, wird also zunächst wieder BinSucheRekursiv (Regal, „Nelly“, 1, 500) aufgerufen. Anders als bei der iterativen Lösung werden dann jedoch die Grenzen nicht innerhalb einer Schleife aufeinander zugeschoben, sondern einfach die BinSucheRekursiv-Funktion mit entsprechend angepassten Grenzen rekursiv aufgerufen. Man erhält z. B. die folgende Sequenz von Aufrufen:

BinSucheRekursiv(Regal, „Nelly“, 1, 500)
BinSucheRekursiv(Regal, „Nelly“, 251, 500)
BinSucheRekursiv(Regal, „Nelly“, 251, 374)
BinSucheRekursiv(Regal, „Nelly“, 313, 374)
BinSucheRekursiv(Regal, „Nelly“, 344, 374)
...

## Anzahl der Suchschritte

Jetzt bleibt noch die Frage, wie viele Suchschritte wir eigentlich durchführen müssen, bis wir den richtigen Eintrag gefunden haben. Wenn wir Glück haben, finden wir den Eintrag schon im ersten Schritt; gibt es den gesuchten Eintrag nicht, müssen wir so oft springen, bis wir die Stelle erreicht haben, an der der Eintrag eigentlich stehen müsste. Wir müssen also überlegen, wie oft man die Liste der Einträge halbieren kann, oder anders herum, wie viele Einträge wir mit einer bestimmten Anzahl von Vergleichen durchsuchen können. Wenn wir davon ausgehen, dass der gesuchte Eintrag in der Liste enthalten ist, können wir mit 1 Vergleich 2 Einträge durchsuchen, mit 2 Vergleichen dann 4 Einträge, mit 3 Vergleichen schon 8 Einträge. Mit $k$ Vergleichen können wir also $2 \cdot 2 \cdot \cdots \cdot 2$ ($k$-mal) $= 2^k$ Einträge durchsuchen. Das wären dann für 10

Vergleiche schon 1024 Einträge, für 20 Vergleiche über eine Million und für 30 Vergleiche über eine Milliarde Einträge! Wir benötigen jeweils eine zusätzliche Anfrage, wenn der gesuchte Eintrag nicht in der Liste vorhanden ist. Will man „rückwärts“ rechnen, d. h. aus der Anzahl von Einträgen die notwendige Anzahl von Vergleichen bestimmen, muss man die Umkehrfunktion zur Zweier-Potenz verwenden, man nennt das den Zweier-Logarithmus und bezeichnet die zugehörige Funktion mit $\log_2$. Allgemein gilt für den Logarithmus:

$$\text{Wenn } a = b^x, \text{ dann ist } x = \log_b a \tag{1}$$

Für den Zweier-Logarithmus ist $b = 2$ und man hat also

$$\begin{aligned}
2^0 &= 1, & \log_2 1 &= 0\\
2^1 &= 2, & \log_2 2 &= 1\\
2^2 &= 4, & \log_2 4 &= 2\\
2^3 &= 8, & \log_2 8 &= 3\\
&\vdots & &\vdots\\
2^{10} &= 1.024, & \log_2 1.024 &= 10\\
&\vdots & &\vdots\\
2^{13} &= 8.192, & \log_2 8.192 &= 13\\
2^{14} &= 16.384, & \log_2 16.384 &= 14\\
&\vdots & &\vdots\\
2^{20} &= 1.048.576, & \log_2 1.048.576 &= 20
\end{aligned}$$

Wenn also mit $k$ Vergleichen $2^k = N$ Einträge durchsucht werden können, braucht man $\log_2 N = k$ viele Vergleiche für $N$ Einträge. Wenn in unserem Regal also 10.000 CDs wären, dann haben wir $\log_2 10.000 \approx 13{,}29$. Da es keine „halben Vergleiche“ gibt, ergeben sich 14 Vergleiche!

Möchte man die Anzahl der Suchschritte bei der binären Suche weiter reduzieren, kann man versuchen, genauer zu erraten, wo sich der gesuchte Schlüssel innerhalb des aktuell betrachteten Bereichs befinden könnte (statt einfach immer das mittlere Element zu verwenden). Sucht man beispielsweise im sortierten CD-Regal nach einem Interpreten, dessen Anfangsbuchstabe weit vorne im Alphabet liegt, z. B. „Eminem“, kann man auch im vorderen Bereich des Regals mit der Suche beginnen, bei „Roy Black“ dagegen relativ weit hinten. Für eine weitere Verbesserung der Suche könnte man berücksichtigen, dass manche Anfangsbuchstaben (z. B. D und S) viel häufiger vorkommen als andere (z. B. X und Y).

## Ratespiel

Heute Abend werde ich Linda mal auf die Probe stellen und sie eine Zahl zwischen 1 und 1.000 raten lassen. Wenn sie in der Uni aufgepasst hat, dürfte

sie dafür nicht mehr als 10 „Ja/Nein“-Fragen benötigen. (Für das Raten einer Zahl zwischen 1 und 16 mit maximal 4 Fragen kann man wie in dem Bild unten vorgehen.)

Um dabei nicht immer dieselbe langweilige Frage „Ist die Zahl größer/kleiner als ...?“ stellen zu müssen, kann man auch mal ein „Ist die Zahl (un)gerade?“ einstreuen. Dadurch fällt auch die Hälfte der verbleibenden Möglichkeiten weg. Ebenso kann gefragt werden „Ist die Zehner- oder Hunderterstelle (un)gerade?“. Auch damit erreicht man jeweils (ungefähr) eine Halbierung des Suchraums. Sind alle Stellen durchgetestet, muss man jedoch wieder auf die übliche Halbierung zurückgreifen (wobei dann die bereits ausgeschlossenen Zahlen zu berücksichtigen sind).

Ganz einfach wird die Sache, wenn man sich die Zahl als Binärzahl vorstellt. Während im Dezimalsystem Zahlen als Summe von Vielfachen von Zehnerpotenzen dargestellt werden, z. B.

$$\begin{aligned} 107 &= \mathbf{1} \cdot 10^2 + \mathbf{0} \cdot 10^1 + \mathbf{7} \cdot 10^0 \\ &= \mathbf{1} \cdot 100 + \mathbf{0} \cdot 10 + \mathbf{7} \cdot 1, \end{aligned}$$

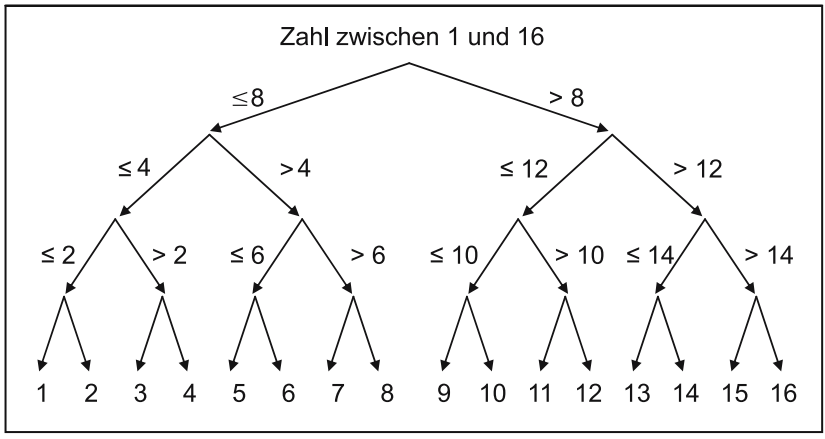
werden Zahlen im Binärsystem als Vielfache von Zweierpotenzen dargestellt:

$$\begin{aligned} 107 &= \mathbf{1} \cdot 2^6 + \mathbf{1} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0 \\ &= \mathbf{1} \cdot 64 + \mathbf{1} \cdot 32 + \mathbf{0} \cdot 16 + \mathbf{1} \cdot 8 + \mathbf{0} \cdot 4 + \mathbf{1} \cdot 2 + \mathbf{1} \cdot 1 \end{aligned}$$

Die Binärdarstellung von 107 ist also 1101011. Um eine Zahl mithilfe der Binärdarstellung zu erraten, reicht es zu wissen, wie viele Binärstellen die zu erratende Zahl maximal haben kann. Die Anzahl der Binärstellen kann man wieder einfach mit dem Zweierlogarithmus berechnen. Soll z. B. eine Zahl zwischen 1 und 1000 erraten werden, berechnet man

$$\log_2 1000 \approx 9{,}97 \text{ (aufrunden!)},$$

es werden also 10 Stellen benötigt. Danach reichen 10 Fragen: „Ist die erste Binärstelle gleich 1?“, „Ist die zweite Binärstelle gleich 1?“, „Ist die dritte



Binärstelle gleich 1?“ usw. Danach sind alle Stellen der Binärdarstellung der Zahl bekannt und man muss nur noch ins Dezimalsystem umrechnen; das kann dann wieder der Taschenrechner übernehmen.

## Zum Weiterlesen

1. Robert Sedgewick: *Algorithmen*. 2. Auflage, 1993.
   Dieses Buch beschreibt ab S. 236 die binäre Suche.
2. Implementierung der binären Suche in Java und C:
   `http://de.wikipedia.org/wiki/Binäre_Suche`
3. Binäre Suche im Java SDK:
   `http://java.sun.com/javase/6/docs/api/java/util/Arrays.html#binarySearch(long[],long)`
4. Um die binäre Suche auf einer Menge von Einträgen ausführen zu können, müssen diese sortiert sein. Wie man die Einträge möglichst schnell sortieren kann, wird in den folgenden Kapiteln erklärt:
   - Kapitel 2 (Sortieren durch Einfügen)
   - Kapitel 3 (Schnelle Sortieralgorithmen)
   - Kapitel 4 (Paralleles Sortieren)

## Danksagung

Die Autoren danken Christoph Brochhaus für die technische Unterstützung und hilfreiche Anmerkungen.

# 2

# Sortieren durch Einfügen

Wolfgang P. Kowalk

Universität Oldenburg

Schon wieder Aufräumen, dabei habe ich doch erst neulich ... Nun ja, wenn alles durcheinander liegt, dann lohnt es sich vielleicht doch, mal eben Ordnung zu schaffen; man findet alles schneller wieder. Lass uns also mal die Bücher auf dem Regal alphabetisch nach ihren Titeln sortieren, so dass man jedes gleich zur Hand hat, wenn man es braucht.

Doch wie schafft man am schnellsten Ordnung? Man kann verschiedene Ansätze verfolgen. So kann man wiederholt alle Bücher durchgehen, und jedesmal wenn man zwei aufeinanderfolgende Bücher sieht, die falsch herum stehen, vertauscht man sie. Das funktioniert, weil irgendwann keine zwei Bücher mehr verkehrt stehen, aber es kann lange dauern. Man kann auch zunächst das Buch mit dem (alphabetisch) ersten Titel heraussuchen und dieses nach ganz links ins Regal stellen. Danach sucht man sich das Buch mit dem ersten Titel aus den restlichen Büchern heraus und stellt es neben das erste Buch. Das macht man dann so weiter, bis alle Bücher sortiert sind. Auch diese Methode funktioniert, kann aber auch lange dauern, da sie sehr viel vorhandene Information immer wieder „vergisst“ und nicht ausnutzt; jedes Mal müssen immer wieder die gleichen Buchtitel gelesen werden. Versuchen wir also etwas anderes.

Die folgende Idee scheint sehr natürlich zu sein. Wir bringen zuerst die beiden Bücher ganz links im Regal in ihre richtige Reihenfolge, dann die ersten drei, dann die ersten vier, und so weiter, bis wir schließlich alle Bücher in sortierter Reihenfolge stehen haben. Dabei können wir in jeder Runde auf das vorher Erreichte zurückgreifen. Wie geht das ganz genau vor sich? Das erste Buch steht für sich allein gesehen am richtigen Platz. Jetzt nehmen wir das zweite Buch hinzu und vertauschen es mit dem ersten, falls es links von diesem stehen muss. Dann nehmen wir das dritte Buch und ordnen es in der richtigen Stelle bezüglich der ersten beiden Bücher ein. Danach nehmen wir das vierte Buch hinzu, sortieren es an die richtige Stelle unter den ersten dreien und so weiter. Allgemein nehmen wir an, dass alle Bücher links vom aktuellen Buch (Position $i$ von links) bereits sortiert sind. Dann nehmen wir das aktuelle Buch an dieser Position, suchen seinen korrekten Platz in den Büchern links

davon und fügen es an dieser Position ein. Dazu müssen die Bücher rechts von diesem richtigen Platz etwas nach rechts verschoben werden. Nun geht es weiter mit der nächsten Position. Nachdem das letzte Buch von ganz rechts im Regal an die richtige Stelle gewandert ist, sind alle Bücher sortiert. Dieses Verfahren führt recht schnell zu einer sortierten Bücherreihe, besonders wenn wir das Verfahren „Binäre Suche“ aus Kap. 1 benutzen, um schnell den richtigen Platz des aktuellen Buches in der schon sortierten Bücherreihe links davon zu finden. Nur das Verschieben der Bücher rechts vom „richtigen Platz“ benötigt eventuell ziemlich viel Kraft. Bei einer sehr langen Buchreihe reicht irgendwann die Kraft nicht mehr aus, alle Bücher gleichzeitig zu verschieben, und man verschiebt wohl dann doch besser Buch für Buch.

Wie kann man unser intuitives Verfahren nun so umsetzen, dass es für jede Anzahl von Büchern durchführbar ist? Statt der Buchtitel schreiben wir im folgenden Nummern, weil die Verfahren dann einfacher zu erklären sind.

In Abb. 2.1 sind die fünf Bücher 1, 6, 7, 9, 11 ganz links bereits sortiert; wird das Buch mit der Nummer 5 hinzugenommen, so steht es offensichtlich falsch. Um es an die richtige Position zu bringen, können wir es zunächst mit dem Buch mit der Nummer 11 vertauschen, dann mit dem mit der Nummer 9 usw., bis es rechts vom Buch mit der Nummer 1 an seiner richtigen Position angekommen ist. Dann fahren wir mit dem Buch mit der Nummer 3 fort und sortieren es durch paarweise Vertauschungen richtig ein, usw. Offenbar kommen dadurch alle Bücher nacheinander an ihren richtigen Platz (siehe Abb. 2.2).

Wie kann man so etwas jetzt programmieren? Das folgende Programm macht das! Es benutzt ein Zahlenfeld (oder Array) $A$, dessen Zellen durchnummeriert sind. Unter $A[i]$ versteht man den Wert an der $i$-ten Stelle des Feldes $A$. Wenn wir $n$ Bücher haben, so brauchen wir ein Zahlenfeld der Länge $n$ mit den Stellen $A[1]$, $A[2]$, $A[3]$, …, $A[n-1]$, $A[n]$, um alle Buchtitel bzw. Buchnummern zu speichern. Der Algorithmus sieht dann folgendermaßen aus.

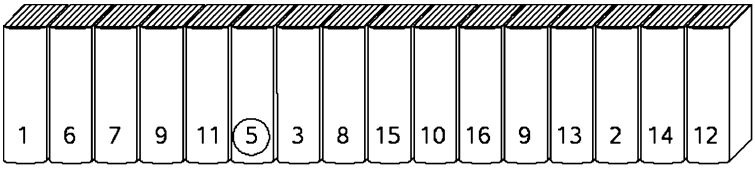



**Abb. 2.1.** Die ersten fünf Bücher sind sortiert

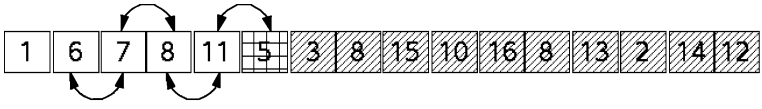



**Abb. 2.2.** Buch „5“ wird an den richtigen Platz gebracht

**Benachbarte Bücher vertauschen:**

```
1   Gegeben: A: Feld mit n Einträgen
2   for i := 2 to n do
3       j := i;      // das Buch an Position i ist aktuell
          // solange korrekte Position des aktuellen Buchs noch nicht erreicht
4       while j ≥ 2 and A[j − 1] > A[j] do
5           Hand := A[j];      // tausche aktuelles Buch mit linkem Nachb.
6           A[j] := A[j − 1];
7           A[j − 1] := Hand;
8           j := j − 1
9       endwhile
10  endfor
```

Wie lange dauert jetzt wohl das Aufräumen? Nehmen wir einmal den schlechtesten Fall an. Dann sind alle Bücher genau verkehrt herum aufgestellt, also das mit der kleinsten Nummer steht ganz rechts, daneben das mit der zweitkleinsten Nummer usw. Was macht unser Algorithmus dann? Er fängt von links an und vertauscht das zweite Buch mit dem ersten, das dritte mit den ersten beiden, das vierte mit den ersten dreien, usw., bis schließlich das letzte mit $n-1$ Büchern zu vertauschen ist. Die Anzahl der Vertauschungen ist also

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

Diese Formel lässt sich leicht anhand von Abb. 2.3 einsehen. Es gibt in dem Rechteck $n \cdot (n-1)$ Felder, und davon wird gerade die Hälfte für das Vergleichen

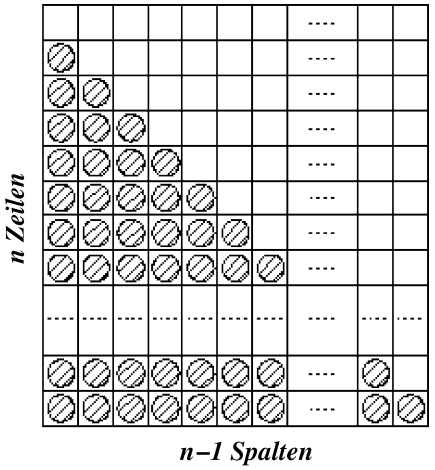



**Abb. 2.3.** Berechnung der Anzahl der Vertauschungen

und Vertauschen verwendet. Das Bild stellt aber den absolut schlechtesten Fall dar. Für den allgemeinen Fall überlegen wir: Um das Buch an Stelle $i$ einzusortieren, braucht man keinesfalls mehr als $i-1$ Vertauschungen. Also gibt es für keine Ausgangssituation mehr als $\frac{1}{2}n(n-1)$ Vertauschungen. Unser Algorithmus ist umso besser, je mehr Bücher bereits richtig stehen, was z. B. der Fall ist, wenn die Bücherreihe schon einmal sortiert worden ist und einige herausgenommene Bücher einfach wahllos wieder zurückgestellt worden sind, wie das vielleicht dein kleiner Bruder gerne tut!

Sicherlich hast du schon bemerkt, dass unser Sortierverfahren umständlicher ist, als es eigentlich sein muss. (Genau genommen ist es eine spezielle Variante des am Anfang erwähnten Verfahrens, immer wieder benachbarte Bücher zu vertauschen, wenn sie falsch herum stehen ...) Wie wir schon oben überlegt haben, muss man ja nicht unbedingt die Bücher durchtauschen, sondern man kann auch einfach die schon einsortierten Bücher nach rechts verschieben, bis der Platz an der richtigen Position des aktuell einzusortierenden Buches geschaffen worden ist, wie in Abb. 2.4 dargestellt.

Statt $k$-mal zwei Bücher zu vertauschen, verschieben wir dann $(k+1)$-mal ein Buch, was offenbar weniger aufwändig ist, da *einmaliges* Vertauschen *drei* Verschiebeoperationen benötigt. Als Algorithmus sieht das dann so aus:

Bücher sortieren durch Einfügen:

```
1   Gegeben: A: Feld mit n Einträgen;
2   for i := 2 to n do
      // ordne Buch an Position i durch Verschiebungen richtig ein
3     Hand := A[i];      // nimm aktuelles Buch in die Hand
4     j := i − 1;
      // solange korrekte Position des aktuellen Buchs noch nicht erreicht
5     while j ≥ 1 and A[j] > Hand do
6       A[j + 1] := A[j];      // schiebe Buch an Position j nach rechts
7       j := j − 1
8     endwhile
9     A[j] := Hand      // füge aktuelles Buch an der richtigen Stelle ein
10  endfor
```

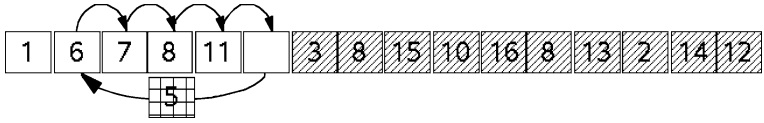



**Abb. 2.4.** Berechnung der Anzahl der Vertauschungen

Weitere Verbesserungen dieses Sortierverfahrens, wie das gleichzeitige Einfügen von mehr als einem Buch über einen einzelnen Verschiebedurchgang, sowie Animierungen dieser Verfahren findest du auf der folgenden Webseite:

`http://einstein.informatik.uni-oldenburg.de/forschung/animAlgo/`

Vielleicht fragst du dich an dieser Stelle, ob man nicht doch irgendwie die Bücher im Computer gleichzeitig verschieben könnte, um dem aktuell einzusortierenden Buch Platz zu schaffen? Das geht zwar in der realen Welt bis zu einer gewissen Anzahl an Büchern gut, aber der Computer kann so etwas leider nicht auf einmal machen. Allerdings sind in der Vergangenheit spezielle Maschinen entwickelt worden, wie die berühmte Hollerith-Maschine zur Auswertung der amerikanischen Volkszählung, die solche Verschiebungen sozusagen parallel durchführen können. Mehr dazu findest du in Kap. 4.

Auch wenn unser Sortierverfahren also in „normalen“ Computern viel Zeit durch die bis zu $\frac{1}{2}n^2$ Verschiebungen verbrauchen kann, wird er trotzdem gerne verwendet, wenn die Anzahl der zu sortierenden Objekte nicht sehr groß ist oder wenn man damit rechnet, dass die Eingabe „fast“ sortiert ist. Angenehm ist, dass er extrem einfach zu implementieren ist. Im folgenden Kap. 3 werden mit MergeSort und QuickSort wesentlich effizientere Sortierverfahren vorgestellt, die dafür aber etwas schwieriger zu verstehen und zu implementieren sind.

## Zum Weiterlesen

1. Das oben vorgestellte Verfahren heißt übrigens auf englisch „Insertion Sort“. Es findet sich in fast jedem Standardwerk über Grundlagen der Algorithmen, beispielsweise ausführlich in: Robert Sedgewick: *Algorithmen in C++*. Pearson Studium, 2002.
2. W.P. Kowalk: *System, Modell, Programm*. Spektrum Akademischer Verlag, 1996 (ISBN 3-8274-0062-7).

# 3

# Schnelle Sortieralgorithmen

Helmut Alt

Freie Universität Berlin

Wie wichtig das Sortieren ist, wurde schon in Kap. 2 beschrieben. Eine effiziente Suche in einer Menge von Daten, wie die in Kap. 1 vorgestellte Binärsuche, ist nur möglich, wenn die Menge vorher sortiert wurde. Stellt euch z. B. die Suche im Telefonbuch von Berlin vor, wenn dieses nicht alphabetisch sortiert wäre. Bei diesem Beispiel haben wir es, wie oft in der Praxis, mit Millionen von Objekten zu tun, die zu sortieren sind. Deswegen ist es wichtig, *effiziente* Sortieralgorithmen zu finden, d. h. solche, die auch bei großen Datenmengen relativ kurze Laufzeiten haben, und diese können für verschiedene Algorithmen für das gleiche Problem sehr unterschiedlich ausfallen.

In diesem Abschnitt stellen wir euch daher zwei Sortieralgorithmen vor, die zunächst recht ungewöhnlich erscheinen, die aber, falls man sehr große Mengen von Objekten sortieren will, eine viel schnellere Laufzeit haben als das in Kap. 2 vorgestellte *Sortieren durch Einfügen*.

In der Beschreibung der Algorithmen formulieren wir das Problem einfachheitshalber so, dass wir mit Zahlen beschriftete Karten sortieren. Wie *Sortieren durch Einfügen* funktionieren diese Algorithmen aber nicht nur für Zahlen, sondern auch z. B. fürs Sortieren von Büchern nach Titeln oder allgemein für alle Objekte, die sich miteinander der Größe nach vergleichen lassen. Auch braucht man nicht unbedingt einen Computer, um diese Algorithmen auszuführen, sondern man kann z. B. auch eine Menge von Paketen dem Gewicht nach sortieren, indem man nach diesen Algorithmen vorgeht und das Gewicht zweier Pakete jeweils mit einer Balkenwaage vergleicht. Der Autor benutzt auch regelmäßig Algorithmus 1, um die Klausuren von Studenten alphabetisch nach Namen zu sortieren.

Bewusst werden deswegen die Algorithmen auch zunächst umgangssprachlich beschrieben anstatt mit einem Programm in einer Programmiersprache oder mit Pseudocode.

## 3.1 Die Algorithmen

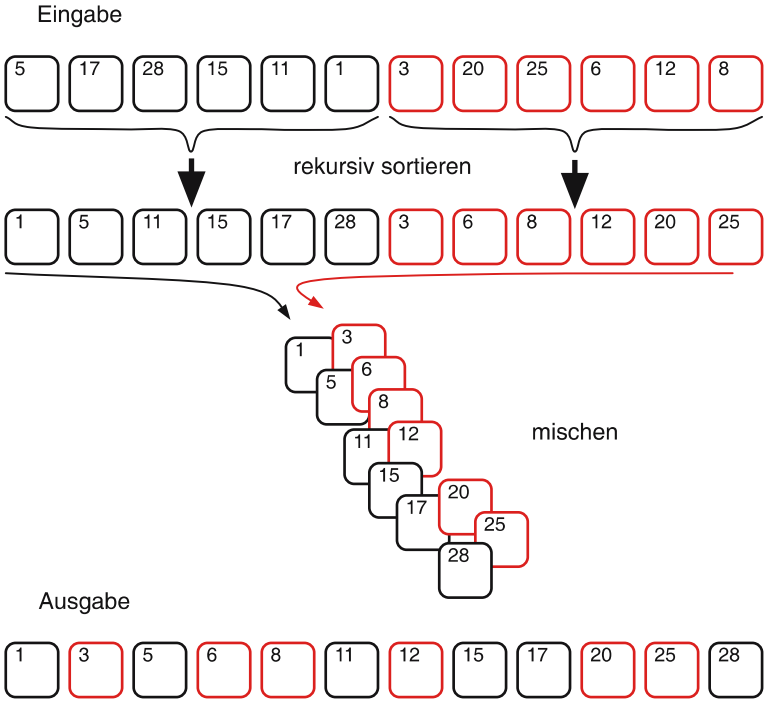
Der Einfachheit halber stelle dir das Problem so vor, dass du von einem Meister einen Stapel von Karten erhältst, die jeweils mit einer Zahl beschriftet sind. Du sollst diese Karten von unten nach oben aufsteigend sortieren und dem Meister zurückgeben.

Dies wird so gemacht:

**Algorithmus 1**

1. Falls der Stapel nur aus einer Karte besteht, gib ihn sofort zurück, andernfalls:
2. Teile den Stapel in zwei möglichst gleich große Teile. Gib jeden Teil je einem Gehilfen mit der Bitte, ihn *rekursiv*, d. h. ebenfalls genau nach dem hier beschriebenen Verfahren zu sortieren.
3. Warte, bis dir beide Gehilfen die sortierten Teile zurückgegeben haben. Dann durchlaufe beide Stapel gleichzeitig von oben nach unten und mische die Karten nach einer Art Reißverschlussprinzip zu einem sortierten Gesamtstapel zusammen.
4. Gib diesen an deinen Meister zurück.

Wir demonstrieren das Vorgehen dieses Algorithmus an einem Beispiel:

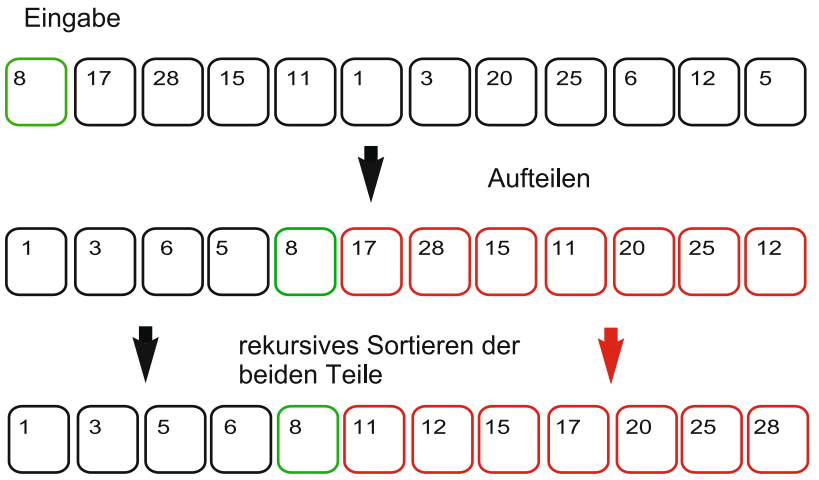


Der zweite Algorithmus löst das gleiche Problem auf eine völlig andere Weise:

**Algorithmus 2**

1. Falls der Stapel nur aus einer Karte besteht, gib ihn sofort zurück, sonst tue Folgendes:
2. Nimm die erste Karte vom Stapel. Durchlaufe die restlichen Karten und teile sie auf in alle mit einem Wert kleiner oder gleich dem der ersten Karte (Stapel 1) und mit Wert größer als dem der ersten Karte (Stapel 2).
3. Gib die beiden so entstandenen Teilstapel, wenn sie überhaupt Karten enthalten, an je einen Gehilfen mit der Bitte, sie *rekursiv*, d. h. ebenfalls genau nach dem hier beschriebenen Verfahren zu sortieren.
4. Warte, bis dir beide Gehilfen die sortierten Teile zurückgegeben haben, dann lege zuunterst den sortierten Stapel 1, darauf die anfangs gezogene Karte, darauf den sortierten Stapel 2 und gib das Ganze als sortiert zurück.

An einem Beispiel demonstriert sieht dies so aus:




## 3.2 Nähere Erläuterungen zu unseren Sortieralgorithmen

Der erste der beiden Algorithmen heißt MergeSort. Er war bereits dem bekannten ungarischen Mathematiker *Johann (Janos, John) von Neumann* (1903–1957)[1] bekannt in einer Zeit, als es Informatik als eigenes Fach noch nicht gab, und wurde bereits in mechanischen Sortiergeräten eingesetzt.

[1] siehe `http://de.wikipedia.org/wiki/John_von_Neumann`

Der zweite Algorithmus heißt QuickSort. Er wurde von dem bekannten britischen Informatiker *C.A.R. Hoare*[2] bereits 1962 entwickelt.

Die Beschreibung im vorigen Abschnitt zeigt, dass man zum Ausführen von Algorithmen natürlich nicht unbedingt einen Computer braucht. Zum besseren Verständnis empfehlen wir, beide Algorithmen „von Hand“ auszuführen, indem man selbst die Rolle der verschiedenen „Gehilfen“ übernimmt.

In allen höheren Programmiersprachen (z. B. C, C++, JAVA) gibt es diese Möglichkeit, dass eine Prozedur „sich selbst“ aufruft, um die gleiche Aufgabe auf die gleiche Art und Weise für ein verkleinertes Teilproblem zu lösen. Man nennt dieses Konzept **Rekursion**, und es hat eine sehr wichtige Funktion in der Informatik. Wendet man z. B. MergeSort auf eine Folge von 16 Zahlen an, so erhalten beide Gehilfen eine Teilfolge der Länge 8 zum Sortieren. Diese rufen wieder ihre je zwei eigenen Gehilfen, um Folgen der Länge 4 zu sortieren usw. Dieser gesamte Ablauf des Algorithmus ist in Abb. 3.1 dargestellt, die man in der Informatik einen *Baum* nennt.

Die Rekursion bricht ab, wenn die Teilprobleme hinreichend klein geworden sind, um direkt gelöst zu werden. In unseren Algorithmen ist dies für Folgen der Länge 1 der Fall, wo nichts mehr getan zu werden braucht, um sie zu sortieren. In den Beschreibungen der Algorithmen sorgt jeweils die Anweisung 1 für diese *Verankerung* der Rekursion.

Die Vorgehensweise, ein größeres Problem dadurch zu lösen, dass man es in kleinere Teilprobleme zerlegt, diese rekursiv löst und die entstehenden Teillösungen zu einer Gesamtlösung zusammenfügt, nennt man in der Informatik *divide and conquer*. Unsere beiden Algorithmen funktionieren nach diesem Prinzip, und es lässt sich auf viele, sehr unterschiedliche Probleme erfolgreich anwenden.

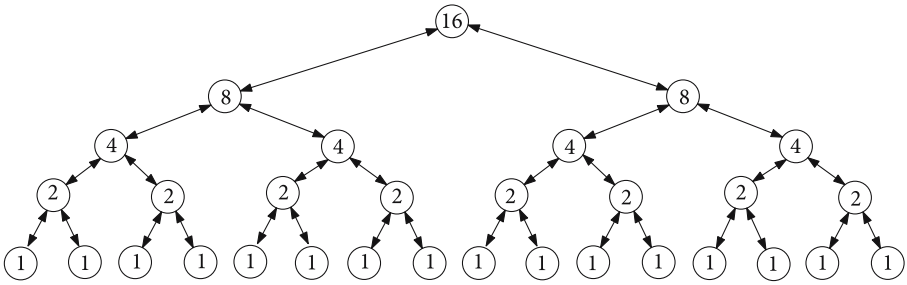



**Abb. 3.1.** Rekursionsbaum für MergeSort

[2] siehe `http://en.wikipedia.org/wiki/C._A._R._Hoare`

## 3.3 Experimenteller Vergleich der Sortieralgorithmen

Natürlich fragt man sich, warum man zum Sortieren, das doch scheinbar ein so einfaches Problem ist, solch merkwürdige Algorithmen nehmen sollte. Dazu haben wir die beiden Algorithmen sowie das *Sortieren durch Einfügen* aus Kap. 2 in JAVA auf einem Computer unseres Instituts *implementiert* (= programmiert) und die Zeit gemessen, die die Algorithmen für verschieden lange Folgen von Zahlen benötigen. Abbildung 3.2 zeigt das Ergebnis. Offensichtlich ist MergeSort wesentlich schneller als *Sortieren durch Einfügen* und QuickSort noch einmal deutlich schneller als MergeSort.

In einer halben Sekunde (500 ms) Rechenzeit kann *Sortieren durch Einfügen* „nur“ Folgen der Länge 8000 sortieren, während MergeSort 20 mal mehr Zahlen in gleicher Zeit schafft. QuickSort ist dann noch einmal etwa 4 mal schneller als *Mergesort*.

## 3.4 Theoretische Bestimmung der Laufzeiten

Wie im Kap. 2 ist es möglich, mit rein mathematischen Methoden herzuleiten, wie die Laufzeit der Algorithmen von der Anzahl $n$ der zu sortierenden Elemente abhängt, ohne dass man dafür den Algorithmus programmieren und auf einem Rechner Zeitmessungen durchführen muss. Dabei stellte sich her-

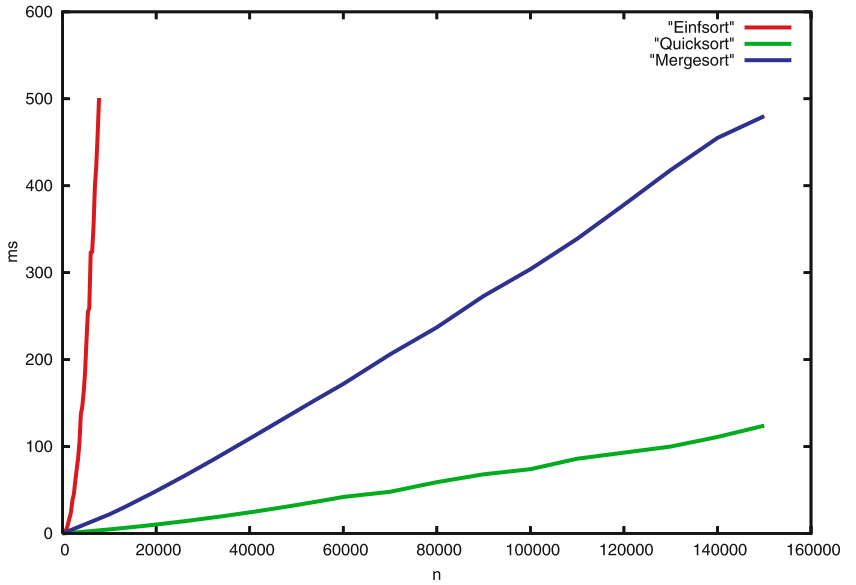



**Abb. 3.2.** Experimentell bestimmte Laufzeiten in Millisekunden der drei Algorithmen zum Sortieren von Folgen der Länge 1 bis 150.000

aus, dass ein einfacher Sortieralgorithmus wie *Sortieren durch Einfügen* eine Laufzeit hat, die proportional zu $n^2$ ist.

Eine ähnliche theoretische Abschätzung der Laufzeit (auch *Laufzeitanalyse* genannt) wollen wir jetzt für MergeSort durchführen.

Dazu überlegen wir uns zunächst, wie viele Vergleiche man für Schritt 3 des Algorithmus, also zum *Mischen* von zwei sortierten Teilfolgen der Länge $n/2$ zu einer sortierten Folge der Länge $n$ benötigt. Beim Mischen werden zunächst die beiden kleinsten Karten beider Teilfolgen miteinander verglichen, dann bildet man mit der kleineren von beiden als kleinste insgesamt den neuen Gesamtstapel und fährt mit den beiden Reststapeln auf die gleiche Art fort. In jedem Schritt werden zwei Karten verglichen und die kleinere davon auf den Gesamtstapel gelegt. Da dieser am Ende aus $n$ Karten besteht, hat man also höchstens $n$ Vergleiche durchgeführt (genau genommen sogar höchstens $n-1$).

Um die rekursive Beschaffenheit des gesamten Algorithmus zu betrachten, schauen wir uns noch einmal den Baum in Abb. 3.1 an. Der Meister an der Spitze hat 16 Karten zu sortieren. Er gibt je 8 an zwei Gehilfen weiter, diese geben je 4 an wieder je zwei Gehilfen weiter usw. Der Meister auf der obersten Ebene muss in Schritt 3 des Algorithmus zwei mal 8 (im allgemeinen zwei mal $n/2$) Karten zu einer Gesamtfolge der Länge 16 ($n$) zusammenmischen, was, wie wir oben gesehen haben, höchstens 16 ($n$) Vergleiche erfordert. Die beiden Gehilfen auf der darunterliegenden Stufe mischen jeweils $n/2$ Karten, benötigen also jeweils $n/2$ Vergleiche, also zusammen auch höchstens $n$. Genauso mischen vier Gehilfen auf der dritten Ebene des Baums jeweils $n/4$ Karten und benötigen alle zusammen wieder höchstens $n$ Vergleiche usw.

Man sieht also, dass pro Ebene des Baums höchstens $n$ Vergleiche notwendig sind. Es bleibt noch, die Anzahl der Ebenen zu berechnen. Für $n = 16$ sind es, wie man am Bild sieht, 4 Ebenen. Wir sehen, dass die Größe der zu sortierenden Teilfolgen, wenn man im Baum absteigt, von $n$ auf der obersten Ebene zu $n/2$ auf der zweiten Ebene und weiter auf $n/4$, $n/8$ usw. absinkt, also von Ebene zu Ebene halbiert wird, bis auf der untersten Ebene 1 erreicht wird. Die Anzahl der Ebenen definiert sich also dadurch, wie oft man $n$ durch 2 dividieren kann bis man bei 1 ankommt. Dies ist bekanntlich (siehe auch Kap. 1) der *Logarithmus* zur Basis 2 von $n$, $\log_2(n)$. Da pro Ebene höchstens $n$ Vergleiche notwendig sind, braucht MergeSort insgesamt höchstens $n \log_2(n)$ Vergleiche, um $n$ Zahlen zu sortieren.

Wir haben bei unserer Analyse einfachheitshalber angenommen, dass sich die Länge der Eingabefolge immer wieder ohne Rest durch 2 teilen lässt, bis man 1 erreicht, also eine Potenz von 2 (d. h. eine der Zahlen 1, 2, 4, 8, 16, ...) ist. Für andere Werte von $n$ lässt sich MergeSort mit etwas mehr Aufwand auch analysieren, die Idee bleibt die gleiche und auch das Ergebnis ist, dass die Anzahl der Vergleiche höchstens $n\lceil \log_2(n) \rceil$ beträgt. $\lceil \log_2(n) \rceil$ ist dabei die nächstgrößere ganze Zahl von $\log_2(n)$.

Wir haben hier nur die Anzahl der Vergleiche abgeschätzt. Multipliziert man diese Zahl mit der Zeit, die der Rechner, auf dem der Algorithmus ausgeführt wird, für einen Vergleich braucht[3], so erhält man die gesamte Zeit, die für Vergleiche benötigt wird. Dies ist noch nicht die gesamte Laufzeit, weil außer Vergleichen auch noch andere Operationen, wie das Umspeichern von den zu sortierenden Elementen, die Organisation der Rekursion usw., benötigt werden. Aber trotzdem ist die Gesamtlaufzeit, wie man sich überlegen kann, zumindest *proportional* zur Zahl der Vergleiche, so dass wir durch unsere Analyse zumindest wissen, dass sie für MergeSort proportional zu $n \log_2(n)$ ist.

Diese Überlegungen erklären die im vorigen Abschnitt experimentell beobachtete Überlegenheit von MergeSort gegenüber *Sortieren durch Einfügen*. Dafür war ja im Kap. 2 $n(n-1)/2$ als Anzahl der Vergleiche hergeleitet worden und diese Funktion wächst in der Tat wesentlich schneller als die Funktion $n \log_2(n)$.

Für QuickSort ist die Situation schwieriger. Man kann zeigen, dass seine Laufzeit für bestimmte Eingaben, z. B. wenn die Eingabefolge schon sortiert ist, auch sehr langsam, d. h. proportional zu $n^2$, ist. Ihr bekommt eine Vorstellung davon, warum dies so ist, wenn ihr ihn einmal auf einer solchen Eingabe „von Hand“ ausführt. Dies ist aber nur der Fall, wenn man als Element $x$ zum Aufspalten der Folge, das sogenannte *Pivotelement*, jeweils das erste Element nimmt. Wählt man stattdessen irgendeines per Zufall aus der Folge, so ist die Wahrscheinlichkeit, dass der Algorithmus langsam ist, sehr gering. Im Mittel ist die Laufzeit auch proportional zu $n \log_2(n)$ mit, wie unsere Experimente zeigen, einer offensichtlich besseren Proportionalitätskonstanten als MergeSort. QuickSort ist in der Praxis tatsächlich der schnellste Sortieralgorithmus, wie ja auch unsere experimentellen Vergleiche im vorigen Abschnitt zeigen.

## 3.5 Implementierung in JAVA

Die Algorithmen sind durch ihre Beschreibung im Abschn. 3.1 bereits vollständig und verständlich dargestellt. Trotzdem werden hier für Kenner der Programmiersprache JAVA, die sich auch für die technischen Einzelheiten interessieren, noch zusätzlich die Implementierungen der Algorithmen in dieser Sprache vorgestellt.

Eigentlich werden bereits beide Algorithmen fertig in JAVA angeboten und können benutzt werden. MergeSort findet sich dabei unter dem Namen „Collections.sort“ und QuickSort unter dem Namen „Arrays.sort“. Diese Methoden kann man nicht nur für Zahlen benutzen, sondern für beliebige Objekte, die man paarweise vergleichen kann.

[3] Für einen Vergleich zweier ganzer Zahlen braucht ein moderner Rechner etwa eine Nanosekunde, d. h. eine milliardstel Sekunde.

Wir zeigen hier aber selbstgeschriebene und leichter zu verstehende, für ganze Zahlen implementierte Methoden, mit denen auch die Messungen im Abschn. 3.3 durchgeführt wurden. Dabei bezieht sich ein Aufruf der Methode immer auf einen Teil eines Arrays $A$, dessen Grenzen angegeben werden.

Zunächst zu MergeSort. Wir zeigen zunächst die Methode zum Mischen zweier sortierter Folgen zu einer sortierten Gesamtfolge:

```
public static void mische (int[] A, int al, int ar,
                           int[] B, int bl, int br,
                           int[] C)
     // mischt ein sortiertes Array-Segment A[al]...A[ar] mit
     // B[bl]..B[br] zu einem sortierten Segment C[0] ...

         { int i = al, j = bl;
           for(int k = 0; k <= ar-al+br-bl+1; k++)
                 { if (i>ar)    // A abgearbeitet
                              {C[k]=B[j++]; continue;}
                   if (j>br)    // B abgearbeitet
                              {C[k]=A[i++]; continue;}

                   C[k] = (A[i]<B[j]) ? A[i++]:B[j++];
                 }}
```

Danach ist MergeSort selbst als Methode in JAVA leicht zu beschreiben:

```
public static void mergeSort (int[] A, int al, int ar)
   { // sortiert das Array-Segment A[al] bis A[ar]

    if(ar>al) {int m = (ar+al)/2;

              // Rekursives Sortieren der Haelften:
                  mergeSort(A,al,m);
                  mergeSort(A,m+1,ar);

              // Mischen ins Array B :
                  int[] B = new int[ar-al+1];
                  mische(A,al,m,  A,m+1,ar,  B);

              // Zurueckspeichern:
                  for(int i=0;i<ar-al+1;i++) A[al+i] = B[i];
              }
     }
```

Das Programm kann noch schneller gemacht werden, indem man sich das Zurückspeichern von Array $B$ nach $A$ spart und abwechselnd das Programm rekursiv auf $A$ und auf $B$ aufruft. Aus Gründen der Einfachheit wurde dies hier nicht getan.

QuickSort hat gegenüber MergeSort den zusätzlichen Vorteil, dass es kein Hilfsarray $B$ braucht, sondern mit dem Array $A$ auskommt, auf dem die Daten stehen. Das Aufspalten (Schritt 2 im Algorithmus) geschieht so, dass man mit einer „Zeigervariablen“ $i$ am Anfang des zu sortierenden Arraysegments losläuft und anhält, sobald ein $A[i]$ gefunden wurde, das größer ist als das Pivotelement, also nicht in die linke Hälfte gehört. Gleichzeitig läuft die Variable $j$ vom rechten Ende des Segments nach links und hält bei Elementen $A[j]$, die kleiner als das Pivotelement sind. Haben beide angehalten, werden $A[i]$ und $A[j]$ vertauscht und der Lauf geht weiter, bis die beiden Zeiger sich treffen.

```
public static void tausche (int[] A, int i, int j)
                        {int t = A[i]; A[i] = A[j]; A[j]=t;}

public static void quickSort (int[] A, int al, int ar)
// sortiert das Segment A[al],...,A[ar]
{if(al<ar)
       {
         int pivot = A[al], // 1. Element als Pivotelement
             i=al, j=ar+1;

       //  Aufspalten:
            while(true)
              {   while (A[++i] < pivot && i<ar){}
                  while (A[--j] > pivot && j>al){}

                  if (i<j) tausche(A,i,j);
                  else
                      break;
              }
              tausche(A,j,al);

              quickSort(A,al,j-1);
              quickSort(A,j+1,ar);
        }
     }
```

## Zum Weiterlesen und Experimentieren

Nach Animationen der hier vorgestellten Algorithmen, d. h. der Veranschaulichung ihrer Arbeitsweise in einer Art Trickfilm, kann man im Internet suchen. Besonders seien folgende Seiten empfohlen:

`http://math.hws.edu/TMCM/java/xSortLab/`
`http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html`
`http://www.cs.princeton.edu/~ah/alg_anim/version1/Animator.html`
`http://www.tcs.ifi.lmu.de/~gruberh/lehre/sorting/sort.html`

Zum Teil werden dort auch andere Sortieralgorithmen sowie der Programmcode in einer höheren Programmiersprache angegeben.

*Sortieren durch Einfügen* ist meist unter dem Namen „insertion sort“ aufgeführt; es hat ein Laufzeitverhalten proportional zu $n^2$, ebenso wie der häufig vorgestellte Algorithmus *Bubblesort*. Schon für kleinere Eingabefolgen mit 100 oder 200 zu sortierenden Objekten merkt man deutlich die Überlegenheit von MergeSort und QuickSort.

# 4

# Paralleles Sortieren – Parallel geht schnell

Rolf Wanka

Friedrich-Alexander-Universität Erlangen-Nürnberg

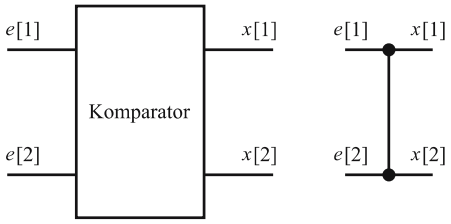
Seit es Hardware gibt, überlegt man sich, spezielle Geräte zu bauen, die ganz besonders schnell die bereits in den Kap. 2 und 3 angesprochene Sortieraufgabe lösen können. Im vorliegenden Kapitel werden wir eine Lösung des Sortierproblems vorstellen, die geeignet ist, auf einem Mikrochip implementiert zu werden, und die ein so genanntes *paralleles* Sortierverfahren ist.


Als Herman Hollerith um 1890 seine berühmte Hollerith-Maschine zur Auswertung der amerikanischen Volkszählung baute, konstruierte er ein zusätzliches Gerät zum Sortieren der Lochkarten, auf denen die erhobenen Datensätze gespeichert waren. Im obigen Foto ist eine solche *Hollerith-Maschine* im Original zu sehen. Das rechte, etwas kleinere Gerät ist dabei die Lochkarten-Sortiereinheit. Das Kabel, das wir sehen können, überträgt natürlich keine Daten, sondern versorgt die Sortiereinheit lediglich mit Strom.

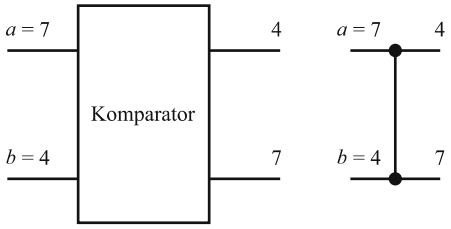
Das Verfahren, mit dem die Lochkarten durch die Maschine sortiert werden, haben wir bereits in Kap. 2 unter dem Namen „Sortieren durch Einfügen“ kennengelernt. Im Zeitalter der Elektronik im Miniaturformat muss eine solche Sortiereinheit, die ja auch keine Lochkarten mehr zu verarbeiten hat, natürlich sehr viel kleiner als Holleriths Gerät ausfallen.

## Sortieren in Hardware: Komparatoren und Sortiernetzwerke

Wir werden im Folgenden eine Konstruktionsvorschrift für einen Hardware-Sortierer angeben. Dieser Sortierer bekommt gleichzeitig auf $n$ Leitungen eine beliebig wirre Folge aus $n$ Zahlen, die er sortieren soll. Dieser Sortierer soll nur aus ganz bestimmten Bausteinen zusammengesetzt werden, die *Komparatoren* genannt werden. Ein solcher Komparator hat zwei Eingänge $e[1]$ und $e[2]$ und zwei Ausgänge $x[1]$ und $x[2]$. Zwei beliebige natürliche Zahlen $a$ und $b$ kommen nun über die Eingangsleitungen in den Komparator hinein. Als Ausgabe wird über die Ausgangsleitung $x[1]$ die kleinere der beiden Zahlen zurückgegeben, also $x[1] = \min\{a, b\}$, und über $x[2]$ die größere der beiden, also $x[2] = \max\{a, b\}$. Die folgende Abbildung zeigt zwei Möglichkeiten, einen solchen Komparator zu zeichnen. Wir werden die rechte, kompaktere Darstellung nutzen. Um das elektronische Innenleben eines Komparators wollen wir uns hier gar nicht kümmern.




Die Eingabe $a = 7$ und $b = 4$ wird also von einem Komparator wie folgt verarbeitet:

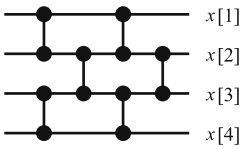



Haben wir nur einen einzigen Komparator zur Verfügung, so können wir ihn einsetzen, um mit ihm in den bereits vorgestellten Verfahren MergeSort und QuickSort (siehe Kap. 3) die dort benutzten bedingten Vertauschungen

durchzuführen. Da wir aber nur einen Komparator haben, muss er nacheinander, d. h. *sequenziell* die notwendigen bedingten Vertauschungen abarbeiten.

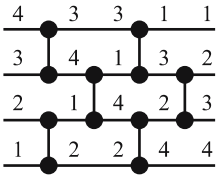
Wir wollen nun mit vielen Exemplaren dieses Bausteins eine Schaltung konstruieren, die Zahlenfolgen der Länge $n$ schneller als die sequenziellen Verfahren sortiert.

Mit einem kleinen Beispiel für eine solche Schaltung aus Komparatoren wollen wir beginnen. Dazu betrachten wir das folgende Bild:




Die Eingabe kommt von links und läuft nach rechts durch die Schaltung hindurch. Statt Schaltung sagt man auch gerne *Netzwerk*. Eine einfache Überlegung zeigt, dass dieses Netzwerk aus 6 Komparatoren jede Zahlenfolge der Länge 4 zu sortieren vermag: Egal, auf welcher Leitung wir links die kleinste Zahl eingeben, sie verlässt immer auf der obersten Leitung rechts das Netzwerk. Für die größte Zahl gilt entsprechend, dass sie unabhängig davon, wo sie das Netzwerk betritt, auf der untersten Leitung das Netzwerk verlässt. Und wie wir auch sehen, wird zum Schluss durch den letzten Komparator gerade dafür gesorgt, dass $x[2] \leq x[3]$ ist. Wir erkennen also, dass dieses Netzwerk jede Eingabefolge sortiert. Darum wird es *Sortiernetzwerk* genannt.

Das nächste Bild zeigt, wie dieses Netzwerk die Eingabefolge $(4, 3, 2, 1)$ verarbeitet, und dass bei dieser Eingabe in jedem Schritt eine Vertauschung durchgeführt wird, also keiner der Komparatoren des Netzwerks überflüssig ist.




Eine weitere interessante Beobachtung können wir der Zeichnung auch noch entnehmen: Alle untereinander stehenden Komparatoren können gleichzeitig ausgeführt werden. Es vergehen also nur 4 Zeiteinheiten, bis die Eingabe sortiert vom Netzwerk ausgegeben wird. Statt von Zeiteinheiten spricht man auch von *parallelen Schritten*.

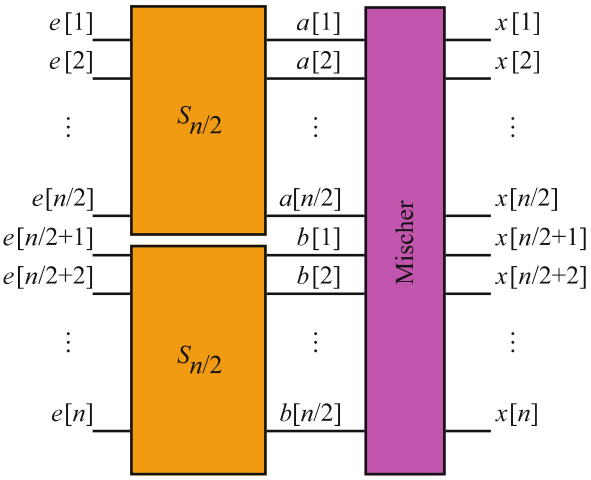
## Das Bitone Sortiernetzwerk: Aufbau

Können wir nun die in Kap. 3 bereits vorgestellten, schnellen sequenziellen (nicht parallelen) Verfahren MergeSort und QuickSort, die ja auch Ver-

gleiche als Grundoperation benutzen, durch Netzwerke aus Komparatoren realisieren? Leider ist das nicht direkt möglich, da man bei ihnen zu Beginn nie weiß, auf welchen Indexpositionen irgendwann später ein Komparator angewandt werden muss. Denn das hängt bei den schnellen sequenziellen Verfahren von der Eingabe ab, oder genauer, davon, wie frühere Vergleiche ausgegangen sind! Bei einem Komparator-Netzwerk darf das nicht der Fall sein, hier haben wir uns schon festgelegt, zwischen welchen Leitungen Komparatoren sind, bevor die Eingabe überhaupt in das Netzwerk hineinkommt.

Deswegen hat Kenneth Batcher, ein Wissenschaftler an der Kent State University in den USA, 1968 ein spezielles Netzwerk konstruiert, das in wenigen, nämlich $\frac{1}{2} \cdot \log_2 n \cdot (\log_2 n + 1)$ parallelen Schritten jede Eingabe sortieren kann. Das heißt, zum Sortieren von $2^{20} = 1.048.576$ Zahlen benötigt dieses Netzwerk nur $\frac{1}{2} \cdot 20 \cdot 21 = 210$ parallele Schritte. Der Ansatz zu diesem Netzwerk folgt dem in Kap. 3 beschriebenen Vorgehen des *divide and conquer*, also des Teilens und Herrschens.

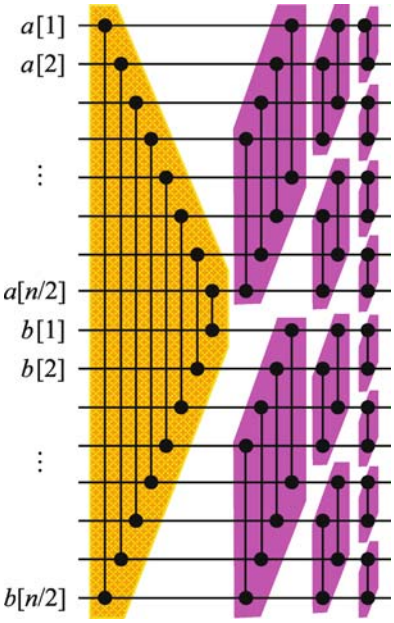
Wir wollen also $n$ Zahlen sortieren. Dazu konstruieren wir das Sortiernetzwerk $S_n$. Weil wir mit der Teile-und-Herrsche-Methode die Zahlenfolgen immer wieder in zwei gleichgroße Teilfolgen zerlegen werden, wählen wir der Einfachheit halber $n = 2^k$, da wir dann immer wieder ohne Rest durch 2 dividieren können. Angenommen, wir wüssten schon, wie wir halb so viele, also $\frac{n}{2} = 2^{k-1}$ Zahlen mit Hilfe eines Netzwerks $S_{\frac{n}{2}}$ sortieren könnten. Dann sortieren wir mit zwei Kopien von $S_{\frac{n}{2}}$ gleichzeitig erst einmal die obere und die untere Hälfte der Eingabe. Dies ist der Teile-Schritt des Teilens und Herrschens.




Das oben stehende Bild zeigt, wie das Netzwerk arbeiten soll. Es stellt sozusagen die *Architektur* des Netzwerks dar. Im Inneren der Kästen sind Komparatornetzwerke, die noch anzugeben sind.

Die beiden „halb so großen“ Kopien des Sortierers $S_{\frac{n}{2}}$ erzeugen die Folgen $a[1], ..., a[\frac{n}{2}]$ bzw. $b[1], ..., b[\frac{n}{2}]$. Und jetzt bleibt als Herrsche-Schritt wie

bei MergeSort in Kap. 3 die Aufgabe des Mischens zu lösen, d. h., nun müssen wir ein Netzwerk aus Komparatoren angeben, das aus den beiden sortierten Zahlenfolgen $a[1], ..., a[\frac{n}{2}]$ und $b[1], ..., b[\frac{n}{2}]$ eine sortierte Gesamtfolge macht. Dazu hat Kenneth Batcher das in der folgenden Abbildung dargestellte Komparator-Netzwerk erfunden, das er den *Bitonen Mischer* nannte. Warum es so heißt, werden wir gleich in der Analyse sehen. Dieses Netzwerk werden wir also im vorherigen Bild für den „Mischer“ einsetzen.




Der Bitone Mischer beginnt immer mit dem gelb unterlegten parallelen Schritt (linkes „Dreieck“; das ist wirklich nur ein einziger paralleler Schritt, die Komparatoren sind nur nebeneinander gezeichnet, um sie besser erkennen zu können) und führt danach die violett unterlegten parallelen Schritte (rechte „Rauten“) aus. Die Architektur dieses Mischers besteht also allgemein aus dem ersten, gelben Dreieck, und dann aus einer Abfolge von violetten Rauten, deren Breite sich bei jedem Schritt halbiert. Den Bitonen Mischer für $n = 32$ sollte die Leserin und der Leser einmal selbst zeichnen.

## Das Bitone Sortiernetzwerk: Korrektheit und Laufzeit

Dass der Bitone Mischer die beiden sortierten Folgen $a = (a[1], ..., a[\frac{n}{2}])$ und $b = (b[1], ..., b[\frac{n}{2}])$ tatsächlich in eine sortierte Gesamtfolge verwandelt, ist bei weitem nicht offensichtlich. Um das zu beweisen, nutzen wir eine tolle Eigenschaft der Komparator-Netzwerke, das so genannte 0-1-*Prinzip*:

> Wenn ein Komparator-Netzwerk jede Folge der Länge $n$, die **nur aus 0-en und 1-en** besteht, sortiert,
> genau dann sortiert es auch **jede beliebige** Zahlenfolge der Länge $n$.

Im Folgenden beschreiben wir die Beweisidee. Da man den Beweis des 0-1-Prinzips nicht für den Korrektheitsbeweis des Bitonen Sortierers benötigt, sondern eben nur das Prinzip, können die Leserin und der Leser beim ersten Lesen den folgenden Abschnitt bis zu (∗∗) überspringen.

Die Beweisidee des 0-1-Prinzips ist ganz einfach. Um sie zu veranschaulichen zeigen wir statt des 0-1-Prinzips das 0-1-2-3-4-Prinzip, bei dem in der obigen Aussage die Formulierung „0-en und 1-en“ durch „0-en, 1-en, 2-en, 3-en und 4-en“ ersetzt wird.

Wir betrachten eine beliebige Folge $a = (a[1], \ldots, a[n])$ aus den Zahlen 1 bis $n$, eine so genannte *Permutation*, die die Eingabe für ein beliebiges, aber fest vorgegebenes Komparatornetzwerk sein soll. Wir picken nun zwei verschiedene Zahlen $i$ und $j$, $i < j$, aus der Folge heraus und konstruieren die Zahlenfolge $b$ mit

$$b[k] = \begin{cases} 0 & \text{falls } a[k] < i \\ 1 & \text{falls } a[k] = i \\ 2 & \text{falls } i < a[k] < j \\ 3 & \text{falls } a[k] = j \\ 4 & \text{falls } j < a[i]. \end{cases}$$

Das heißt alle Zahlen kleiner als $i$ werden zu 0, $i$ wird zu 1, alle Zahlen zwischen $i$ und $j$ werden zu 2, $j$ wird zu 3, und alle Zahlen größer als $j$ werden zu 4. Aus $a = (6, 1, 5, 2, 3, 4, 7)$ mit $i = 3$ und $j = 5$ wird z. B. die Folge $b = (4, 0, 3, 0, 1, 2, 4)$. Nun lassen wir die Folgen $a$ und $b$ durch das Komparatornetzwerk durchlaufen. Wenn wir die Wege, die $i$ und $j$ bei Eingabe $a$ durch das Netzwerk nehmen, rot bzw. blau markieren und dann die Wege betrachten, die 1 und 3 bei Eingabe $b$ durch das Netzwerk nehmen, dann sehen wir, dass die 1 genau auf dem roten Weg läuft und die 3 auf dem blauen. Besteht nämlich das Netzwerk nur aus einem Komparator, gilt das ganz offensichtlich, und ein beliebiges Netzwerk ist nur eine Hintereinanderausführung einzelner Komparatoren! Also verlässt auch die Zahl $i$ der Folge $a$ das Netzwerk an derselben Stelle wie die 1 der Folge $b$, und genauso wird die Zahl $j$ von $a$ an derselben Stelle ausgegeben wie die 3 der Folge $b$.

Würde es eine Zahlenfolge $a$ geben, die das Netzwerk nicht sortiert, würde es zwei Zahlen $i$ und $j$, $i < j$ geben, die in der falschen Reihenfolge ausgegeben würden. Die zugehörige Folge $b$ würde dann auch unsortiert ausgegeben, da ja die 1 an der Stelle des $i$ und die 3 an der Stelle des $j$ ausgegeben würde. Wenn also alle 0-1-2-3-4-Folgen sortiert werden, kann es keine allgemeine Folge geben, die nicht sortiert wird.

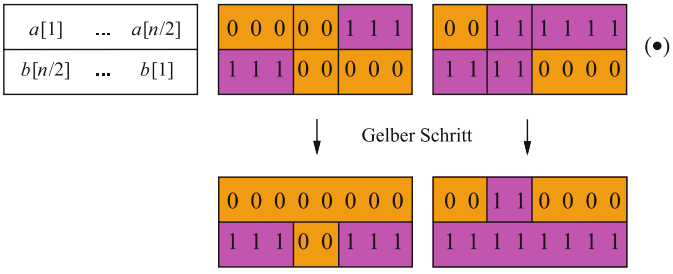
Den Schritt zum 0-1-Prinzip machen wir, indem wir in der Folge $b$ die Zahlen 0, 1 und 2 durch 0 ersetzen und die Zahlen 3 und 4 durch 1 und so die Folge $c$ erhalten. Im oben angeführten Beispiel ergibt das die Folge

$c = (1, 0, 1, 0, 0, 0, 1)$. Durch genaues Hinschauen lässt sich wiederum zeigen, dass an der Stelle, an der die Zahl $i$ von $a$ aus dem Netzwerk herauskommt, bei Eingabe $c$ eine 0 das Netzwerk verlässt, und analog an der Stelle von $j$ eine 1. Ebenso können wir auch fallen lassen, dass $a$ eine Permutation sein muss. Der Rest der Argumentation ist wie gerade: Zu jeder Eingabefolge, die nicht sortiert wird, gibt es eine 0-1-Eingabefolge, die ebenfalls nicht sortiert wird. Wenn also jede 0-1-Folge sortiert wird, wird es sogar jede beliebige Eingabe.[1]

(∗∗) Ab jetzt betrachten wir also erst einmal als Eingaben nur noch 0-1-Folgen. Nun sind wir soweit, dass wir zeigen können, dass der Bitone Mischer aus den sortierten 0-1-Folgen $a$ und $b$ eine sortierte Gesamtfolge macht. Und hier kommt nun der Begriff *biton* ins Spiel. Bitone Folgen entstehen, indem eine monoton steigende 0-1-Folge $x$ (das ist eine Folge, bei der die einzelnen Zahlen von links nach rechts größer werden oder gleich bleiben, aber niemals kleiner werden) und eine monoton fallende 0-1-Folge $y$ (das ist eine Folge, bei der die einzelnen Zahlen von links nach rechts kleiner werden oder gleich bleiben, aber niemals größer werden) in beliebiger Reihenfolge aneinander geklebt werden, d. h. sowohl $xy$ als auch $yx$ sind bitone 0-1-Folgen. Und daher kommt dann auch der Name: zweimal (bi) monoton (ton).

Klingt kompliziert? Ist es aber nicht, wenn wir uns einige Beispiele ansehen: 00111000, 11100011, 0000, 11111000 und 11111111 sind alles bitone Folgen. Sicher finden wir alle die notwendigen Folgen $x$ und $y$, oder? Es gibt jedesmal sogar ganz viele.

Was macht nun der gelbe Schritt, also der erste parallele Schritt des Bitonen Mischers? Drehen wir die Folge $b$ einfach mal um und schreiben sie unter die Folge $a$. Dann entsteht zum Beispiel das folgende Bild.



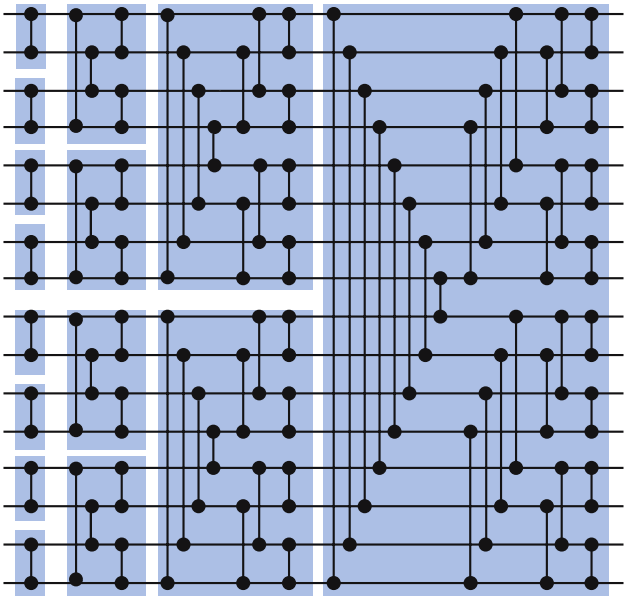

Auf in (•) untereinander stehende Zahlen werden die Komparatoren des gelben Schrittes angewandt, d. h., die kleinere der beiden Zahlen wird nachher oben, die größere unten stehen. Die beiden farbigen Teile zeigen zwei Beispiele. Nach Anwendung des gelben Schrittes ist immer eine ganze Hälfte, und zwar die richtige, voller 0-en bzw. voller 1-en, und die andere Hälfte ist biton! Diese Hälfte ist jetzt die Eingabe der violetten Schritte. Nun schneiden wir eine bitone 0-1-Folge in der Mitte durch und schreiben die Hälften untereinander: Es

[1] Einen kompakten Beweis des 0-1-Prinzips finden wir in Donald Knuths Buch *The Art of Computer Programming, Vol. 3: Sorting and Searching* (siehe auch am Ende des Kapitels unter „Zum Weiterlesen“) auf S. 223.

entsteht das gleiche Bild wie gerade, und wieder werden auf untereinander stehende Zahlen die Komparatoren des nächsten parallelen Schrittes angewandt. Es ist sehr interessant, dies einmal für einige Beispiele selbst durchzuprobieren. Zum Schluss, nach dem letzten violetten Schritt, muss die ganze 0-1-Folge also auch sortiert sein!

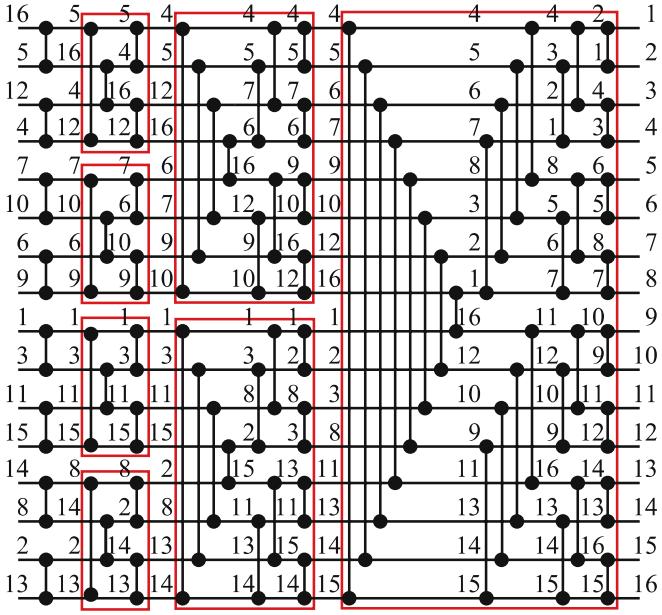
So, und nun können wir uns überlegen, wie $S_{\frac{n}{2}}$ aussieht. Auch $S_{\frac{n}{2}}$ endet mit einem Bitonen Mischer, diesmal nur für $\frac{n}{2}$ viele Eingangsleitungen. Gespeist wird dieser Bitone Mischer von zwei Kopien von $S_{\frac{n}{4}}$. Das wiederholt sich, bis die einspeisenden Sortierer nur noch zwei Zahlen zu sortieren haben, wofür wir dann jeweils einfach einen Komparator nehmen. Insgesamt bekommen wir also z. B. das nachfolgend dargestellte Gesamtnetzwerk $S_{16}$, das jede 0-1-Folge der Länge 16 und damit, wegen des 0-1-Prinzips, auch jede Folge aus 16 Zahlen in 10 parallelen Schritten sortiert! Dabei ist jeder blau unterlegte Kasten ein Bitoner Mischer. Das gesamte Netzwerk heißt dann *Bitoner Sortierer*.

Das nächste Bild zeigt den Bitonen Sortierer $S_{16}$.



Um den Zusammenhang dieser Konstruktion mit unserem Ansatz des Teilens und Herrschens herzustellen, sollten wir einmal dieses Netzwerk mit dem Architekturbild weiter oben vergleichen und die beiden Kopien von $S_8$ finden und markieren.

Im nächsten Bild durchläuft eine Folge aus 16 Zahlen das Netzwerk. Wir betrachten die Ausgaben der rot markierten Kästen, d. h. der Bitonen Mischer und sehen: Sie sind jeweils sortiert!



Nachdem wir damit die Korrektheit des Netzwerks illustriert haben, schließen wir die Analyse des Bitonen Sortierers mit der Untersuchung der Laufzeit und der Anzahl der Komparatoren ab.

Für die Laufzeit $t(n)$ mit $n = 2^k$ bekommen wir allgemein

$$t(n) = 1 + 2 + \cdots + (k-1) + k = \sum_{i=1}^{k} i$$
$$= \frac{1}{2} \cdot k \cdot (k+1) = \frac{1}{2} \cdot \log_2 n \cdot (\log_2 n + 1)$$

parallele Schritte. Das vergleichen wir einmal mit der Laufzeit des sequenziellen MergeSort aus Kap. 3. Dort haben wir gelernt, dass diese $n \log_2 n$ ist. Den dort vorkommenden Faktor von $n$ haben wir hier durch den Faktor $\frac{1}{2} \cdot (\log_2 n + 1)$ ersetzen können! Für das Beispiel mit $n = 2^{20}$ wird also der Faktor 1.048.576 durch den Faktor 11,5 ersetzt, der parallele Sortierer ist also um fast den Faktor 10.000 schneller als MergeSort. Jede Eingabefolge der Länge $2^{20}$ ist bereits nach 210 parallelen Schritten sortiert.

Dieser Laufzeitgewinn wird natürlich durch einen Mehraufwand an Hardware erkauft. Wir bestimmen nun, aus wie vielen Komparatoren der Bitone Sortierer besteht, d. h. wie viele Hardware-Komponenten wir benötigen. Es ist ganz einfach zu sehen, dass in *jedem* parallelen Schritt $\frac{n}{2}$ Komparatoren eingesetzt werden. Wenn wir mit $s(n)$ die Anzahl der Komparatoren bezeichnen, ergibt sich also sofort

$$s(n) = \frac{n}{2} \cdot t(n) = \frac{1}{4} \cdot n \cdot \log_2 n \cdot (\log_2 n + 1)\,.$$

Für $n = 2^{20}$ ist diese Zahl ziemlich groß, nämlich 110.100.480, aber derartig viele Komparatoren auf einem Chip unterzubringen, ist durchaus realistisch. Dabei muss man beachten, dass ein Komparator noch ein „Innenleben“ hat, er besteht nicht nur aus einem einzigen Transistor.

Für kleinere $n$ ist der Bitone Sortierer auf jeden Fall realisierbar und auch schon realisiert worden.

## Abschließende Bemerkungen

In diesem Kapitel haben wir beschrieben, wie man mit Hilfe eines parallelen Sortierers, nämlich mit Batchers Bitonem Sortierer, die Sortierzeit erheblich reduzieren kann um den Preis, dass man mehr Hardware benötigt.

Die Anzahl der parallelen Schritte ist proportional zu $(\log_2 n)^2$. Hier schließt sich die Frage an, ob es vielleicht noch besser geht. Und in der Tat haben Miklós Ajtai, János Komlós und Endre Szeméredi, drei ungarische Wissenschaftler, ein Sortiernetzwerk beschrieben (siehe „Zum Weiterlesen“ 5.), dessen Anzahl an parallelen Schritten proportional zu $\log_2 n$ ist. Es heißt nach den drei Erfindern das *AKS-Netzwerk*. Leider aber ist der Proportionalitätsfaktor, den Mike Paterson bestimmte (siehe „Zum Weiterlesen“ 6.), ungefähr 6.200 und die Architektur nicht so schön regelmäßig wie beim Bitonen Sortierer. Wir ersetzen also bei den Faktoren $\frac{1}{2}\log_2 n$ durch 6.200. Das heißt, dass das AKS-Netzwerk erst dann besser ist, wenn $n \geq 2^{12.400}$ ist, eine wahrhaft astronomische Zahl! Eine weitere Verbesserung stammt von Vašek Chvátal (siehe „Zum Weiterlesen“ 7.).

## Zum Weiterlesen

1. Friedhelm Meyer auf der Heide, Rolf Wanka: *Von der Hollerith-Maschine zum Parallelrechner. Die alltägliche Aufgabe des Sortierens als Fortschrittsmotor für die Informatik*. ForschungsForum Paderborn (FFP) 3 (2000) 112–116.
   `http://www.upb.de/cs/ag-madh/WWW/wanka/pubs/abstracts/FFP00ABS.html`

   Dieser Aufsatz zeigt, dass bei vielen Meilensteinen der Entwicklung der Informatik das Sortierproblem eine maßgebliche Rolle gespielt hat, ob es sich nun um die Hollerith-Maschine handelte, das erste jemals geschriebene Computer-Programm oder den ersten zufallsgesteuerten (*randomisierten*) Algorithmus.

2. Kapitel 3 (Schnelle Sortieralgorithmen)

   Das Kap. 3 über schnelle Sortieralgorithmen stellt den auch vom Bitonen Sortierer verwendeten Ansatz des Sortierens durch Mischen vor.

3. Kenneth E. Batcher: *Sorting networks and their applications*. In AFIPS Conf. Proc. 32, 307–314, 1968.
   `http://www.cs.kent.edu/~batcher/`
   `http://www.cs.kent.edu/~batcher/sort.ps`

Kenneth Batchers Arbeit, die den Bitonen Sortierer und seine Korrektheit vorstellt. Da 1968 das 0-1-Prinzip noch nicht entdeckt war, ist Batchers Korrektheitsbeweis etwas komplizierter.

4. Donald E. Knuth: *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, 2. Auflage 1998.

   Dieses Buch ist *der* Klassiker auf dem Gebiet des Sortierens im Allgemeinen und der Netzwerke für das parallele Sortieren im Besonderen. Der Abschn. 5.3.4 ist den Komparator-Netzwerken gewidmet.

5. Miklós Ajtai, János Komlós, Endre Szeméredi: *Sorting in $c \cdot \log n$ parallel steps*. Combinatorica, 3:1–19, 1983.

   Dieser Aufsatz beschreibt das bislang „schnellste“ parallele Sortiernetzwerk, das wegen der Namen der drei Autoren unter der Bezeichnung AKS-Netzwerk berühmt geworden ist. Leider ist die Konstante $c$ im Titel des Aufsatzes viel zu groß, als dass das Netzwerk in der Praxis eingesetzt werden kann. Paterson (siehe 6.) hat eine Konstruktion angegeben, die den Faktor auf 6.200 reduziert.

6. Mike S. Paterson: *Improved sorting networks with $O(\log n)$ depth*. Algorithmica, 5:75–92, 1990.

   Hier wird die Konstruktion des AKS-Netzwerks (siehe 5.) erheblich vereinfacht und sehr gut lesbar dargestellt. Diese Variante des AKS-Netzwerks benötigt „nur“ noch ca. $6.200 \cdot \log_2 n$ Schritte. Chvátal (siehe 7.) senkt die Konstante sogar auf 1.830.

7. Vašek Chvátal: *Lecture Notes on the New AKS Sorting Network*. Technischer Bericht DCS-TR-294, Computer Science Department, Rutgers University, Oktober 1992.

   Eine weitere Variante des AKS-Netzwerks (siehe 5. und 6.), die „nur“ noch höchstens $1.830 \cdot \log_2 n - 58.657$ parallele Schritte benötigt. Sie schlägt den Bitonen Sortierer ab $n \geq 2^{3.627}$.

# 5

# Topologisches Sortieren – Mit welcher Aufgabe meiner ToDo-Liste fange ich an?

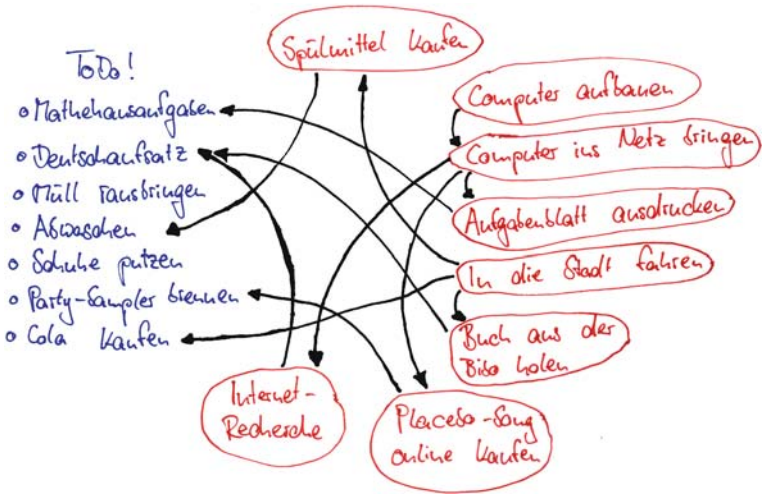
Hagen Höpfner

International University in Germany, Bruchsal

Wie soll ich das nur alles schaffen? Es sind zwar Ferien, aber ich muss noch die Mathehausaufgaben machen. Dann ist da noch der Deutschaufsatz zu schreiben, wofür ich aber auch noch das Buch über die Geschichte der Informatik aus der Bibliothek holen und ein wenig im Internet recherchieren muss. Dabei fällt mir ein, dass auch der Computer nach der letzten LAN-Party noch nicht wieder aufgebaut und angeschlossen ist. Ganz nebenbei bemerkt dürften die Aufgabenblätter für die Mathehausaufgaben auch noch nicht ausgedruckt sein und in meiner E-Mail-Box auf dem GMX-E-Mail-Server schlummern. Dann ist heute abend auch noch eine Party, für die ich noch einen Sampler zusammenstellen und brennen wollte. Natürlich muss auf diese CD auch der neue Song von Placebo, den ich mir noch bei iTunes kaufen wollte. Als ob das noch nicht genug wäre hat mich meine Mutter zum Müllraustragen, Schuheputzen und Abwaschen eingeteilt und das, obwohl ich für die Party ja auch noch einen Kasten Cola aus dem Supermarkt aus der Stadt holen muss. Glücklicherweise liegt der Supermarkt auf dem Weg zur Bibliothek und das Spülmittel zum Abwaschen ist ja ohnehin alle. Aber was erledige ich nun zuerst?


Hm, es ist klar, dass ich die Punkte auf meiner ToDo-Liste nicht in der Reihenfolge abarbeiten kann, in der sie da stehen. Schließlich kann ich die CD

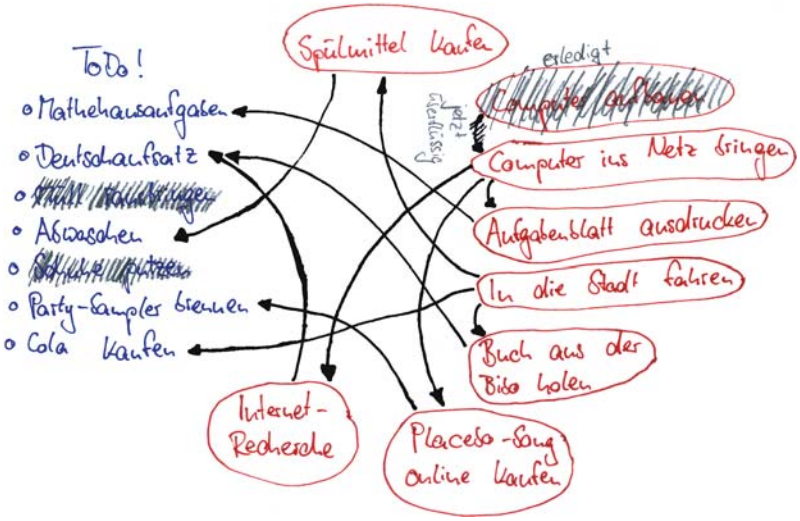
erst brennen, wenn ich alle Songs habe und die bekomme ich nur zusammen, wenn mein Rechner aufgebaut und ans Internet angeschlossen ist. Irgendwie bestehen also Abhängigkeiten zwischen den einzelnen Aufgaben, und noch nicht alle Teilaufgaben sind bisher aufgelistet. Darum nehme ich mir also einen Stift und vervollständige zuerst einmal meine ToDo-Liste. Bevor ich abwaschen kann, muss ich Spülmittel kaufen. Deshalb zeichne ich einen Pfeil von „Spülmittel kaufen“ zu „Abwaschen“. Das Spülmittel bekomme ich nur in der Stadt, weshalb ich einen Pfeil von „In die Stadt fahren“ zu „Spülmittel kaufen“ zeichne usw.




Mann, da wird einem erst mal richtig bewusst, was noch alles zu tun ist, aber womit fange ich denn nun an? Ein Pfeil zeigt an, dass irgendetwas nur dann erledigt werden kann, wenn zuvor etwas anderes erledigt wurde. Also kann ich nur mit etwas anfangen, auf das kein Pfeil zeigt. Zur Auswahl stehen in meinem Fall demnach:

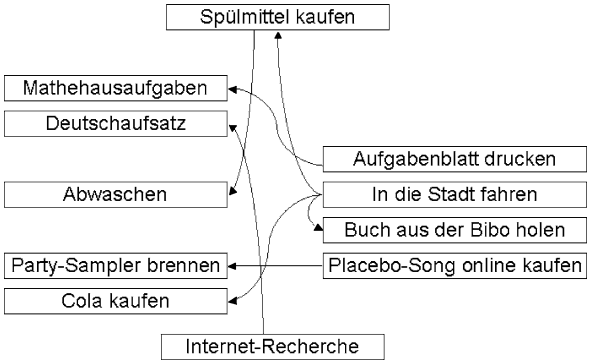
- Müll rausbringen
- Schuhe putzen
- Computer aufbauen
- In die Stadt fahren

Eigentlich ist es ja egal, welche dieser vier Aufgaben ich als erste in Angriff nehme. Aber ich bin mal nett und bringe den Müll raus, putze dann die Schuhe und baue erst anschließend den Computer auf. Nachdem ich das alles erledigt habe, kann ich meine Aufgabenübersicht korrigieren und die erledigten Aufgaben abstreichen. Dadurch fallen natürlich auch etwaige Pfeile, die von erledigten Aufgaben ausgehen (hier von „Computer aufbauen“ nach „Computer ins Netz bringen“), weg.




Natürlich wäre es übersichtlicher, wenn ich einen Bleistift verwendet hätte, dann könnte ich die gestrichenen Pfeile und Aufgaben einfach ausradieren. Nun, der Computer steht und ich kann meine Aufgabenliste ja auch elektronisch abarbeiten. Ich zeichne mir das Ganze also mit einem Graphikprogramm ab, wobei mir einfällt, dass Informatiker, wie mein Bruder, die Struktur meiner Aufgabenliste als Graph bezeichnen. Die zu erledigenden Aufgaben werden durch Knoten in einem Graphen dargestellt und die Abhängigkeiten werden zu gerichteten Kanten zwischen Knoten. Gerichtet bedeutet hier, dass die Lesereihenfolge (in Pfeilrichtung) festgelegt ist. Ist es in dem Graphen möglich, ausgehend von einem Knoten durch Nachzeichnen (ohne Absetzen des Stiftes) der Kanten in Pfeilrichtung wieder beim Ausgangsknoten anzukommen, so wird dieser Graph zyklisch genannt – es gibt dann einen Kreis (auch Zyklus genannt) in dem Graphen.

Doch was könnte ich als Nächstes tun? Nun, an der Tatsache, dass ich jetzt in die Stadt fahren könnte, hat sich nichts geändert. Da ich aber den Computer inzwischen aufgebaut habe, kann ich ihn jetzt auch ins Netz bringen. Schließlich habe ich den einzigen Abhängigkeitspfeil, der auf die Aufgabe „Computer ins Netz bringen“ gezeigt hat, soeben „gelöscht“. Alle anderen Aufgaben können noch nicht erledigt werden. Da ich aber gerade am Rechner sitze, bringe ich ihn auch gleich ins Netz. Dadurch ändert sich mein ToDo-Graph erneut.

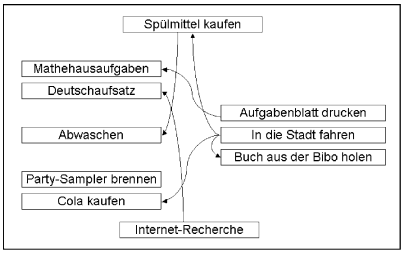


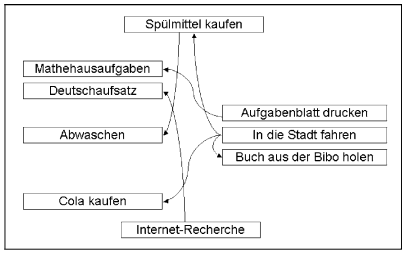

Nun könnte ich immer noch in die Stadt fahren, die Internet-Recherche für den Deutschaufsatz machen, den Placebo-Song kaufen oder aber meine Mathehausaufgaben ausdrucken ...

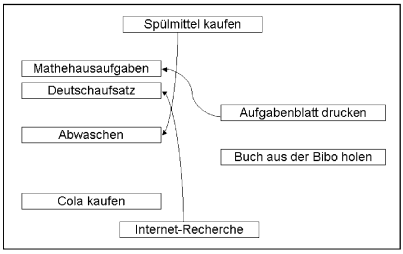
So, gleich geht es zur Party und ich wollte nur ganz kurz zusammenfassen, in welcher Reihenfolge ich meine Aufgaben heute erledigt habe:

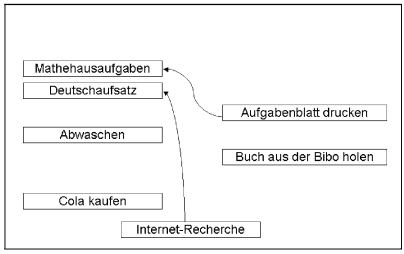
1. Müll rausgebracht
2. Schuhe geputzt
3. Computer aufgebaut
4. Computer ins Netz gebracht
5. Placebo-Song gekauft
6. Party-Sampler gebrannt
7. In die Stadt gefahren
8. Spülmittel gekauft
9. Cola gekauft
10. Buch aus der Bibo geholt
11. Abgewaschen
12. Internet-Recherche erledigt
13. Deutschaufsatz geschrieben
14. Aufgabenblatt gedruckt
15. Mathehausaufgaben gemacht

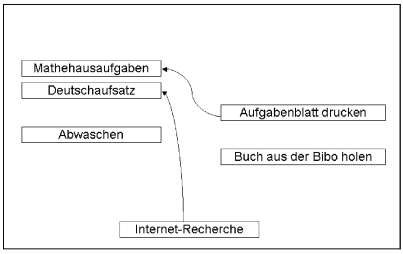
Nach jeder erledigten Aufgabe habe ich den entsprechenden Eintrag in meinem ToDo-Graph nebst der von dem Eintrag ausgehenden Pfeile entfernt, was dazu führte, dass alle Knoten des Graphen nach und nach entfernt wurden. Ich habe die Einzelschritte gespeichert (von links nach rechts und von oben nach unten zu lesen).

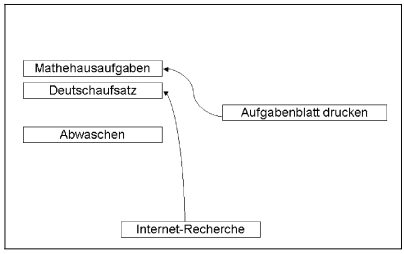

Spülmittel kaufen
Mathehausaufgaben
Deutschaufsatz
Aufgabenblatt drucken
Abwaschen
In die Stadt fahren
Buch aus der Bibo holen
Party-Sampler brennen
Cola kaufen
Internet-Recherche

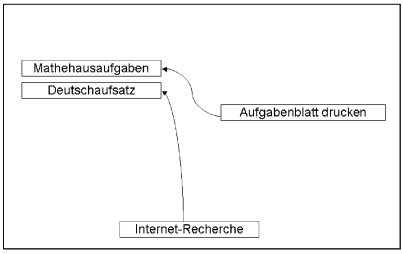

Spülmittel kaufen
Mathehausaufgaben
Deutschaufsatz
Aufgabenblatt drucken
Abwaschen
In die Stadt fahren
Buch aus der Bibo holen
Cola kaufen
Internet-Recherche

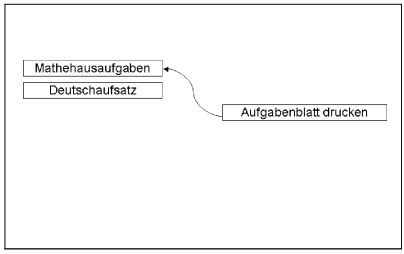

Spülmittel kaufen
Mathehausaufgaben
Deutschaufsatz
Aufgabenblatt drucken
Abwaschen
Buch aus der Bibo holen
Cola kaufen
Internet-Recherche

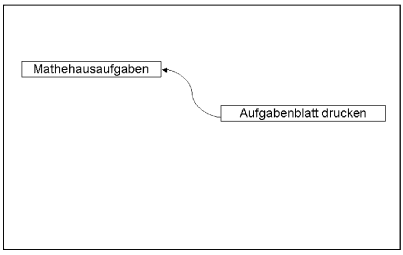

Mathehausaufgaben
Deutschaufsatz
Aufgabenblatt drucken
Abwaschen
Buch aus der Bibo holen
Cola kaufen
Internet-Recherche

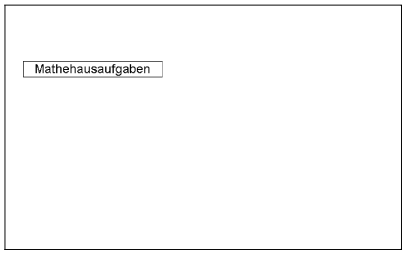

Mathehausaufgaben
Deutschaufsatz
Aufgabenblatt drucken
Abwaschen
Buch aus der Bibo holen
Internet-Recherche


Mathehausaufgaben
Deutschaufsatz
Aufgabenblatt drucken
Abwaschen
Internet-Recherche


Mathehausaufgaben
Deutschaufsatz
Aufgabenblatt drucken
Internet-Recherche


Mathehausaufgaben
Deutschaufsatz
Aufgabenblatt drucken


Mathehausaufgaben
Aufgabenblatt drucken


Mathehausaufgaben

Mein großer Bruder, der Informatik studiert, hat mich gerade darüber aufgeklärt, dass mein Vorgehen „topologisches Sortieren“ genannt wird. Er hat mir auch die folgende Algorithmenbeschreibung geschenkt:

Der Algorithmus TopSort gibt die Knoten eines gerichteten Graphen in topologischer Reihenfolge aus. Der Graph $G = (V, E)$ besteht hierbei aus der Menge der enthaltenen Knoten $V$ und der Kantenmenge $E$ der Form (*Knoten*1,*Knoten*2), wobei die Abhängigkeit von *Knoten*1 nach *Knoten*2 geht und beide Knoten auch in $V$ vorhanden sein müssen.

```
1  Funktion TopSort
2     while V ist nicht leer do
3        Zyklus:=true
4        for each v in V do
5           if es gibt keine Kante e in E der Form (X, v) then
              //  X ist hierbei ein beliebiger anderer Knoten
6              lösche v aus V
7              lösche alle Kanten der Form (v, X) aus E
8              Zyklus:=false
9              print v      // Ausgabe des Knotens
10          endif
11       endfor
12       if Zyklus=true then
13          print Zyklische Abhängigkeit kann nicht aufgelöst werden!
14          break      // Abbrechen der while-Schleife
15       endif
16    endwhile
17 end
```

Der Algorithmus erkennt auch, ob ein Graph zyklische Abhängigkeiten enthält. Derartige zyklische Graphen lassen sich nicht topologisch sortieren, daher muss bei jedem Schritt geprüft werden, ob auch tatsächlich ein Knoten entfernt wurde. Ist dies einmal nicht der Fall, so bricht der Algorithmus automatisch ab.

Das oben verwendete Beispiel verdeutlicht auch ein wesentliches Problem von Computern. Sie arbeiten normalerweise „stupide“ ihre Arbeitsschritte ab. Ziel von TopSort ist es, *eine* mögliche topologische Sortierung zu finden. Eine so bestimmte gültige topologische Sortierung wäre z. B. auch:

- ...
- In die Stadt gefahren
- Spülmittel gekauft
- Abgewaschen
- ...
- Cola gekauft
- ...

Hätten wir nicht nach dem „In die Stadt fahren“ alle Einkäufe erledigt, dann hätten wir das Problem, dass wir später noch einmal in die Stadt fahren müssen, aber diese Information wurde dann ja schon aus dem Graphen entfernt. Ein klein wenig Organisationstalent gehört also neben dem algorithmischen Abarbeiten einer Aufgabenliste auch noch dazu, wenn man seinen Alltag planen will.

## Weitere Anwendungen

Topologisches Sortieren findet eine Anordnung, welche die durch die Kanten angegebenen (zeitlichen) Abhängigkeiten berücksichtigt. Dabei kommt es nicht darauf an, welchen Sachverhalt so ein Graph und die Knoten repräsentieren, da der Algorithmus nicht auf den Inhalt der einzelnen Knoten achtet, sondern lediglich ein- bzw. ausgehende Kanten der Reihe nach entfernt. Daher ist dieser Algorithmus in der Informatik in zahlreichen Teilgebieten verwurzelt. Beispielsweise können mit seiner Hilfe Deadlocks, also Verklemmungen, entdeckt werden. Diese können wie folgt entstehen: Soll eine Ressource (z.B. eine Datei) im Computer exklusiv von einem Programm genutzt werden, so wird diese für alle anderen Programme gesperrt. Diese müssen dann warten, bis die Ressource wieder freigegeben wurde. Deadlocks treten dann ein, wenn ein Programm, welches auf die Ressourcenfreigabe wartet, eine andere Ressource sperrt, die wiederum durch das erste Programm anschließend angefordert wird. Programm 2 wartet dann auf Programm 1 und Programm 1 auf Programm 2. Im Wartegraphen würde diese Beziehung als Zyklus dargestellt werden. Derartige Verklemmungen werden durch TopSort erkannt und eines der blockierten Programme kann abgebrochen werden.

## Zum Weiterlesen

1. G. Saake und K. Sattler: *Algorithmen und Datenstrukturen – Eine Einführung mit Java*. dPunkt-Verlag, Heidelberg, 3. Auflage, 2006.

   Dieses Buch gibt eine generelle Einführung in verschiedenste Algorithmen und wie sich diese mit der Programmiersprache Java umsetzen lassen. Topologisches Sortieren, aber mit einem anderen Verfahren (auf DFS beruhend) wird auf den Seiten 439–441 behandelt.

2. Aus der Wikipedia:
   `http://de.wikipedia.org/wiki/Topologisches_Sortieren`

# 6

# Texte durchsuchen – aber schnell! Der Boyer-Moore-Horspool Algorithmus

Markus E. Nebel

TU Kaiserslautern

Viele Objekte, die ein Computer verarbeitet, speichert er intern in Form eines Textes. Das naheliegendste Beispiel ist ein Text selbst, erstellt mit einem Editor oder Textverarbeitungsprogramm, aber auch Dokumente im Internet werden auf einem Webserver als HTML-Dokument, also als Text mit integrierten Formatierungsbefehlen und Verweisen auf Bilddateien etc., vorgehalten. In diesem Kapitel wollen wir uns entsprechend mit der Suche nach Wörtern in Texten befassen. Warum? Ganz einfach – wir stellen uns beispielsweise vor, wir hätten in Google nach etwas gesucht und seien so auf eine Webseite mit sehr viel Text gestoßen. Es stellt sich sofort die Frage, wo überall im Text unser Suchwort auftritt. Wir wollen die entsprechenden Textstellen natürlich nicht selbst aufspüren, sondern erwarten von unserem Browser, dass er die Vorkommen geeignet hervorhebt. Damit dies aber möglich ist, benötigt der Browser eine Routine, die alle Vorkommen eines Wortes in einem Text möglichst schnell findet. Es dürfte klar sein, dass diese oder ähnliche Anforderungen oft gestellt werden, weshalb wir uns in diesem Abschnitt mit dem so genannten *String Matching Problem* befassen, also mit der Suche nach allen Vorkommen eines Wortes $w$ in einem Text $t$.

## Das naive Vorgehen

Ein Computer speichert Texte zeichenweise ab. Entsprechend ist ein Computer nicht in der Lage, das Wort $w$ mit einem Teil des Textes in nur einem Schritt zu vergleichen, um festzustellen, ob $w$ an der entsprechenden Position im Text vorkommt – auch dieser Vergleich muss Zeichen für Zeichen durchgeführt werden. Wir nehmen also an, dass der Text $t$ aus $n$ Zeichen und das Wort $w$ aus $m$ Zeichen besteht. Dann wollen wir für eine natürliche Zahl $i$ zwischen 1 und $n$ mit $t[i]$ das $i$-te Zeichen des Textes bezeichnen; $t[1]$ ist also das erste Zeichen des Textes, $t[2]$ sein zweites Zeichen usw., $t[n]$ repräsentiert entsprechend das letzte Zeichen des Textes. Analog verwenden wir die Notation $w[j]$, um das $j$-te Zeichen des gesuchten Wortes zu bezeichnen. Hier muss $j$ nun eine Zahl

zwischen 1 und $m$ sein. Suchen wir beispielsweise im Text Such die Nadel im Heu. nach dem Wort Nadel, so sehen $t$ und $w$ im Detail wie folgt aus (die Spalte mit der Zahl $i$ enthält dabei das Zeichen $t[i]$ bzw. $w[i]$):

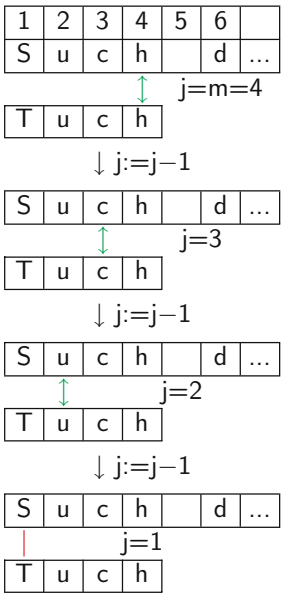
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | u | c | h | | d | i | e | | N | a | d | e | l | | i | m | | H | e | u | . |

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| N | a | d | e | l |

Es ist also $n = 22$ und $m = 5$, und beispielsweise $t[1] = \mathsf{S}$, $t[2] = \mathsf{u}$, $w[4] = \mathsf{e}$ usw. Es ist zu beachten, dass auch die Leerzeichen unseres Textes als Zeichen gelten und nicht ignoriert werden dürfen. Wir werden nachfolgend fast ausschließlich diesen Text als Beispiel betrachten, jedoch nach unterschiedlichen Wörtern $w$ suchen. Um festzustellen, ob dieser Beispieltext mit $w =$ Nadel beginnt, muss ein Algorithmus den Anfang des Textes $t$ und das Wort $w$ zeichenweise vergleichen. Stimmen alle Zeichen überein, so werden wir fündig und das erste Vorkommen von $w$ liegt an erster Position des Textes. Für unser Beispiel ist dies offensichtlich nicht der Fall. Ein Programm muss für diese Feststellung $w[1]$ mit $t[1]$ vergleichen und stellt eine Nichtübereinstimmung fest; $t[1] = \mathsf{S} \neq w[1] = \mathsf{N}$. Damit kann unser Programm folgern, dass $w$ nicht an erster Position in $t$ vorkommt. Nur wenn alle $m$ Vergleiche der Zeichen von $w$ mit den entsprechenden Zeichen von $t$ eine Übereinstimmung ergeben, kommt $w$ an entsprechender Stelle in $t$ vor. In unserem Beispiel genügte ein Vergleich, um festzustellen, dass dies nicht der Fall ist, doch das ist natürlich nicht immer so. Suchen wir beispielsweise in unserem Text nach dem Wort Suche, so führen die vier ersten Vergleiche zu einer Übereinstimmung und erst an fünfter Position, beim Vergleich von $t[5]$ mit $w[5]$, würde ein Unterschied zwischen der betrachteten Textstelle und dem Wort entdeckt – das Leerzeichen ist ungleich dem Buchstaben e. Folgendes kleines Programm führt die eben beschriebenen Vergleiche nacheinander durch. Im Unterschied zu unserer bisherigen Diskussion beginnen wir dabei jedoch mit dem letzten Zeichen von $w$ anstatt mit dem ersten. Der Grund hierfür wird uns später klar werden.

Zeichenweiser Vergleich von Wort $w$ und Text $t$ an erster Position.

```
1  j := m;
2  while (j > 0) and (w[j] = t[j]) do
3      j := j − 1;
4  if (j = 0) then print("Vorkommen an Position 1");
```




Die nebenstehende Grafik verdeutlicht das Vorgehen dieses kleinen Programmes am Beispiel des obigen Textes $t$ und $w = \mathsf{Tuch}$. Ein grüner Doppelpfeil zwischen zwei Zeichen steht dabei für einen Vergleich zweier identischer Zeichen, ein roter Strich verbindet zwei Zeichen, für die eine Nichtübereinstimmung entdeckt wurde. Der Anfang des Textes wird beginnend mit $w[4]$ Zeichen für Zeichen mit dem Wort $w$ verglichen, bis entweder $j$ zu 0 wird (dies ist in unserem Beispiel nicht der Fall) oder die verglichenen Zeichen nicht übereinstimmen (geschieht in unserem Beispiel für $j = 1$). Diese beiden Bedingungen werden in der while-Schleife abgefragt.

In Worten ausgedrückt besagt

$$\texttt{while } (j > 0) \texttt{ and } (w[j] = t[j]) \texttt{ do } j := j - 1;$$

nämlich, dass $j$ solange um 1 verkleinert werden soll, wie es größer als 0 ist und das $j$-te Zeichen des Textes gleich dem $j$-ten Zeichen des Wortes ist. Stimmen die ersten $m$ Zeichen des Textes also nicht mit $w$ überein, so ist die zweite Bedingung irgendwann verletzt und $j$ ist dabei noch größer als 0. Kommt das Programm dann zur Abfrage `if` $(j = 0)$ `...` wird folgerichtig kein Vorkommen gemeldet (die Ausgabe des Textes „Vorkommen an Position 1“ dient uns hier als Platzhalter für eine beliebige Aktion, die wir für ein gefundenes Vorkommen ausführen wollen). Stimmen umgekehrt alle $m$ Zeichen von $w$ mit den ersten $m$ Zeichen von $t$ überein, so wird die while-Schleife beendet, da $j = 0$ gilt. In diesem Fall meldet unser Programm Erfolg.

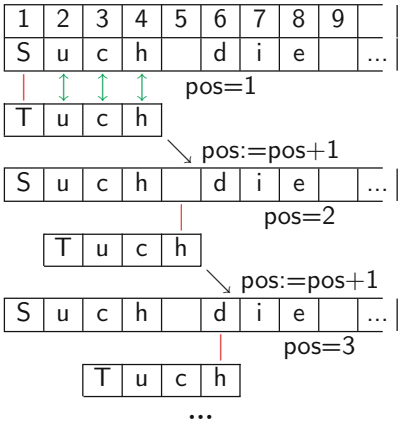
Da die uns gestellte Aufgabe darin besteht, alle Vorkommen von $w$ als Teil von $t$ zu finden, können wir selbstverständlich nicht nur am Anfang von $t$ nach $w$ suchen. Vielmehr kann $w$ ja an jeder beliebigen Stelle von $t$ beginnen, was unser Programm überprüfen muss. Jede beliebige Stelle von $t$ heißt dabei, dass wir ein Vorkommen von $w$ an zweiter, dritter usw. Position von $t$ vermuten müssen. Für die zweite Position müssen wir dann feststellen, ob $w[1] = t[2]$ und $w[2] = t[3]$ und ... und $w[m] = t[m+1]$ gilt. Entsprechend sind die dritte, vierte usw. Position zu betrachten. Die letzte mögliche Position ist dabei die $(n - m + 1)$-ste, da dort $w[m] = t[n]$ in Übereinstimmung gebracht wird. Betrachten wir also die Position *pos* (*pos* sei eine Variable), so ist $w[1]$ mit $t[pos]$,

$w[2]$ mit $t[pos+1]$, ..., $w[m]$ mit $t[pos+m-1]$ zu vergleichen (unser Algorithmus wird dies wieder in umgekehrter Reihenfolge erledigen). Indem wir in obigem Programm eine zusätzliche Variable *pos* verwenden, können wir es so erweitern, dass es entsprechend unserer Überlegungen $w$ an allen möglichen Positionen des Textes sucht (die von oben übernommenen Programmteile sind blau hervorgehoben).

Naiver String-Matching Algorithmus

```
1   procedure Naive
2   pos := 1;
3   while pos ≤ n − m + 1 do      // suche an allen Positionen
4       j := m;
5       while (j > 0) and (w[j] = t[pos + j − 1]) do
6           j := j − 1;
7       if (j = 0) then print("Vorkommen an Position", pos);
8       pos := pos + 1;
9   wend;
10  end.
```

Die äußere while-Schleife stellt dabei sicher, dass wir tatsächlich alle möglichen Positionen für $w$ als Teil innerhalb von $t$ betrachten. Die nachfolgende Abbildung verdeutlicht die dafür vorgenommenen Neuerungen an unserem Programm:




Für $pos = 1$ werden vier Vergleiche durchgeführt, drei Übereinstimmungen, eine Nichtübereinstimmung. Diese Vergleiche werden auch hier durch die schrittweise Verkleinerung von $j$ realisiert. Durch die anschließende Erhöhung von *pos* um eins wird das Wort anschaulich um eine Position nach rechts verschoben. Der dort durchgeführte erste Vergleich ist erfolglos, so dass *pos* erneut erhöht wird ($w$ anschaulich um eine Position nach rechts verschoben wird) usw. An dieser Stelle können wir einen ersten Hinweis dazu wagen, warum wir das Wort von rechts nach links mit dem Text vergleichen: Wie

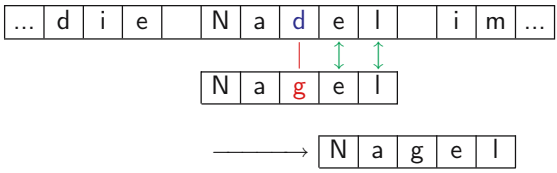
wir nachfolgend sehen werden, ist es nicht notwendig, stets alle Positionen (alle möglichen Werte der Variable *pos*) zu betrachten. Manche können übersprungen werden, ohne ein Vorkommen von $w$ zu verpassen. Ein Vergleich von rechts nach links ermöglicht dabei große Sprünge, ohne dafür komplizierte Berechnungen durchführen zu müssen.

Mit diesem Algorithmus haben wir eine erste Lösung für das String-Matching-Problem gefunden. Unser Programm wird stets alle Vorkommen von $w$ innerhalb $t$ finden (wo sonst sollte denn auch ein Vorkommen versteckt sein, wenn wir alle möglichen Positionen betrachten?). Zu kritisieren bleibt lediglich der hohe Aufwand, den wir dafür betreiben. Im schlimmsten Fall vergleichen wir nämlich ungefähr (Anzahl der Zeichen des Textes) × (Anzahl der Zeichen des Wortes) oft ein Zeichen des Textes mit einem Zeichen des Wortes. Ein Beispiel für diese Situation ist der Text $t =$ aaaaaaaaaaaaa und das Wort $w =$ baaa.

Nun könnte es sein, dass es gar nicht möglich ist, alle Vorkommen von $w$ in $t$ mit weniger Vergleichen zu finden. In diesem Fall bliebe uns nichts anderes übrig, als diesen hohen Aufwand in Kauf zu nehmen. Dies ist aber nicht der Fall, wie wir in folgendem Abschnitt sehen werden.

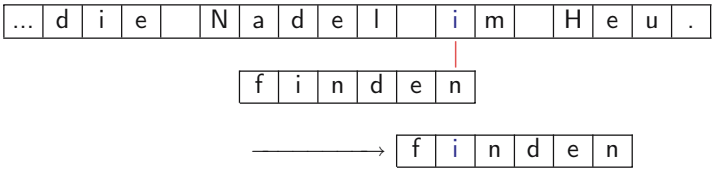
## Der Boyer-Moore-Horspool Algorithmus

In diesem Abschnitt wollen wir sehen, wie wir unseren naiven Algorithmus zur Lösung des „String Matching Problems“ durch wenige Änderungen deutlich beschleunigen können. Die Idee, die wir dabei aufgreifen, wird durch nachfolgendes Beispiel verdeutlicht:

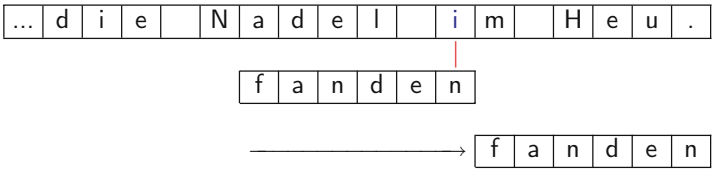



Wenn wir in dieser Situation $w$ mit $t$ zeichenweise vergleichen, stellen wir fest, dass ein d im Text nicht mit dem g im Wort übereinstimmt. Da das Zeichen d in $w$ aber gar nicht vorkommt, kann auch eine oder zwei Positionen weiter rechts kein Vorkommen von $w$ liegen, da jedesmal das d des Textes mit einem Zeichen ungleich d aus $w$ in Übereinstimmung gebracht werden müsste. Wir dürfen $w$ also um 3 Positionen nach rechts verschieben (in unserem Programm $pos := pos + 3$ setzen), ohne ein Vorkommen von $w$ zu verpassen. Dies zeigt, dass unser Algorithmus bisher unnötig viele Positionen betrachtet und dabei unnötig viele Vergleiche durchführt.

Betrachten wir ein weiteres Beispiel (von nun an richten wir den Versatz immer am bisher rechtesten von $t$ betrachteten Zeichen aus):




Ist der Vergleich zwischen $w$ und $t$ an der betrachteten Position abgeschlossen (in unserem Beispiel durch den erfolglosen Vergleich zwischen i und n), so können wir $w$ so weit nach rechts schieben, bis das i im Text mit dem rechtesten i in $w$ in Überdeckung gebracht wird. Bei jeder kürzeren Verschiebung von $w$ nach rechts würde irgendwann das i im Text mit einem Zeichen von $w$ ungleich i verglichen, was wir schon wissen konnten und somit vermeiden sollten. Verschieben wir beispielsweise $w =$ finden nur um 2 Positionen, steht das d unter dem i; eine programmierte Nichtübereinstimmung. Nur der Versuch, $w$ an der zuvor dargestellten Position zu suchen, ist sinnvoll, alle kürzeren Verschiebungen von $w$ enden definitiv in einer Nichtübereinstimmung. Steht in $t$ an der betrachteten Stelle ein Zeichen (im nachfolgenden Beispiel das i), das in $w$ gar nicht vorkommt, so können wir entsprechend $w$ um $m$ (hier also um 6) Positionen nach rechts verschieben, ohne ein Vorkommen zu verpassen:



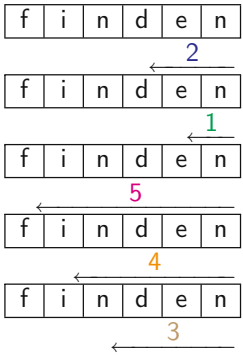

Das Wesentliche ist nun, dass die Anzahl der Positionen, um die wir $w$ verschieben können, nur in Kenntnis von $w$ (unabhängig von der gerade betrachteten Position) bestimmt werden kann. Dazu müssen wir für ein zu suchendes Wort $w$ nur einmal bestimmen, mit welchem Abstand zum rechten Ende ein jedes überhaupt mögliche Zeichen am weitesten rechts in $w$ vorkommt. Diese Information speichern wir in einer Tabelle (einem Feld) $D$ ab, die für jedes mögliche Zeichen einen Eintrag besitzt. Im folgenden verwenden wir dann $D$[a], um den Eintrag zum Zeichen a zu adressieren, für die anderen Zeichen entsprechend. Ist dabei beispielsweise das vorletzte Zeichen von $w$ ein e, so setzen wir $D$[e] $= 1$ (Zeichen e ist eine Position vom rechten Ende von $w$ entfernt). Ist das Zeichen v zwar prinzipiell möglich, kommt aber in unserem $w$ nicht vor, so setzen wir $D$[v] $= m$ (denselben Eintrag speichern wir in $D$ für alle möglichen in $w$ nicht benutzten Zeichen ab). Folgende Beispiele verdeutlichen dieses Vorgehen. Zur übersichtlicheren Darstellung wurden die Spalten der in $w$ nicht vorhandenen Zeichen weggelassen (deren Einträge, wie gesagt, alle gleich der Länge $m$ des Wortes $w$ sind):

$w =$ finden

Tabelle $D =$

| d | e | f | i | n |
|---|---|---|---|---|
| 2 | 1 | 5 | 4 | 3 |

Begründung:




Das letzte (rechte) n bleibt dabei unberücksichtigt, da es zu einem Eintrag von 0 in der Tabelle führen würde, also um einen *Versatz* des Wortes $w$ um keine Position.

$w =$ eine

Tabelle $D =$

| e | i | n |
|---|---|---|
| 3 | 2 | 1 |

Begründung:
Im Wort $w =$ eine ist das rechteste e 3 Zeichen vom rechten Ende entfernt. Wir müssen dabei auch hier das letzte Zeichen (ein e) ignorieren, da seine Berücksichtigung einen Versatz um 0 Positionen bewirken würde. Das rechteste i ist 2 und das rechteste n ist 1 Zeichen vom rechten Ende entfernt.

$w =$ Nadel

Tabelle $D =$

| a | d | e | l | N |
|---|---|---|---|---|
| 3 | 2 | 1 | 5 | 4 |

Begründung:
Im Wort $w =$ Nadel ist das rechteste a 3, das rechteste d 2, das rechteste e 1, das rechteste N 4 Zeichen vom rechten Ende entfernt. Da das Zeichen l nur als letztes Zeichen vorkommt, dieses aber ignoriert wird, ist der Eintrag für l derselbe, als gäbe es kein l im Wort, also gleich der Länge 5 des Wortes.

Allgemein können wir die Einträge von $D$ durch folgende Formel beschreiben:

$$D[\mathsf{x}] = \begin{cases} m & \text{falls } \mathsf{x} \text{ keines der } m-1 \text{ vorderen Zeichen von } w \text{ ist,} \\ m-i & \text{falls } i \text{ die rechteste Position} \neq m \text{ ist, an der } \mathsf{x} \text{ in } w \text{ steht.} \end{cases}$$

Der erste Fall signalisiert gemäß obiger Beispiele, dass wir $w$ um seine ganze Länge verschieben können, tritt das betrachtete Textsymbol nirgends oder nur an letzter Position in $w$ auf. Um die Berechnung von $D$ zu programmieren, müssen wir einfach zwei Schleifen wie folgt hintereinander ausführen:

Die Berechnung der Tabelle $D$.

```
1   for all symbols x do
2       D[x] := m;
          // D[x] = m für ein in w nicht vorkommendes Zeichen x

3   for i := 1 to m − 1 do
4       D[w[i]] := m − i;
          // Für die gesehenen Zeichen überschreibe die Initialisierung
```

Wir sind nun soweit und können aus unserem naiven Algorithmus den sogenannten Boyer-Moore-Horspool Algorithmus machen. Es handelt sich dabei um einen Algorithmus für das String-Matching-Problem, den R. Horspool 1980 als eine Vereinfachung des Algorithmus von Boyer und Moore publizierte (siehe auch Abschnitt *Zum Weiterlesen*). Dazu müssen wir nur

1. vor der Suche einmalig $D$ berechnen
2. die Zeile $pos := pos + 1$ durch $pos := pos + D[t[pos + m - 1]]$ ersetzen.

Wir erhalten so (die Berechnung von $D$ unberücksichtigt):

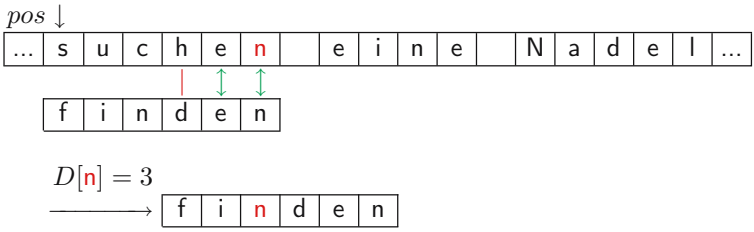
Der Boyer-Moore-Horspool Algorithmus.

```
1   procedure BMH
2   pos := 1;
3   while pos ≤ n − m + 1 do begin      // suche an allen Positionen
4       j := m;
5       while (j > 0) and (w[j] = t[pos + j − 1]) do
6           j := j − 1;
7       if (j = 0) then print("Vorkommen an Position", pos);
8       pos := pos + D[t[pos + m − 1]];
9   wend;
10  end.
```

Kommt dann der Algorithmus bei seiner Ausführung an die Stelle, an der $w$ weiter nach rechts verschoben ($pos$ erhöht) wird, so wird sichergestellt, dass an der neuen Position ein Zeichen in $w$ mit dem bisherigen $t[pos+m-1]$

übereinstimmt, ohne dabei ein Vorkommen von $w$ zu verpassen. Auch kann es geschehen, dass $w$ ganz an dem Zeichen $t[pos + m - 1]$ vorbeigeschoben wird, falls diese Übereinstimmung unmöglich ist.

**Beispiel:**




Dabei wissen wir, dass die beiden rot dargestellten Zeichen n übereinstimmen, ohne dass wir den entsprechenden Vergleich durchgeführt haben.

Doch was haben wir durch diese Änderung an unserem Programm gewonnen? Zunächst ist zu sagen, dass wir für die denkbar schlechteste Eingabe gar keine Effizienzsteigerung erreicht haben, da dort $D[\mathsf{x}] = 1$ für alle im Text vorkommenden Zeichen x gilt (der Text besteht für einen solchen schlechtesten Fall also aus einer Wiederholung des vorletzten Zeichens in $w$), wir also $w$ weiterhin in Einerschritten nach rechts verschieben. Ein Beispiel für eine solche Eingabe ist der Text

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a | a |

zusammen mit dem Wort $w =$ baaaa. In diesem Fall werden für jede Position 5 Vergleiche durchgeführt, da die vier a des Wortes stets mit den Zeichen des Textes übereinstimmen und ein fünfter Vergleich erforderlich ist, um festzustellen, dass $w$ an der betrachteten Position nicht vorkommt. Da für dieses $w$ zusätzlich $D[\mathsf{a}] = 1$ gilt, werden so insgesamt $18 \times 5 = 90$ Vergleiche durchgeführt.

Natürlich ist es sehr unwahrscheinlich, dass in der praktischen Anwendung ein solcher Text mit einem solchen Suchwort vorkommt, und tatsächlich ist der neue Algorithmus in den allermeisten Fällen der Anwendung wesentlich schneller als unsere naive Lösung. Betrachten wir zum Vergleich noch einmal das Beispiel des Textes

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | u | c | h |  | d | i | e |  | N | a | d | e | l |  | i | m |  | H | e | u | . |

mit dem Suchwort $w =$ Nadel, für das wir ja schon wissen, dass $D$ wie folgt aussieht:

| a | d | e | l | N |
|---|---|---|---|---|
| 3 | 2 | 1 | 5 | 4 |

Wie man leicht nachprüfen kann, stellt der naive Algorithmus 22 Vergleiche zwischen Text und Wort an, um das eine Vorkommen von $w$ zu finden. Unser verbesserter Algorithmus ist da wesentlich schneller. Um $D$ zu bestimmen, muss er vor der Suche 5 Zeichen betrachten. Für die eigentliche Suche benötigt er dann noch 8 Vergleiche, von denen alleine 5 zur Feststellung des einen Vorkommens von Nadel notwendig sind. Da unser Text 22 Zeichen lang ist, können wir folglich einen Text nach allen Vorkommen eines Wortes durchsuchen, ohne alle Zeichen des Textes zu betrachten – im ersten Moment klingt das verrückt. Schlüssel zu diesem (in den allermeisten Fällen) effizienten Vorgehen war der Vergleich von Text und Wort zeichenweise von rechts nach links. Nur so betrachten wir anfangs ein Textsymbol (nämlich $t[pos + m - 1]$), das für spätere Positionen erneut mit $w$ in Übereinstimmung zu bringen ist, das also, sofern es nicht oder nur weit links in $w$ vorkommt, einen Versatz um viele Positionen impliziert. Damit ist der Vergleich von rechts nach links Grundlage unserer Verbesserung – eine kleine Modifikation mit großer Wirkung also.

## Zum Weiterlesen

1. Kapitel 1 (Binäre Suche)

   Hier geht es um die schnelle Suche nach Daten, die durch einen so genannten *Schlüssel* adressiert werden. Ein Schlüssel muss dabei ein Datum eindeutig identifizieren, so wie beispielsweise ein KFZ-Kennzeichen ein Auto oder eine ISBN-Nummer ein Buch eindeutig bezeichnen. Aus der Sortierung der Daten gemäß ihres Schlüssels können wir dann Kapital schlagen und jedes Element in kürzester Zeit finden.

2. Kapitel 20 (Hashing)

   In diesem Kapitel wird eine weitere Idee vorgestellt, wie wir eine Menge von Daten effizient verwalten können. Auch hier geht man davon aus, dass alle Daten durch einen Schlüssel eindeutig identifiziert werden. Man benutzt den Schlüssel dann, um aus ihm eine Adresse (Position) innerhalb eines Speicherbereichs abzuleiten, unter der das zugehörige Element abgespeichert wird.

3. `http://de.wikipedia.org/wiki/Boyer-Moore-Algorithmus`

   Dieser Wikipedia-Artikel behandelt den Boyer-Moore-Algorithmus für das String Matching Problem. Dieser Algorithmus ist eine Variante des hier behandelten Boyer-Moore-Horspool Verfahrens, das verschiedene Heuristiken einsetzt, um das Wort $w$ in möglichst großen Schritten entlang des Textes $t$ zu verschieben.

4. `http://de.wikipedia.org/wiki/String-Matching-Algorithmus`

   Dieser Artikel betrachtet das Problem des String Matching im Allgemeinen und enthält Verweise auf viele verschiedene Ansätze, wie man alle Vorkommen eines Wortes in einem Text mit Hilfe eines Algorithmus finden kann.

# 7

# Tiefensuche (Ariadne und Co.)

Michael Dom, Falk Hüffner und Rolf Niedermeier

Friedrich-Schiller-Universität Jena

> „Das eben geschieht den Menschen, die in einem Irrgarten hastig werden: Eben die Eile führt immer tiefer in die Irre."
>
> Lucius Annaeus Seneca (4 n. Chr. – 65 n. Chr.)

Ariadne, nach griechischer Sage die Tochter von Minos, dem König von Kreta, verliebte sich in Theseus. Dieser Athener Held war beauftragt, den Minotaurus (ein Ungeheuer halb Mensch, halb Stier) zu töten. Die Herausforderung wurde ungleich größer dadurch, dass der Minotaurus im Labyrinth versteckt war. Die kluge Ariadne stattete ihren Helden mit einer Rolle Faden aus: Indem Theseus ein Fadenende am Eingang des Labyrinths festknotete und den Faden beim Durchforschen des Labyrinths abrollte, konnte er zum einen vermeiden, gleiche Teile des Labyrinths mehrfach zu durchsuchen, und zum anderen sicherstellen, auch wieder den Weg zurück in Ariadnes Arme zu finden.

Nicht nur die alten Griechen mussten sich mit der geschickten Durchmusterung von Suchräumen, wie z.B. Labyrinthen, befassen, sondern auch in der Informatik von heute spielt diese Aufgabe eine zentrale Rolle. Eine Methode hierzu stellt die Tiefensuche (englisch *depth-first search*) dar, die wir nachfolgend genauer betrachten wollen.

## Algorithmische Idee und Umsetzung

Wie bereits angekündigt, gilt es die Aufgabe zu lösen, ein Labyrinth vollständig zu durchsuchen. Ein Labyrinth ist dabei ein aus Gängen, Sackgassen

und Kreuzungen bestehendes Gebilde – die Aufgabe besteht somit darin, jede Kreuzung und jede Sackgasse mindestens einmal zu besuchen. Wünschenswert wäre zudem, wenn kein Gang des Labyrinths mehr als einmal pro Richtung durchlaufen werden würde – schließlich soll Theseus am Ende noch genügend Kraft übrig haben, um weder beim Minotaurus noch bei Ariadne schlappzumachen.

Die wahrscheinlich einfachste Idee zur Lösung dieses Problems ist, vom Startpunkt aus einfach immer weiter ins Labyrinth hineinzulaufen und alle Kreuzungen, auf die man unterwegs stößt, als erledigt abzuhaken. Landet man in einer Sackgasse oder auf einer Kreuzung, die man schon kennt, dreht man einfach um, geht zur letzten Kreuzung zurück und versucht es von dort aus erneut in einer anderen, noch unbekannten Richtung. Gibt es von der letzten Kreuzung aus keine unbekannte Richtung mehr, so geht man von dort aus nochmals eine Kreuzung weiter zurück usw.

Führt diese Vorgehensweise auch zum Ziel? Sehen wir uns die Suche etwas genauer an: Anstatt eines Fadens benutzen wir, der einfacheren Beschreibung wegen, ein Stück Kreide. Mit diesem markieren wir an jeder Kreuzung die abgehenden Gänge, und zwar mit einem Haken für bereits einmal durchlaufene Gänge und mit zwei Haken für zweimal durchlaufene. Konkret lauten die Regeln für unsere Suche im Labyrinth folgendermaßen.

- Befindet man sich in einer Sackgasse, so dreht man um und geht zurück zur letzten Kreuzung.
- Hat man dagegen eine Kreuzung erreicht, zeichnet man erstmal einen Haken an die Wand des Ganges, durch den man gekommen ist, um später ggf. wieder zurückfinden zu können. Anschließend gibt es mehrere Möglichkeiten:
  1. Zunächst kontrolliert man, ob man im Kreis gelaufen ist: Wenn der Gang, durch den man gekommen ist, soeben seinen ersten Haken bekommen hat und wenn außerdem noch weitere Haken an anderen Gängen der Kreuzung sichtbar sind, so ist dies der Fall, und man macht einen zweiten Haken an den Gang, aus dem man kam, und dreht um.
  2. Ansonsten prüft man, ob die Kreuzung noch unerkundete Gänge hat: Falls es noch Gänge ohne Markierungen gibt, so wählt man von diesen einen beliebigen – sagen wir, den ersten von links – aus, zeichnet dort einen Haken an die Wand und verlässt die Kreuzung durch diesen Gang. (Dieser Fall gilt übrigens auch zu Beginn der Suche an der Startkreuzung!)
  3. Andernfalls gibt es höchstens einen Gang mit nur einem Haken, und alle anderen Gänge haben zwei Haken. Man hat also bereits alle von der aktuellen Kreuzung abgehenden Gänge untersucht und geht durch den Gang zurück, der nur einen Haken hat, wobei man diesem Gang der Ordnung halber einen zweiten Haken verpasst. Gibt es gar keinen solchen Gang, d. h., haben gar alle Gänge zwei Haken, so steht man wieder am Start und hat das Labyrinth vollständig durchsucht.

Betrachten wir nun folgendes, in Abb. 7.1 dargestellte Beispiel, in dem ein Weg vom Start A zum Ziel F gesucht wird. (Es soll also wiederum das ganze Labyrinth durchsucht werden, wobei die Suche jedoch abgebrochen werden kann, sobald F gefunden wird.) Wir gehen davon aus, dass eine Sackgasse nur unmittelbar vor ihrem Ende als solche erkannt werden kann.

Man geht los vom Start A Richtung Norden. Die erste Kreuzung ist C. Dort macht man eine Markierung am Südausgang der Kreuzung (1). Natürlich ist hier sonst noch keine Markierung, also nimmt man den ersten unmarkierten Weg von links, d. h. den in Richtung Westen, und markiert ihn (2). Dann erreicht man bei B eine Sackgasse, also dreht man um. Wieder bei C angelangt, hat der Weg nach Westen jetzt schon zwei Markierungen, der nach Süden eine, aber nach Norden ist noch gar nichts markiert, also geht man dort lang. Bei E ist wieder eine unberührte Kreuzung, und man wählt unter den drei Möglichkeiten die nach Westen. Nach zwei Kurven geht man in G geradeaus über die Kreuzung, zwei Markierungen hinterlassend (7 und 8). In H trifft man auf eine Sackgasse, also dreht man wieder um. In G ist nur eine Möglichkeit erlaubt: nach Süden zu E. Hier tritt zum ersten Mal die Regel gegen das Im-Kreis-Laufen in Kraft: Man hat beim Betreten von E einen Haken am Nordausgang von E gemacht (11); außerdem existiert an dieser Kreuzung bereits eine Markierung am Südausgang (5) und eine am Westausgang (6) – demnach muss man also umdrehen. Über die Kreuzung G und zwei Kurven geht es zurück, sodass nun der nördliche Teil komplett abgesucht ist und man wieder bei E landet. Nach Osten gibt's hier noch gar keine Markierung, also geht man dort lang. Schließlich erreicht man in F das Ziel.

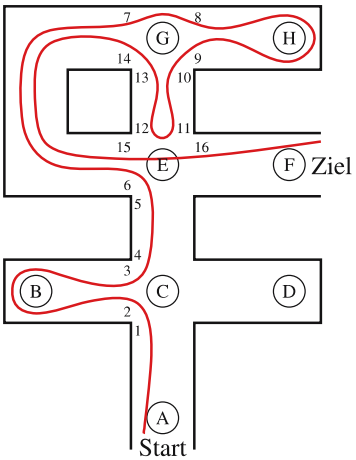



**Abb. 7.1.** Beispiel für die Tiefensuche im Labyrinth. Es wird, ausgehend von A, ein Weg nach F gesucht. Die Ziffern kennzeichnen die Stellen, an denen Kreidemarkierungen hinterlassen werden

Das Prinzip, das wir hier kennengelernt haben, nennt sich *Tiefensuche*, da man, wie schon erwähnt, immer möglichst tief in das Labyrinth hineingeht und nur dann, wenn es nicht mehr weitergeht oder man auf eine schon bekannte Stelle trifft, ein Stück zurückgeht und es von einem früheren Punkt in anderer Richtung erneut versucht.

Die Regeln für die Tiefensuche sind so einfach, dass man sie mit wenigen Zeilen einem Computer beibringen kann. Dabei merkt man sich für jede Kreuzung einen „Zustand“, wobei am Anfang alle Kreuzungen „unentdeckt“ sind. Wird die TIEFENSUCHE-Funktion an einer Kreuzung $X$ gestartet, so wird zunächst getestet, ob man im Kreis gelaufen ist (Zeile 2 im nachfolgend abgebildeten Programmstück). Als nächstes wird geschaut, ob man das Ziel gefunden hat (Zeile 3) – falls ja, so wird das Programm mit dem Befehl „exit“ beendet, und die Suche ist zu Ende. Andernfalls geht es weiter, und die Kreuzung $X$ wird als „entdeckt“ markiert (Zeile 4). Nun müssen alle noch nicht erkundeten benachbarten Kreuzungen besucht werden – dies geschieht, indem sich die TIEFENSUCHE-Funktion für jede benachbarte Kreuzung $Y$ selbst aufruft (Zeilen 5–7). Dies ist ein beim Programmieren häufig benutzter Trick namens *Rekursion*, der schon in Kap. 1 beschrieben wurde. Bemerkt die frisch aufgerufene TIEFENSUCHE-Funktion, dass $Y$ schon besucht wurde und man somit im Kreis gelaufen ist, so kehrt sie sofort (Zeile 2) per „return“ zur aufrufenden Funktion an der Kreuzung $X$ zurück. Andernfalls geht es nun an der Kreuzung $Y$ weiter...

TIEFENSUCHE I

```
1  function TIEFENSUCHE(X):
       // Definition der TIEFENSUCHE-Funktion
2      if Zustand[X] = „entdeckt“ then return; endif
3      if X = Ziel then exit „Ziel gefunden!“; endif
4      Zustand[X] := „entdeckt“;
5      for each benachbarte Kreuzung Y von X
6          TIEFENSUCHE(Y);
7      end for
8  end function     // Ende der TIEFENSUCHE-Funktion
9  TIEFENSUCHE(Startkreuzung);     // Hauptprogramm
```

Manchmal möchte man Rekursionen vermeiden, z.B. weil bei jedem rekursiven Funktionsaufruf zusätzliche Zeit für das Anlegen von Variablen im Speicher etc. benötigt wird. In diesem Fall kann die Tiefensuche mit Hilfe eines *Stapels* auch ohne Rekursion programmiert werden. Unter einem Stapel versteht man dabei eine Datenstruktur, die es erlaubt, Objekte (in unserem Fall Kreuzungen) oben auf den Stapel zu legen oder aber das momentan oberste Objekt vom Stapel zu nehmen. Der Stapel dient bei uns dazu, den „Rückweg“ zu speichern, man legt also immer die Kreuzung $X$, die man gerade verlässt, oben auf den Stapel, und zwar zusammen mit einer Zahl „Ausgänge“, die angibt, zu wie vielen benachbarten Kreuzungen man von $X$ aus schon aufge-

brochen ist. Zu jeder Kreuzung existiert eine Liste (genauer: ein Array), in der alle Nachbarkreuzungen verzeichnet sind – somit kann man bei Bedarf gezielt beispielsweise die fünfte Nachbarkreuzung einer Kreuzung $X$ auswählen.

TIEFENSUCHE II

```
 1  X := Startkreuzung;   Ausgänge := 0;
 2  repeat
 3     if Zustand[X] ≠ „entdeckt“ then
 4        if X = Ziel then exit „Ziel gefunden!“; endif
 5        Zustand[X] := „entdeckt“;
 6     else
 7        nimm das oberste Paar (X, Ausgänge) vom Stapel;
 8     endif
 9     if Ausgänge < Anzahl benachbarter Kreuzungen von X then
10        Ausgänge := Ausgänge + 1;
11        lege das Paar (X, Ausgänge) oben auf den Stapel;
12        X := (Ausgänge)-te benachbarte Kreuzung von X;
13        Ausgänge := 0;
14     else
15        if Stapel ist leer then
16           exit „Ziel nicht gefunden!“;
17        else
18           nimm das oberste Paar (X, Ausgänge) vom Stapel;
19           gehe zu Zeile 9;
20        endif
21     endif
22  end repeat
```

## Anwendungen

Das Verfahren TIEFENSUCHE funktioniert nicht nur für Labyrinthe, sondern findet auch in deutlich anderen Zusammenhängen Anwendung, wie wir im folgenden Abschnitt sehen werden:

### Suche im Web

Im folgenden Anwendungsbeispiel geht es nochmals ums Suchen, allerdings irrt hier nicht Theseus im Labyrinth umher, sondern ein Schüler namens Sinon sucht eine bestimmte Webseite.

Sinon Schneider war kürzlich auf einer Party seiner Mitschülerin Ariadne und ist dort kurz mit einem netten Mädchen ins Gespräch gekommen. Er würde sie nun gerne wiedersehen, aber dummerweise hat er sie nicht nach ihrem Namen gefragt. Was tun? Sinon könnte natürlich die Gastgeberin Ariadne fragen, aber erstens traut er sich nicht, und zweitens kennt sie selbst auch

nur einen Teil ihrer Partygäste. Schließlich kommt Sinon die rettende Idee: Wieso nicht einfach im Internet bei „GaudiVZ.de“ nachsehen? In diesem berühmten Schülerverzeichnis haben fast alle Schüler in Deutschland ein Profil von sich angelegt, meistens sogar mit einem Photo und Links zu den Profilen von Freunden. Also könnte Sinon doch im GaudiVZ Ariadnes Profil besuchen und sich von dort aus durch alle ihre Freunde und alle Freunde von ihren auf der Party anwesenden Freunden und alle Freunde von Freunden von Freunden usw. klicken, bis er entweder seine Angebetete gefunden hat (hoffentlich hat sie ein Bild von sich eingestellt!) oder aber das ganze in Frage kommende Umfeld von Ariadne abgesucht hat. Sinon steht also vor folgender Aufgabe: Durchmusterung aller Profile des GaudiVZ, die von Ariadne aus über Links erreichbar sind und deren Eigentümer er auf der Party gesehen hat. Auch hier liegt die Schwierigkeit darin, einerseits nicht endlos im Kreis zu laufen und andererseits alles gewissenhaft und systematisch zu überprüfen. Dies lässt sich effektiv erledigen mit Hilfe einer Tiefensuche im Geflecht der einzelnen Profile des GaudiVZ.

Nehmen wir also an, Sinon beginnt mit seiner Suche und startet beim Profil von Ariadne. Also klickt er auf den ersten Link in Ariadnes Freundeliste und landet beim Profil von Theseus (siehe Abb. 7.2). Da es sich dabei nicht um die von ihm gesuchte Person handelt, geht es weiter mit dem ersten Link dieser Seite, wobei Sinon jedoch darauf achtet, keinen Link anzuklicken, der zu einem Profil führt, das er bereits besucht hat – solche Links erkennt er als geübter Websurfer daran, dass sie in seinem Browser in Violett anstatt in Blau erscheinen (diese hilfreiche Funktion des Browsers entspricht also gewissermaßen dem Malen von Kreidehaken in unserem ersten Beispiel). Sobald er schließlich ein Profil erreicht, dessen Benutzer seiner Meinung nach nicht auf der Party war oder wenn alle Freunde eines Profils abgearbeitet – d. h. besucht und daher violett dargestellt – sind, so klickt er auf den „Zurück“-Knopf seines Browsers und fährt mit den Freunden des nun angezeigten Profils fort. Wie die Tiefensuche II verwendet die „Zurück“-Funktion des Browsers einen Stapel, auf den beim Anklicken eines Links die Adresse der aktuellen Seite abgelegt und von dem beim Benutzen des „Zurück“-Knopfs die oberste Adresse entnommen wird.

Wenn Sinons Herzensdame ein Bild von sich eingestellt hat und wenn sie von Ariadnes Profil aus über eine Folge von verlinkten Profilen erreichbar ist, deren Besitzer alle auf der Party waren (wovon wir ja ausgegangen sind), dann wird er sie mit dieser Methode zwangsläufig finden! Sollte er allerdings irgendwann beim Klicken auf den „Zurück“-Knopf zum wiederholten Male beim Profil von Ariadne landen, und die Links zu all ihren Freunden sind violett gefärbt, d.h. schon besucht, dann würde das für ihn bedeuten, dass er Pech gehabt hat – aber immerhin könnte er sich sicher sein, keines der in Frage kommenden Profile ausgelassen zu haben!

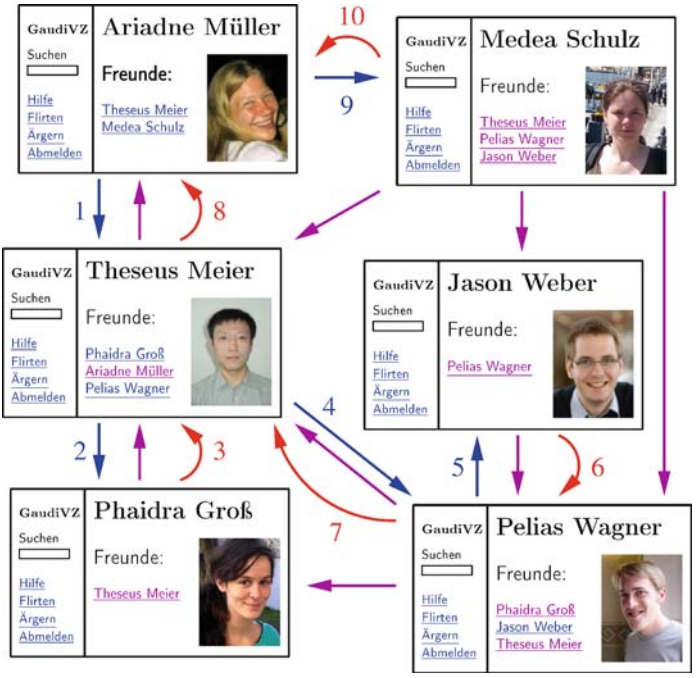
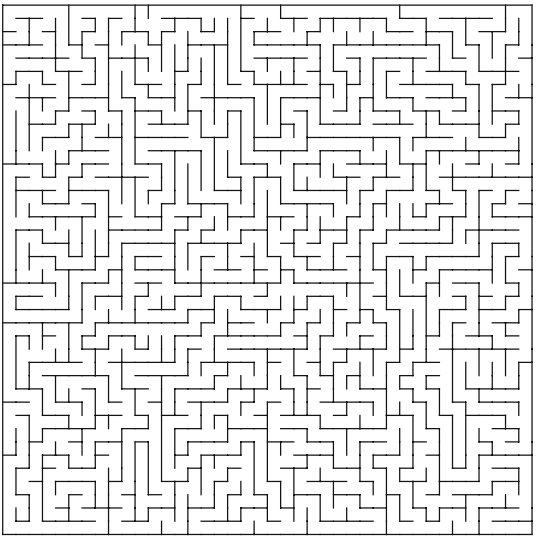



**Abb. 7.2.** Tiefensuche im GaudiVZ: Die Zahlen verdeutlichen die Reihenfolge der Sprünge von Seite zu Seite. Ein gerader Pfeil bedeutet, dass eine Seite einen Link auf eine andere Seite enthält. Gebogene Pfeile bedeuten, dass an dieser Stelle der „Zurück"-Knopf benutzt wird. Man beachte, dass nicht alle dargestellten Links angeklickt werden, da manche davon zum Zeitpunkt des Besuchs bereits in Violett angezeigt werden

### Erstellen von Labyrinthen

Nicht nur für Theseus, sondern auch für den Minotaurus ist Tiefensuche nützlich. Sie kann nämlich auch zum Entwurf besonders verwirrender Labyrinthe benutzt werden. Die Vorgehensweise ist recht einfach: Man startet mit einem regelmäßigen rechtwinkligen Gitter. Die Tiefensuche startet in einer beliebigen Zelle. Dann wird die Tiefensuche für alle Nachbarzellen in zufälliger Reihenfolge aufgerufen (dabei kann beispielsweise der Zufallszahlen-Algorithmus aus Kap. 25 helfen). Wird dabei eine Zelle zum ersten Mal besucht, so wird die Wand zur Vorgängerzelle eingerissen. Es ergibt sich ein Muster wie das folgende:



Da die Tiefensuche letztlich jede Zelle besucht, muss es von jeder Zelle einen Weg zur Startzelle geben und somit auch von jeder Zelle zu jeder anderen – nur leicht zu finden ist dieser Weg nicht unbedingt...

### Weitere Beispiele: Unterhaltungsshows und Verkehrsplanung

Wie wir gesehen haben, ist Tiefensuche ein sehr einfaches Grundprinzip; in der wissenschaftlichen Literatur lässt sich eine Vielzahl von Anwendungsbeispielen finden. Wir wollen hier nur noch zwei „lebensnahe" erwähnen – weitere Beispiele finden sich in anderen Kapiteln dieses Buchs (siehe Hinweise am Ende des Kapitels).

Angenommen, die Fernsehshow „Sick Sister" plant die Produktion von zwei neuen Staffeln. Bei dieser beliebten Show werden die Teilnehmer in einen Container gesteckt und rund um die Uhr mit Fernsehkameras beobachtet. Damit das Ganze nicht zu langweilig wird, sollten sich die Teilnehmer möglichst oft in die Haare geraten, was wiederum bedeutet, dass bei jeder Staffel möglichst keine zwei Teilnehmer im Container sein sollten, die sich sympathisch sind. Nehmen wir nun an, die Teilnehmer für die nächsten zwei Staffeln der Show sind bereits ausgewählt und es geht nun nur noch darum, für jeden der Teilnehmer festzulegen, ob er bei der ersten oder der zweiten Staffel mitmachen soll, wobei natürlich die erwähnte Regel „keine Personen im Container, die sich mögen" einzuhalten ist.

Zur Lösung dieser Aufgabe könnte man zunächst einen „Sympathiegraphen" erstellen, d. h., man zeichnet auf ein Blatt Papier für jeden Teilnehmer einen Kringel – *Knoten* genannt – und verbindet zwei Kringel durch eine Linie – *Kante* genannt –, wenn sich die entsprechenden Personen sympathisch sind

(siehe Abb. 7.3 für ein Beispiel).[1] Die Aufgabe ist nun, die Knoten mit einer von zwei Farben jeweils so anzumalen, dass jeder Knoten genau eine Farbe erhält und dass keine zwei Knoten, die durch eine Kante verbunden sind, die gleiche Farbe erhalten. Die Farbe eines Knotens gibt am Ende an, bei welcher der beiden Staffeln die entsprechende Person teilnehmen soll. Die gesuchte „Zweifärbung" des Graphen kann nun mit Hilfe einer Tiefensuche gefunden werden: Man wählt zunächst einen beliebigen Knoten aus und gibt ihm eine beliebige der beiden zur Verfügung stehenden Farben. Anschließend startet man bei diesem Knoten die Tiefensuche. Immer dann, wenn man, ausgehend von einem Knoten $X$, auf einen noch unentdeckten Knoten $Y$ stößt, gibt man diesem die Farbe, die $X$ nicht hat. Kommt man hingegen von einem Knoten $X$ zu einem bereits entdeckten Knoten $Y$, so kontrolliert man, ob $X$ und $Y$ unterschiedliche Farben haben. Ist dies nicht der Fall (hat man also zwei verbundene Knoten derselben Farbe gefunden), so kann der Graph nicht wie gewünscht zweigefärbt werden. Andernfalls liefert die Tiefensuche die gewünschte Zweifärbung und sorgt so für einen hohen Unterhaltungswert von „Sick Sister".[2]

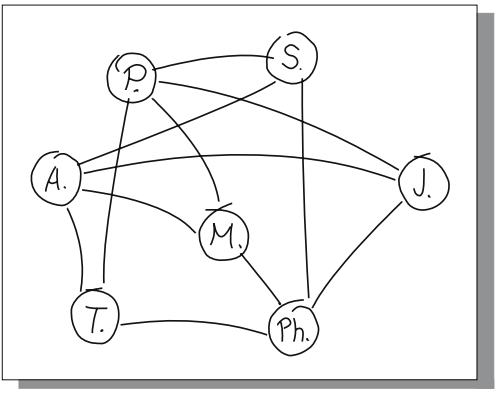



**Abb. 7.3.** Ein „Sympathiegraph": Sind sich zwei Personen sympathisch, so sind die entsprechenden beiden Knoten durch eine Linie miteinander verbunden

[1] Diese Art von Graphen mit Knoten und Kanten hat übrigens nichts zu tun mit Graphen von Funktionen, wie man sie aus der Analysis kennt. Mit „Knoten-und-Kanten"-Graphen können die unterschiedlichsten Sachverhalte und Objekte modelliert werden, z. B. auch Labyrinthe: Jede Sackgasse und jede Kreuzung wird dann jeweils zu einem Knoten und jeder Gang zu einer Kante.

[2] In dem Spezialfall, dass der Sympathiegraph aus mehreren, untereinander nicht verbundenen Teilen, den so genannten „Zusammenhangskomponenten", besteht, muss man die Tiefensuche für jeden Teil einzeln anwenden. Dabei bietet sich unter Umständen sogar die Möglichkeit, die Teilnehmer zahlenmäßig möglichst ausgewogen auf die zwei Staffeln zu verteilen, indem man nämlich im Nachhinein bei manchen Zusammenhangskomponenten die beiden Farben vertauscht.

Graphen, die wie beschrieben zweigefärbt werden können, werden übrigens *bipartit* genannt – es sind dies genau diejenigen Graphen, die keinen Kreis ungerader Länge enthalten. Sollen die Teilnehmer allerdings auf drei statt auf zwei Staffeln aufgeteilt werden, so wird die Aufgabe plötzlich sehr viel schwieriger, und niemand weiß, ob man sie mit Hilfe von Tiefensuche effizient lösen kann. (Das Problem ist, dass man beim Besuchen eines bisher noch unentdeckten Knotens zwei Farben zur Auswahl hat und nicht weiß, welche davon man auswählen soll.)

Eine weitere Anwendung der Tiefensuche ist die folgende: Angenommen, Stadtrat Hermes will zwecks Verkehrsberuhigung in der Innenstadt eine Vielzahl von Straßen zu Einbahnstraßen erklären. Dabei muss Hermes sich aber vor dem Zorn der Autofahrer in Acht nehmen und dafür Sorge tragen, dass das Straßennetz nicht so eingeschränkt wird, dass abgeschnittene „Inseln“ entstehen, die man nicht erreichen oder verlassen kann, sprich, dass ein Autofahrer nicht mehr von überall nach überall kommen kann. Graphentheoretisch modelliert bedeutet das, dass der zugrundeliegende Straßennetzgraph weiterhin eine einzige „starke Zusammenhangskomponente“ bleiben muss. Auch dieses Problem kann effizient mit Tiefensuche behandelt werden – in Kap. 9 wird hierauf genauer eingegangen.

## Breitensuche

Ein Problem bei der Tiefensuche ist, dass man sich schnell sehr weit vom Start entfernen kann. In vielen Fällen weiß man aber, dass das Ziel nicht allzuweit weg ist; im GaudiVZ etwa kann man davon ausgehen, dass sich das gesuchte Profil in einer Distanz von z. B. höchstens drei zur Gastgeberin befindet. In diesem Fall bietet sich die Breitensuche (auf englisch *breadth-first search*) an: Sie sucht den Graphen um den Ausgangspunkt schichtenweise ab, also erst alle direkten Nachbarn (Abstand 1), dann alle Knoten im Abstand 2 usw. Dazu benutzt man die Datenstruktur „Warteschlange“, in die man einen Knoten hinten einstellen kann und aus der man vorne einen Knoten herausnehmen kann, zu dem man dann springt. Für die Labyrinthsuche ist Breitensuche also nicht geeignet: Man kann nicht einfach eine Kreuzung auf einer Liste notieren und dann bei Bedarf dort „hinspringen“. Für viele andere Anwendungen (wie die Web-Suche) ist das jedoch kein Problem.

Das unten angegebene Programmstück zeigt die Breitensuche im Detail. Dabei werden in der Warteschlange immer die Knoten gespeichert, die noch besucht werden müssen. Am Anfang kommt also der Startknoten in die Warteschlange (Zeile 2). Solange die Warteschlange nicht leer ist, wird immer der erste Knoten herausgenommen (Zeilen 3 und 4). Dann werden alle Nachbarn dieses Knotens in die Warteschlange eingefügt (Zeilen 8 und 9). Damit wir von einem Knoten nicht mehrmals die Nachbarn absuchen, wird ein so behandelter Knoten als „entdeckt“ markiert (Zeile 7), und bereits markierte Knoten werden übersprungen (Zeile 5).

Breitensuche

```
1   begin     // am Anfang ist die Warteschlange leer
2     stelle den Startknoten hinten in die Warteschlange;
3     while Warteschlange ist nicht leer
4       nimm den vordersten Knoten X aus der Warteschlange;
5       if Zustand[X] ≠ „entdeckt“ then
6         if X = Ziel then exit „Ziel gefunden!“; endif
7         Zustand[X] := „entdeckt“;
8         for each benachbarter Knoten Y von X
9           stelle Y hinten in die Warteschlange;
10        end for
11      endif
12    end while
13  end
```

Als Beispiel sehen wir uns den gleichen Graphen an, der bereits bei der Tiefensuche zum Einsatz kam: das Labyrinth (Abb. 7.4). Am Anfang hat man eine Warteschlange, die nur den Knoten A enthält. Dieser wird herausgenommen, und alle Nachbarn von A werden eingefügt: das ist nur C. Die sich ergebende Warteschlange ist neben A abgebildet. Weiter geht's beim vordersten Element der Warteschlange, also C. Streng genommen würden jetzt alle vier Nachbarn von C an die Warteschlange angehängt; als kleine Optimierung ignorieren wir jetzt jedoch A, da es schon den „entdeckt“-Status hat und beim Wiederherausnehmen aus der Warteschlange sowieso nichts bewirken würde. Es kommen also B, E und D hinzu. Bei B kommen gar keine neuen Knoten hinzu, es geht direkt weiter bei E. Dort werden dann G und F angehängt,

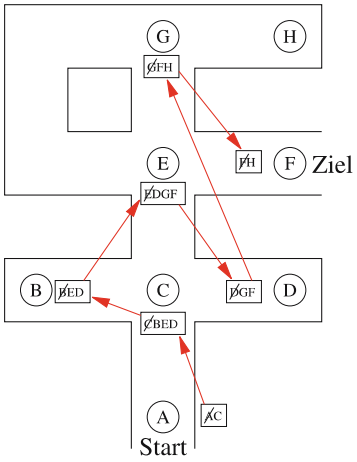



**Abb. 7.4.** Beispiel für die Breitensuche im Labyrinth

diesmal wird C ignoriert. Bei D passiert wieder nichts, es geht direkt weiter bei G. Dort wird noch H angehängt, aber bei F finden wir dann bereits das Ziel.

Man sieht sofort, dass die Reihenfolge, in der die Knoten besucht werden, ganz anders ist als bei der Tiefensuche (Abb. 7.1). Bei der Breitensuche werden nämlich die Knoten in der Reihenfolge ihrer Entfernung vom Startknoten besucht: Zuerst Knoten C (Abstand 1), dann B, E und D (Abstand 2), dann G und F (Abstand 3). Daraus ergibt sich auch, dass Breitensuche immer einen kürzestmöglichen Weg zum Ziel findet und außerdem zwischendurch keine längeren Pfade untersucht.

Übrigens: Verwendet man hier anstatt einer Warteschlange einen Stapel, so führt der Algorithmus statt einer Breitensuche eine Tiefensuche durch! Im Gegensatz zum Algorithmus Tiefensuche II wird dabei jedoch der Stapel nicht dazu verwendet, um den „Rückweg“ zu speichern, sondern um sich diejenigen Kreuzungen zu merken, zu denen man bereits die Gänge entdeckt hat, die man jedoch nicht gleich besucht hat. Dadurch kann der Stapel deutlich größer werden kann als bei Tiefensuche II.

Was soll man nun nehmen bei einem konkreten Problem, Tiefen- oder Breitensuche? Tiefensuche ist normalerweise etwas einfacher zu programmieren, da man sich bei Benutzung von Rekursion nicht explizit um eine Datenstruktur wie die Warteschlange bei Breitensuche kümmern muss. Außerdem benötigt Breitensuche im allgemeinen mehr Speicher; bei schwierigen Problemen kann es sogar sein, dass der vorhandene Speicher schlicht nicht ausreicht. Dafür findet Breitensuche immer einen kürzestmöglichen Pfad (gemessen an der Anzahl Kanten) und ist insbesondere dann schnell, wenn man einen sehr großen Graphen hat, sich das Ziel aber in der Nähe des Starts befindet; hier kann sich die Tiefensuche leicht in weit entfernten Regionen „verfranzen“. Es kommt also auf die konkrete Situation an, welcher Algorithmus der bessere ist.

## Zum Weiterlesen

1. Darstellungen zur Tiefensuche finden sich in den meisten Lehrbüchern über Algorithmen.
2. Kapitel 9 (Zyklensuche in Graphen)

   In diesem Kapitel wird eine weitere Anwendung für Tiefen- und Breitensuche vorgestellt.
3. Kapitel 8 (Der Pledge-Algorithmus)

   In unserem Labyrinth-Beispiel sind wir davon ausgegangen, dass man an einer Kreuzung stehend, alle Ausgänge sehen kann. Aber was ist, wenn die Fackel ausgeht und man im Dunkeln steht? Auch dann kann man noch zum Ziel finden; wie das geht, ist in Kap. 8 beschrieben.
4. Kapitel 34 (Kürzeste Wege)

Die Breitensuche findet einen kürzesten Weg, wenn man die Anzahl durchlaufener Kanten als Maßstab nimmt. Oft sind aber die Abstände von Knoten unterschiedlich groß, und man möchte einen Weg, bei dem die Summe der Kantenlängen möglichst klein ist. Dieses Problem wird in Kap. 34 behandelt.

## Danksagung

Wir danken Martin Dietzfelbinger (Ilmenau) für seine vielen konstruktiven Verbesserungsvorschläge.

„Alles auf Erden lässt sich finden, wenn man nur zu suchen sich nicht verdrießen lässt."

Philemon von Syrakus (um 360 v. Chr. – 264 v. Chr.)

# 8

# Der Pledge-Algorithmus: Wie man im Dunkeln aus einem Labyrinth entkommt

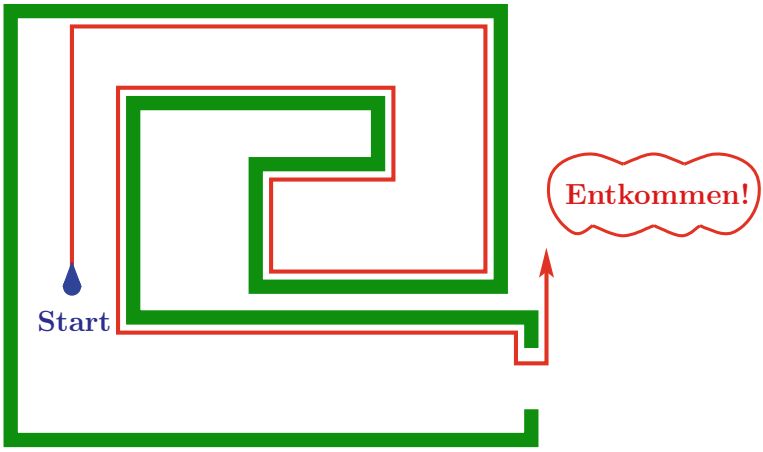
Rolf Klein und Tom Kamphans

Universität Bonn
Technische Universität Braunschweig

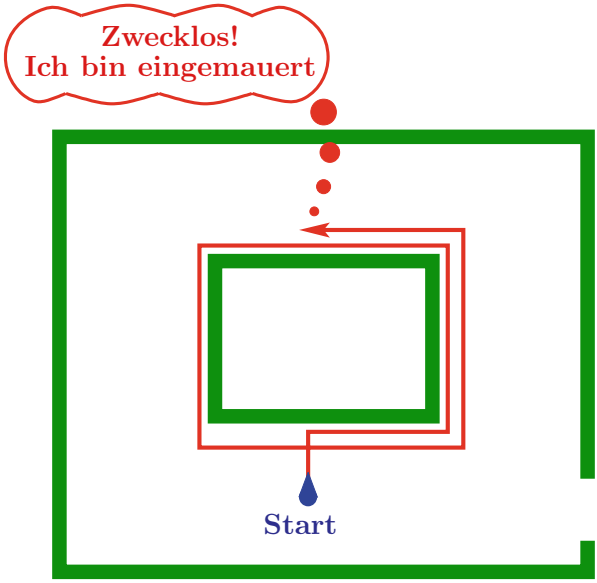
> "There must be some way out of here," said the joker to the thief,
> "There's too much confusion, I can't get no relief."
>
> Aus „ All Along the Watchtower" von Bob Dylan

„Oh weh – nun ist das Licht aus! Wie komme ich jetzt bloß hier heraus? War vielleicht doch keine gute Idee, allein in das Tunnelsystem unter den Kaiserthermen einzusteigen. Halt! Da haben wir doch neulich gelesen, wie man ein Labyrinth aus Kreuzungen und Gängen systematisch durchsucht: Tiefensuche. Aber dazu brauchte man Licht und Kreide für Markierungen. Geht also nicht, bei dieser Finsternis. Muss ich etwa die Nacht hier verbringen?"

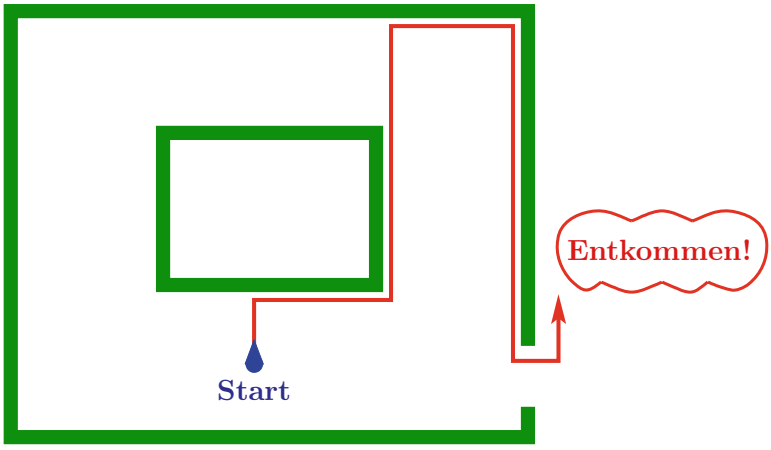
Was könnte man hier tun? Erst einmal vorsichtig der Nase nach gehen, bis man auf eine Wand trifft. Dann dreht man sich nach rechts, und geht mit der linken Hand immer an der Wand weiter. In diesem Labyrinth findet man so zum Ausgang:



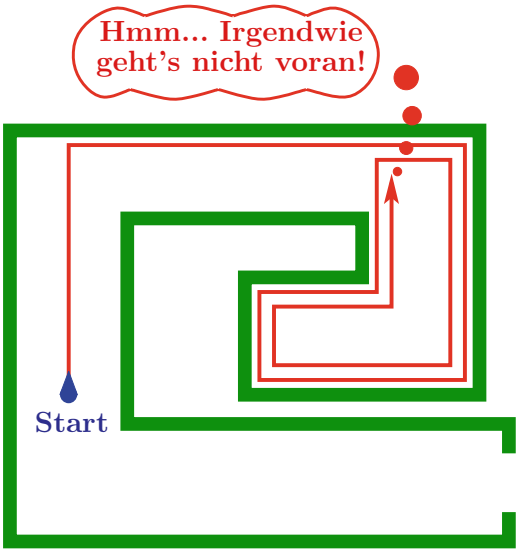
Aber wenn man auf eine Säule trifft, läuft man für immer im Kreis herum:




So geht es nicht! Man muss sich irgendwann wieder von solchen Säulen lösen. Also neuer Versuch: der Nase nach bis zur Wand, der Wand so lange folgen, bis man wieder in die alte Richtung läuft, und dann wieder der Nase nach. Jetzt macht die Säule keine Probleme mehr:



Aber dafür klappt's jetzt im ersten Beispiel nicht mehr:




„Langsam wird es mir unheimlich! Was immer ich versuche, geht schief. Aber es muss doch einen Weg nach draußen geben – schließlich bin ich auch hereingekommen!"

Natürlich gibt es einen Weg nach draußen. Die Frage ist nur, wie er aussieht. Kann man wirklich einen allgemeinen Algorithmus angeben, der für jedes denkbare ebene Labyrinth, aus dem es einen Ausweg gibt, einen Weg ins Freie findet? Und das auch im Dunkeln?

Erstaunlicherweise geht das tatsächlich. Es reicht aber nicht, nur auf die Richtung der Nase zu achten; man muss auch die Drehungen zählen, die unterwegs an den Ecken und beim Antreffen einer Wand ausgeführt werden.

Pledge-Algorithmus

```
1   Setze Umdrehungszähler auf 0;
2   repeat
3       repeat
4           Gehe geradeaus;
5       until Wand erreicht;
6       Drehe nach rechts;
7       repeat
8           Folge dem Hindernis;
9       until Umdrehungszähler = 0;
10  until ins Helle gelangt;
```

Angenommen, alle Ecken sind rechtwinklig, wie in unseren Beispielen. Dann kommen nur Rechtsdrehungen und Linksdrehungen um jeweils 90 Grad vor. Wir zählen die Vierteldrehungen, aber mit Richtungen: für eine Linksdrehung zählen wir um 1 hoch, für eine Rechtsdrehung um 1 herunter (auch bei der ersten Rechtsdrehung, die nach dem Auftreffen auf eine Wand ausgeführt wird).

Entdeckt haben soll diesen Algorithmus ein zwölfjähriger Junge namens John Pledge; deshalb nennt man ihn den Pledge-Algorithmus.

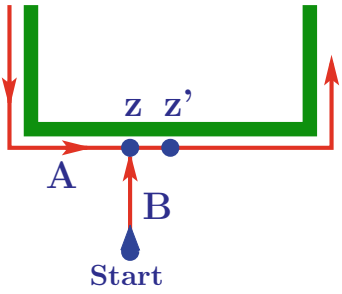
Er funktioniert nicht nur bei unseren beiden Beispielen, sondern in jedem Labyrinth mit rechtwinkligen Ecken! Und wir können das auch beweisen.

Dazu muss man sich Folgendes überlegen: Wenn man mit dem Pledge-Algorithmus aus einem Labyrinth nicht herausfindet, so durchläuft man einen Teil des Weges immer und immer wieder. Für eine eventuelle Änderung unserer Bewegungsrichtung kommen nämlich nur wenige Punkte in Frage: die Ecken von Wänden im Labyrinth und von jeder Wandecke aus der in Startrichtung erste Punkt auf dem nächsten Hindernis.

Falls wir einen solchen Eckpunkt zweimal mit demselben Zählerstand besuchen, wiederholen wir den Weg dazwischen zyklisch immer wieder, weil sich unser Verhalten nicht ändert, und es folgt die Behauptung.

Wenn aber jeder Eckpunkt höchstens einmal mit demselben Zählerstand besucht wird, erreichen wir nur endlich oft Eckpunkte mit Zählerstand null. Sobald diese Besuche erfolgt sind, können wir nie wieder von einem Hindernis abspringen; wir folgen also danach nur noch einem einzigen Hindernis. Unser Weg wird auch in diesem Fall zyklisch. □

Wir können weiter zeigen, dass sich der immer wieder besuchte Teil des Weges nicht selbst kreuzen kann. In einer Kreuzung würden zwei gerade Abschnitte des Weges, nennen wir sie $A$ und $B$, aufeinander treffen. Einer der beiden, sagen wir $B$, muss dabei *frei* sein, also nicht an einem Hindernis entlangführen, da sich Hinderniswände nicht schneiden.




Sei $z$ der Schnittpunkt von $A$ und $B$ und seien $W_A(z')$ und $W_B(z')$ die Zählerstände an einem Punkt $z'$ kurz hinter $z$ bei Anreise über $A$ bzw. über $B$. Dann ist

$$W_B(z') = -1$$
$$W_A(z') = -1 + 4 \cdot k \quad \text{mit} \quad k \in \mathbb{Z},$$

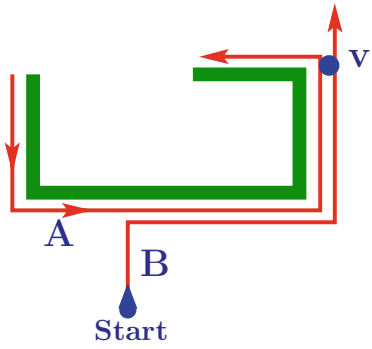
denn hinter $z$ zeigt unsere Nase stets in dieselbe Richtung. Wäre $k \geq 1$, so ergäbe sich

$$W_A(z') = -1 + 4k > 0.$$

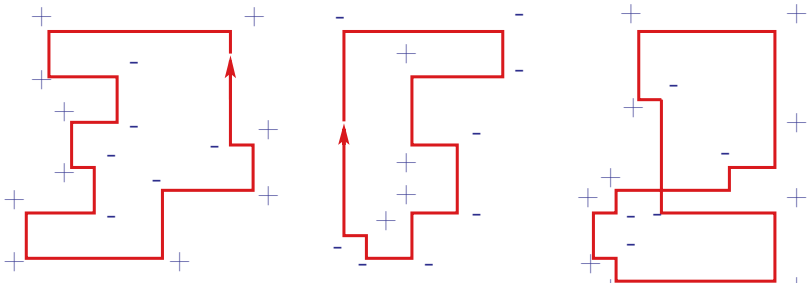
Unser Zählerstand kann aber nie positiv werden, denn nach dem Einschwenken zu einer Wand ist er zunächst bei $-1$. Sobald er den Wert 0 erreicht, verlässt man die Wand ja schon wieder, und beim Auftreffen auf die nächste Wand wird der Zähler wieder negativ. Also ist $k \leq 0$.

Aus $k = 0$ würde $W_A(z') = W_B(z')$ folgen. In diesem Fall würden sich die Wegstücke über $A$ und $B$ hinter $z$ niemals wieder trennen. Wenn also nach $z$ als nächstes etwa das Segment $B$ besucht würde, käme das Segment $A$ niemals wieder an die Reihe. Das steht im Widerspruch zu der Tatsache, dass sowohl $A$ als auch $B$ Teile der Endlosschleife sind!

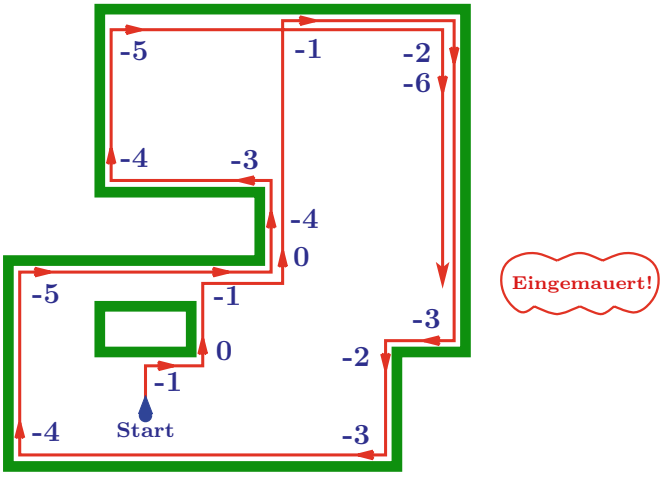
Also bleibt nur der Fall $k \leq -1$ übrig. Dann ist $W_A(t) < W_B(t)$ für alle Punkte $t$ ab $z'$ bis zu dem Punkt $v$, an dem sich die Wege wieder trennen; dort muss dann $W_B(v) = 0$ sein. Also sieht die Fortsetzung der Wegstücke im Prinzip so aus wie die folgende Abbildung zeigt. Es liegt daher keine echte Kreuzung vor, sondern nur eine Berührung. □



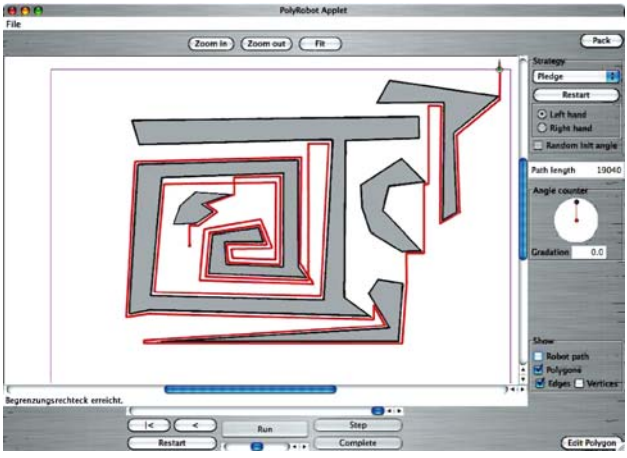

Wir wissen nun: Wenn man mit dem Pledge-Algorithmus aus einem Labyrinth nicht herausfindet, so durchläuft man einen Teil des Weges immer und immer wieder, und dieser Teil des Weges kann keine Kreuzungen enthalten.



Würde diese endlose Schleife gegen den Uhrzeigersinn durchlaufen, enthielte sie vier Linksdrehungen mehr als Rechtsdrehungen. Dann würde der Umdrehungszähler irgendwann positive Werte annehmen, und wir hatten schon gesehen, dass dieser Fall nicht eintreten kann. Folglich wird die Endlosschleife im Uhrzeigersinn durchlaufen. Bei jedem Durchlauf nimmt dann der Zähler um 4 ab, so dass er irgendwann nur noch negative Werte hat. Also folgt man mit der linken Hand ständig einer Wand, ohne sich je von ihr zu lösen. Dann gibt es aber auch gar keinen Weg nach draußen!




Wer mag, kann sich mit dem Java Applet `http://www.geometrylab.de/Pledge/` selbst ein Labyrinth zeichnen und verfolgen, wie der Pledge-Algorithmus vorgeht. Er funktioniert übrigens auch, wenn die Ecken nicht rechtwinklig sind. Nur muss man dann statt des Umdrehungszählers einen Winkelzähler verwenden, in dem die Winkel mit Vorzeichen exakt aufaddiert werden. Hier ein Beispiel, das mit Hilfe des Java Applets erstellt wurde:



## Zum Weiterlesen

1. Kapitel 7 (Tiefensuche)

   Hier wird beschrieben, wie man zur Verfügung stehende Hilfsmittel wie einen Faden oder Kreide ausnutzen kann.

2. Kapitel 9 (Zyklensuche in Graphen)

   hilft zu verhindern, dass man im Kreis läuft.
   Die „Knoten“, von denen in Kap. 9 gesprochen wird, sind einem Labyrinth die Räume mit mehreren Ausgängen. Die „Kanten“ sind Gänge und Türen, die je zwei Räume verbinden.

3. `http://www.geometrylab.de/Pledge/`

   Hier gibt es ein Java-Applet, mit dem man den Pledge-Algorithmus selbst ausprobieren kann, und außerdem einen kurzen Film, in dem ein Miniaturroboter des Typs Khepera II den Pledge-Algorithmus anwendet, um aus einem Labyrinth herauszufinden.

4. Rolf Klein: *Algorithmische Geometrie — Grundlagen, Methoden, Anwendungen*. Springer, Heidelberg, 2. Auflage, 2005.

   Harold Abelson, Andrea A. diSessa: *Turtle Geometry*. MIT Press, Cambridge, 1980. (engl.)

   In diesen Büchern wird gezeigt, dass der Pledge-Algorithmus auch den Ausgang aus nicht-rechtwinkligen Labyrinthen findet. Darüber hinaus behandeln beide Bücher viele andere Fragestellungen, z. B. wie ein Roboter einen Zielpunkt finden kann.

5. Bernd Brüggemann, Tom Kamphans, Elmar Langetepe: *Leaving an unknown maze with one-way roads*. In: *Abstracts 23rd European Workshop Comput. Geom.*, 2007, S. 90–93. (engl.)
   `http://web.informatik.uni-bonn.de/I/publications/bkl-lumow-07.eps`

   Leider funktioniert der Pledge-Algorithmus nicht mehr ohne weiteres, wenn es Passagen gibt, die man nur in einer Richtung durchqueren kann, wie z. B. Einbahnstraßen. In diesem Artikel wird erklärt, wie man dennoch entkommen kann.

## Danksagung

Die Autoren danken Martin Dietzfelbinger für wertvolle Hinweise.

# 9

# Zyklensuche in Graphen

Holger Schlingloff

Humboldt-Universität zu Berlin

In diesem Kapitel geht es darum, *Zyklen* in *Graphen* zu suchen. Das bedeutet, wir wollen feststellen, ob in einer Menge von Knoten, die durch Kanten miteinander verbunden sind, ein Zyklus enthalten ist. Ein *Zyklus* oder Kreis ist dabei ein Weg von einem Knoten zu sich selbst.

### Szenario 1

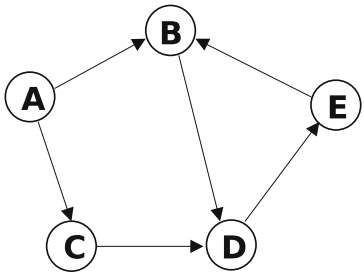
Du bist mit dem Flugzeug mitten in einem Urwald abgestürzt und suchst einen Weg zurück in die Zivilisation. Durch den Dschungel führen etliche Trampelpfade, die von Einheimischen angelegt wurden; außerhalb der Wege ist das Dickicht undurchdringlich. Durch das dichte Blätterdach kannst du nicht einmal die Sonne sehen. Du packst also die verfügbare Ausrüstung zusammen und marschierst in die erstbeste Richtung los. Schon nach wenigen Minuten kommst du an eine Weggabelung; du entschließt dich, den rechten Weg weiterzulaufen. Dann kommt eine Kreuzung, an der du geradeaus gehst.

Leider ist das ein Holzweg, eine Sackgasse, die nicht weiter führt. Also kehrst du um, gehst zur Kreuzung zurück und wendest dich nach rechts. Bei der nächsten Abzweigung gehst du nach links, dann wieder nach rechts und so weiter. Auf einmal lichtet sich der Urwald und du stehst – wieder vor deinem Flugzeug, von dem du losgelaufen bist: offenbar bist du die ganze Zeit im Kreis gelaufen. Wie kannst du verhindern, dass dir so etwas beim nächsten Versuch wieder passiert?

**Szenario 2**

Andy möchte mit Benny und Charly ins Kino gehen. Charly muss aber zu Hause Babysitten und kann nur dann weg, wenn Dany kommt und ihn ablöst. Benny darf erst los, wenn er seine Hausaufgaben erledigt hat. Dazu braucht auch er die Hilfe von Dany, die versprochen hat, zu ihm zu kommen, sobald sie das Aufgabenheft von Eddy zurückbekommt, das sie ihm in der Schule geliehen hatte. Eddy grübelt aber noch an den Aufgaben und hofft, dass Benny ihm eine Mail mit der Lösungsidee schickt. Warum wird Andy den Film wahrscheinlich verpassen?

Beide Szenarien können auf dasselbe Problem zurückgeführt werden, nämlich die Zyklensuche in Graphen. Ein (gerichteter) *Graph* ist eine Struktur bestehend aus *Knoten* und *Kanten*, wobei eine Kante jeweils von einem Knoten zu einem anderen Knoten führt. Knoten werden als Kreise gezeichnet und Kanten als Pfeile zwischen zwei Knoten. Zum Beispiel könnten wir im zweiten Szenario für jede Person einen Knoten zeichnen und eine Kante vom Knoten $x$ zum Knoten $y$ eintragen, wenn Person $x$ auf Person $y$ wartet. Das ergibt dann den folgenden Graphen:




Wie man sofort sieht, gibt es hier einen Zyklus B → D → E → B, d. h. eine Folge von Knoten, die durch Kanten verbunden sind, und der Anfang und das Ende der Folge sind derselbe Knoten. Benny wartet auf Dany, Dany wartet auf Eddy, und Eddy wartet auf Benny: Wenn sie nichts dagegen tun, werden sie so ewig warten. Solche Zyklen können leicht dazu führen, dass bestimmte Prozesse nie aufhören (wie im ersten Beispiel) oder gar nicht weiterkommen (wie im zweiten Beispiel). Wenn das bei einem Computer passiert, spricht man von einer Endlosschleife oder von einer Verklemmung. Beides führt meist dazu, dass das Programm keine Reaktion mehr zeigt und man es von außen abbrechen muss. Daher ist es oft eine wichtige Aufgabe, Zyklen zu erkennen (und nach Möglichkeit zu vermeiden).

## Zyklensuche mittels Tiefensuche

Wie geht man jetzt zur Zyklensuche praktisch vor? Kommen wir zum Urwaldbeispiel zurück. Wenn du einen Ausweg aus dem Dschungel suchst, kannst du

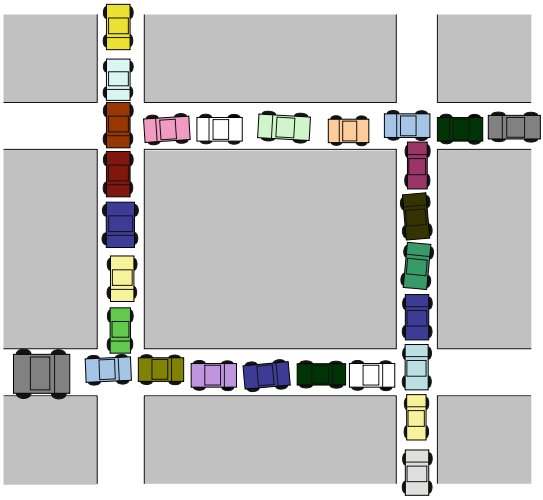


**Abb. 9.1.** Verklemmung im Straßenverkehr

wie einst Hänsel und Gretel den Weg mit Steinchen markieren. Falls du dann beim Laufen zu einem vorher hingelegten Steinchen kommst, weißt du, dass du schon einmal da gewesen sein musst und dass dich das letzte Wegstück für die Suche nach einem Ausweg nicht weiter bringt. Das ist also im Prinzip dasselbe Problem wie beim Entkommen aus einem Labyrinth, wo der zurückgelegte Weg mit einem Ariadnefaden oder mit Kreide markiert wird (siehe Kap. 7). Um die Urwaldexpedition durch ein Computerprogramm zu simulieren, können wir daher wie in Kap. 7 eine Tiefensuche verwenden. Wir modellieren zunächst die Urwald-Landkarte durch einen Graphen: Jede Wegkreuzung oder -gabelung ist dann ein Knoten des Graphen, und jedes Wegstück zwischen zwei Verzweigungen ist eine Kante. Ziel der Suche ist ein Knoten außerhalb des Urwalds. Den Tiefensuche-Algorithmus können wir ähnlich wie in Kap. 7 formulieren:

Tiefensuche

```
1  procedure TIEFENSUCHE (Knoten x)
2  begin
3     if Ziel erreicht then stop
4     else if x unmarkiert then
5        markiere x;
6        for all Nachfolgerknoten y von x do TIEFENSUCHE(y) endfor
7     endif
8  end
```

Dabei nehmen wir an, dass anfänglich alle Knoten „unmarkiert“ sind. Die Tiefensuche wird ausgelöst, indem man für irgendeinen Startknoten $x_1$ die

Prozedur TIEFENSUCHE($x_1$) aufruft. Falls $x_1$ die Nachfolger $y_1$, $y_2$ usw. hat, wird dann der Reihe nach TIEFENSUCHE($y_1$), TIEFENSUCHE($y_2$) usw. aufgerufen. Falls dabei ein $y_i$ die Nachfolger $z_1$, $z_2$ usw. hat, dann startet der Aufruf TIEFENSUCHE($y_i$) zuerst die Tiefensuche für alle diese $z$, bevor mit $y_{i+1}$ weitergemacht wird. Falls bei der Suche ein Knoten erreicht wird, der keine Nachfolger hat (eine Sackgasse) oder der schon bereits markiert ist, so wird die Suche nicht fortgesetzt, sondern zum vorherigen Knoten zurückgekehrt.

Abbildung 9.2 verdeutlicht den Ablauf des Algorithmus für den Graphen des zweiten Beispiels. Die Knoten sind dabei von oben nach unten gezeichnet, die Nummern geben die Reihenfolge an, in der die Knoten erreicht und markiert werden. Die Suche startet beim Knoten A, d. h. mit dem Aufruf TIEFENSUCHE(A). Da A noch nicht markiert ist, werden nacheinander TIEFENSUCHE(B) und TIEFENSUCHE(C) aufgerufen. Die Bearbeitung des Aufrufs TIEFENSUCHE(C)wird dabei zurückgestellt, bis die Ausführung von TIEFENSUCHE(B) erledigt ist. TIEFENSUCHE(B) ruft also zunächst TIEFENSUCHE(D), dieses wiederum TIEFENSUCHE(E), und dieses TIEFENSUCHE(B). B ist allerdings schon markiert, also wird zu E zurückgekehrt. Hier sind schon alle Nachfolgerknoten verarbeitet, also erfolgt die Rückkehr zu D, dann zu B und A. Jetzt kann der zurückgestellte Aufruf von TIEFENSUCHE(C) bearbeitet werden. Da D bereits markiert ist, erfolgt die Rückkehr zu C und A. Jetzt sind alle Aufrufe bearbeitet und die Ausführung des Algorithmus wird beendet.

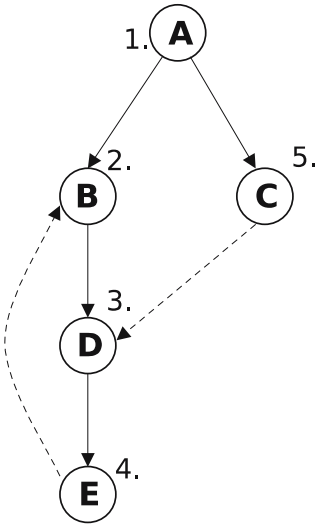



**Abb. 9.2.** Tiefensuche

Bei der Zyklensuche geht es nicht darum, den (erstbesten) Ausweg zu finden, sondern festzustellen, ob der Graph Zyklen enthält und diese gegebenenfalls auszugeben. Für die Zyklensuche muss der Algorithmus also ein wenig modifiziert werden. Wie wir im obigen Beispiel sehen, gibt es drei Arten von Kanten:

1. Vorwärtskanten, z. B. von A nach C
2. Querkanten, z. B. von C nach D, und
3. Rückwärtskanten, z. B. von E nach B.

In einem gerichteten Graphen kann nur eine Rückwärtskante einen Zyklus verursachen. Rückwärtskanten können von Querkanten dadurch unterschieden werden, dass sie auf einen Knoten verweisen, der bisher noch nicht vollständig abgearbeitet wurde. Wir können das berücksichtigen, indem wir die Markierung erweitern: Statt einfach nur „unmarkiert“ oder „markiert“ merken wir uns in der Markierung jedes Knotens, ob die Bearbeitung noch nicht

begonnen hat, der Knoten sich in Bearbeitung befindet oder die Bearbeitung abgeschlossen ist.

Zyklensuche

```
1  procedure Zyklensuche (Knoten x)
2  begin
3     if Markierung(x) = „in Bearbeitung“ then Zyklus gefunden
4     else if Markierung(x) = „noch nicht begonnen“ then
5        Markierung(x) := „in Bearbeitung“;
6        for all Nachfolgerknoten y von x do Zyklensuche(y) endfor;
7        Markierung(x) := „abgeschlossen“
8     endif
9  end
```

Für den Beispielgraphen ergibt sich die nachfolgende Aufrufreihenfolge.

```
Zyklensuche (A) // A noch nicht begonnen
| A in Bearbeitung
| Zyklensuche (B) // B noch nicht begonnen
| | B in Bearbeitung
| | Zyklensuche (D) // D noch nicht begonnen
| | | D in Bearbeitung
| | | Zyklensuche (E) // E noch nicht begonnen
| | | | E in Bearbeitung
| | | | Zyklensuche (B) // B in Bearbeitung
| | | | | Zyklus gefunden!
| | | | E abgeschlossen // Zeitpunkt der Momentaufnahme
| | | D abgeschlossen
| | B abgeschlossen
| Zyklensuche (C) // C noch nicht begonnen
| | C in Bearbeitung
| | Zyklensuche (D) // D abgeschlossen
| | C abgeschlossen
| A abgeschlossen
```

## Zusammenhangskomponenten

Der oben angegebene Algorithmus „Tiefensuche“ stellt fest, ob vom Ausgangsknoten aus ein Zyklus erreicht werden kann. Man kann aber daran nicht erkennen, welche Knoten tatsächlich auf dem Zyklus liegen. Der Algorithmus ist also nicht geeignet, um die Verklemmung im zweiten Szenario aufzulösen. Um das zu tun, muss man sich den zurückgelegten Weg, d. h. die Folge der Knoten „in Bearbeitung“ merken. Bei der oben angegebenen Momentaufnahme ist der aktuelle Weg A → B → D → E → B, d. h., B, D und E liegen auf dem Zyklus. Wenn man also auf einen Knoten stößt, bei dem man schon

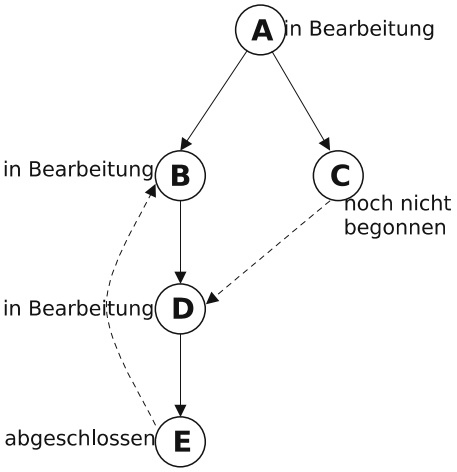



**Abb. 9.3.** Momentaufnahme bei der Ausführung der Prozedur ZYKLENSUCHE

einmal war, so sind alle Knoten, die danach besucht wurden, in dem Zyklus enthalten. Falls die Suche bei einem Knoten abgeschlossen ist, man also zum darüber liegenden Knoten zurückkehrt, muss dieser Knoten natürlich auch aus dem aktuellen Pfad entfernt werden. Algorithmisch könnte das in etwa so aussehen:

Zyklenfinden

```
1  procedure Zyklenfinden (Knoten x)
2  begin
3      if Markierung(x) = „in Bearbeitung“ then
4          Zyklus gefunden;
5          alle Knoten auf dem aktuellen Pfad ab x liegen auf dem Zyklus
6      else if Markierung(x) = „noch nicht begonnen“ then
7          Markierung(x) := „in Bearbeitung“;
8          Verlängere den aktuellen Suchpfad um x;
9          for all Nachfolgerknoten y von x do Zyklenfinden(y) endfor;
10         Markierung(x) := „abgeschlossen“;
11         Entferne x (das letzte Element) aus dem aktuellen Pfad
12     endif
13 end
```

Hier nehmen wir an, dass anfänglich alle Knoten als „noch nicht begonnen“ markiert sind und der anfängliche Suchpfad leer ist.

Was würde in diesem Algorithmus passieren, wenn der Graph mehrere Zyklen enthielte, im Beispiel etwa noch eine zusätzliche Kante von D nach A? Nun, wir erhalten zuerst den Zyklus B → D → E → B und dann den Zyklus A → B → D → A. Es gibt natürlich in diesem Fall noch weitere Zyklen, z. B. E → B → D → A → B → D → E. Wir sagen, dass zwei Knoten mitein-

ander *zusammenhängen*, wenn sie auf einem gemeinsamen Zyklus liegen. Mit anderen Worten, Knoten A hängt mit Knoten E zusammen, wenn es irgend einen Zyklus $A \to \cdots \to E \to \cdots \to A$ gibt. Alle Knoten, die miteinander zusammenhängen, liegen in derselben *Zusammenhangskomponente* (englisch: *Strongly Connected Component, SCC*).

In gewissem Sinne können alle Knoten in einer Zusammenhangskomponente miteinander identifiziert werden: Wenn A und E miteinander zusammenhängen und es gibt einen Weg von A nach C, dann gibt es auch einen Weg von E nach C. Der Graph, der dadurch entsteht, dass alle Knoten einer Zusammenhangskomponente zusammengefasst werden, nennt man den *Quotientengraphen*. Dieser enthält dann natürlich keine Zyklen mehr!

Robert E. Tarjan hat den obigen Algorithmus ZYKLENFINDEN so ausgearbeitet, dass nicht nur die Zyklen, sondern auch die vom Startknoten aus erreichbaren Zusammenhangskomponenten damit gefunden werden können. Dabei werden jedem Knoten zwei Zahlen zugeordnet: zum einen die Nummer

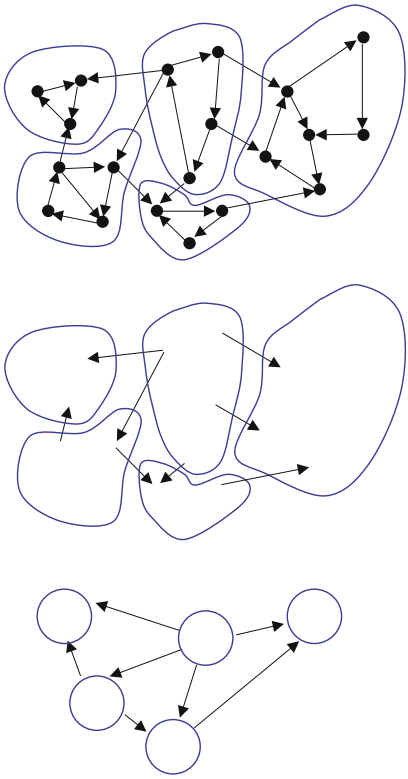


**Abb. 9.4.** Zusammenhangskomponenten und Quotientengraph

der Reihenfolge, in der er bei der Tiefensuche auftaucht, und zum anderen die Nummer des ersten Knotens der betreffenden Zusammenhangskomponente.

Zusammenhangskomponenten

```
 1  procedure KOMPONENTENFINDEN (Knoten x)
 2  begin
 3     if Markierung(x) = „in Bearbeitung“ then Zyklus gefunden
 4     else if Markierung(x) = „noch nicht begonnen“ then
 5        Markierung(x) := „in Bearbeitung“;
 6        Tiefensuchnummer(x) := Zähler;
 7        Komponentennummer(x) := Zähler;
 8        Zähler := Zähler +1;
 9        for all Nachfolgerknoten y von x do
10           if Markierung(y) nicht „abgeschlossen“ then
11              KOMPONENTENFINDEN(y);
12              if Komponentennummer(y) < Komponentennummer(x)
                then
13                 Komponentennummer(x) := Komponentennummer(y)
14              endif
15           endif
16        endfor;
17        Markierung(x) := „abgeschlossen“;
18        if Tiefensuchnummer(x) = Komponentennummer(x) then
19           Zusammenhangskomponente gefunden;
20           Knoten mit gleicher Nummer liegen in derselben Komponente
21        endif
22     endif
23  end
```

Der Zähler ist mit irgendeinem festen Wert (z. B. 1) initialisiert. Ein erweitertes Beispiel für die Markierungen der Knoten nach Ablauf von KOMPONENTENFINDEN ist in Abb. 9.5 angegeben. Der sich ergebende Quotientengraph enthält die Komponenten 1, 2 und 5, wobei Kanten von 1 nach 2 und von 1 nach 5 sowie von 5 nach 2 führen.

## Zyklensuche mittels Breitensuche

Wie wir gesehen haben, ist die Tiefensuche gut geeignet, um alle Zyklen oder Zusammenhangskomponenten in einem Graphen zu finden. Falls es lediglich darum geht, festzustellen, ob ein gegebener Ausgangsknoten in einem Zyklus liegt, so können wir einen einfacheren Algorithmus verwenden: Wir können mittels einer so genannten *Breitensuche* die Menge der vom Ausgangsknoten erreichbaren Knoten berechnen. Dazu gehen wir davon aus, dass wir eine effiziente Methode haben, mit der wir für eine Menge von Knoten die Nachfolgermenge berechnen können, d. h., die Menge der Knoten, die durch eine Kante mit der ursprünglichen verbunden sind. Und dann starten wir einfach mit der

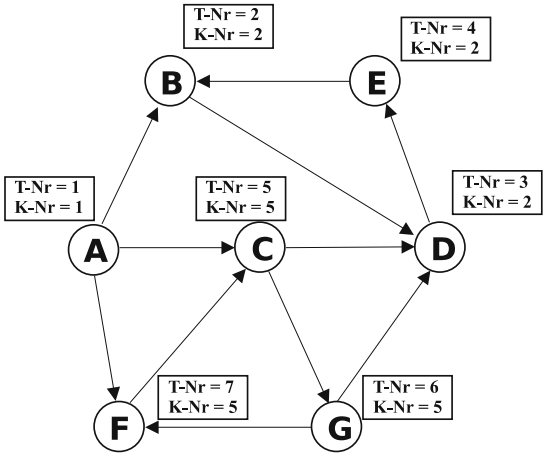


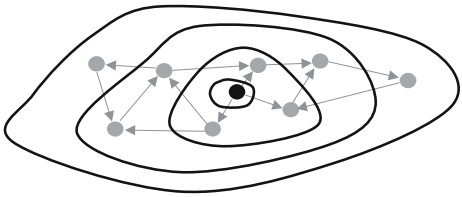

**Abb. 9.5.** Markierungen der Knoten nach Ablauf der Prozedur KOMPONENTEN-FINDEN

Menge, die nur den Ausgangsknoten enthält, berechnen die Nachfolgermenge, die Nachfolger-Nachfolgermenge, davon wieder die Nachfolgermenge usw., bis wir irgendwann entweder keinen neuen Knoten hinzubekommen oder den Ausgangsknoten wieder erreichen.

Breitensuche

```
1   procedure BREITENSUCHE (Knoten x)
2   begin
3       Erreichbar := {}; Front := {x};
4       repeat
5           Front := {y  |  y ist Nachfolgerknoten irgendeines z aus Front
            und y ist nicht in Erreichbar }
6           if x aus Front then Zyklus von x nach x existiert, stop;
7           endif;
8           Erreichbar := Erreichbar vereinigt mit Front;
9       until Front = {}
10  end
```

Man kann sich eine Breitensuche in etwa wie die Ausbreitung von Wellen vorstellen, die von einem Stein ausgehen, der in einen ruhigen See geworfen wird. Die Wellenfront, also die vorderste Welle, umfasst jeweils den erreichbaren Teil der Wasseroberfläche.



Die im Breitensuche-Algorithmus verwendete Schleife wird höchstens so oft wiederholt, wie es Knoten im Graphen gibt, üblicherweise jedoch viel weniger oft. (Genau genommen wird die Laufzeit durch den längsten Pfad von $x$ zu irgendeinem anderen Knoten bestimmt.) Andererseits müssen bei der Breitensuche die beiden Mengen „Front“ und „Erreichbar“ angelegt und verwaltet werden, die (im Vergleich zur Tiefensuche) unter Umständen sehr viele Knoten enthalten können. Die Komplexität hängt daher sehr wesentlich von der Effizienz der verwendeten Mengenoperationen ab. Bei einigen Programmiersprachen gibt es sehr schnelle Bibliotheksfunktionen für große Mengen und Relationen, so dass die Breitensuche effizient implementiert werden kann.

## Historische Notizen

Das Problem, Zyklen in Graphen zu suchen, ergab sich schon sehr früh in der Geschichte der Informatik. Erste Anwendungsbeispiele in den 1950-ern waren die Suche nach Schleifen in Schaltkreisen oder Datenflussdiagrammen. Die Tiefensuche und damit verbunden die rekursive Zyklensuche ist seit den 1960-ern bekannt und wird oft als Standardbeispiel für Backtracking-Algorithmen verwendet. Tarjans Algorithmus zur Berechnung von starken Zusammenhangskomponenten erschien 1972. Eine wesentliche Anwendung von Algorithmen zur Zyklensuche besteht in der Erkennung von Verklemmungen in Betriebsmittelgraphen: In jedem Mehrprozess-Betriebssystem können bei mangelnder Synchronisation zyklische Wartebedingungen auftreten. Bekannte Veranschaulichungen dafür sind Dijkstras Problem der dinierenden Philosophen oder Lamports Problem der Bäckereiwarteschlangen. Seit den 1970-ern tauchen immer wieder Computerspiele (z. B. „dungeons and dragons“) auf, in denen der Spieler durch ein virtuelles Labyrinth (einen Graphen) wandert, in dem diverse Gefahren zu bestehen sind. In den 1990-ern wurden neue effiziente Algorithmen und Datenstrukturen zur Zyklenerkennung und Quotientenbildung im Rahmen von Zustandsraum-Suchverfahren bei der automatischen Verifikation von Modellen entwickelt.

## Zum Weiterlesen

1. H. Peter Gumm, Manfred Sommer: *Einführung in die Informatik.* Oldenbourg, 7. Auflage, pp. 381 ff, 2006.

Ein sehr empfehlenswertes Lehrbuch für die letzten Schul- und ersten Universitätsjahre, das ständig aktualisiert wird.

2. Robert Sedgewick: *Algorithmen*. Pearson, 2. Auflage, Kap. 29, 2002.

   Der „Klassiker“ unter den Algorithmen-Lehrbüchern. Es gibt das Buch in verschiedenen Versionen für die unterschiedlichen Programmiersprachen.

3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*. MIT Press, 2001.

   Ein im angelsächsischen Raum weit verbreitetes Lehrbuch zum Thema.

4. Robert E. Tarjan: *Depth-first search and linear graph algorithms*. In: *SIAM Journal on Computing* 1(2), pp. 146–160, 1972.

   Die Original-Referenz zum Algorithmus für die Berechnung von Zusammenhangskomponenten.

5. Edsgar W. Dijkstra: *Hierarchical ordering of sequential processes*. Acta Informatica 1 (2), pp. 115–138, 1971.
   `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF`

   Dieser Artikel beschreibt das Problem der Verklemmung von Prozessen anhand von Philosophen, die seit 37 Jahren vor einer Schüssel Spaghetti sitzen.

6. Wikipedia zum Thema:
   - Tiefensuche:
     `http://de.wikipedia.org/wiki/Tiefensuche`,
     `http://en.wikipedia.org/wiki/Depth-first_search`
   - Breitensuche:
     `http://de.wikipedia.org/wiki/Breitensuche`,
     `http://en.wikipedia.org/wiki/Breadth-first_search`
   - Algorithmus von Tarjan:
     `http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm`

# 10

# PageRank: Was ist wichtig im World Wide Web?

Ulrik Brandes und Gabi Dorfmüller

Universität Konstanz

Die zweifellos populärste Form der Nutzung des Internets ist das *World Wide Web* (WWW), ein Netzwerk aus Milliarden von Dateien. Es enthält vor allem *Web-Seiten* aus Text- und Bilddateien, die durch *Links* (Verweise) miteinander verknüpft sind. Selbst wenn man sich das ganze Leben Zeit nähme, Tag und Nacht nichts anderes täte, als Web-Seiten anzuschauen, und keine Seite länger als eine Sekunde betrachtete, bekäme man trotzdem nur einen kleinen Bruchteil aller Seiten je zu sehen.[1] Wenn man im WWW etwas finden möchte, muss man daher wissen, wo es steht, oder zumindest, über welche Links man dorthin kommt.

Praktisch alle Surfer(innen) im WWW benutzen deshalb Suchmaschinen, d. h. Web-Seiten, auf denen man das Gesuchte mit ein paar Stichwörtern beschreibt (Anfrage) und als Antwort eine Liste von Web-Seiten erhält, die damit zu tun haben könnten (Treffer). Mit Hilfe von vielen Informatik-Methoden sind moderne Suchmaschinen in der Lage, Milliarden von Seiten zu verwalten und in Sekundenbruchteilen diejenigen Seiten herauszufischen, auf denen die Suchbegriffe vorkommen.

Da man aber sogar auf einen Suchbegriff wie `Algorithmus` Millionen von Treffern erhält, ist selbst die Antwort zu umfangreich, um ganz gelesen zu werden. Suchmaschinen ordnen die Liste deswegen so, dass diejenigen Seiten zuerst angezeigt werden, die als besonders relevant angesehen werden.

Frage:

Wie schaffen es Suchmaschinen, unter Millionen von Web-Seiten solche herauszufinden, die (oft) auch uns als relevant erscheinen?

Die derzeit wohl bekannteste Suchmaschine wird von der Firma Google betrieben,[2] und einer der Gründe dafür ist, dass Google Ende der 1990er Jahre als erste Suchmaschine nicht nur einen großen Fundus von Seiten durchsucht,

[1] Wieviele Sekunden hat ein normales Menschenleben?

[2] `www.google.de`

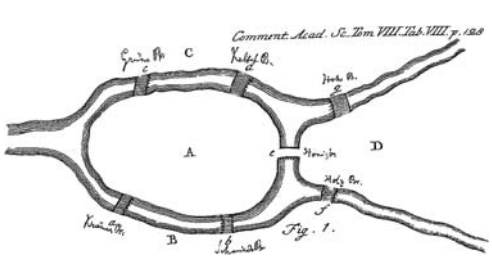
sondern auch einen besonders cleveren Algorithmus zur Reihung der Ergebnisse verwendet hat. Beispielsweise rangiert die Web-Seite zum „Algorithmus der Woche“ aus dem Informatikjahr 2006 derzeit auf Platz 2 der zweieinhalb Millionen Treffer zur Suchanfrage `Algorithmus` (hinter dem Wikipedia-Artikel zu diesem Begriff).

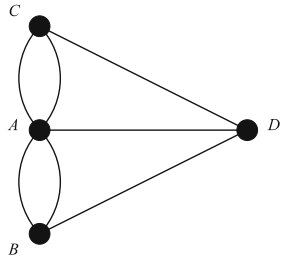
Neben vielen einfachen Kriterien wie z. B. der Position der Suchbegriffe auf einer Seite (in Überschriften? nah beieinander?) und vielen unbekannten Regeln ist ein zentraler Teil des Algorithmus die Auswertung der Links. Dieser Baustein wird als PageRank bezeichnet und in diesem Kapitel erläutert.

## Touristische Trampelpfade

Erklärungen von PageRank verwenden häufig als Motivation, dass eine Seite umso relevanter werden soll, je öfter man bei zielloser Suche im WWW auf diese Seite gelangt. Diese Idee wollen wir uns zunächst an einem völlig anderen Beispiel näher ansehen.

Wenn im 18. Jahrhundert der Mathematiker Leonhard Euler nicht die Unmöglichkeit bewiesen, sondern nach langem Suchen den in Kap. 29 erwähnten Rundgang (also eine so genannte Euler-Tour) über die sieben Brücken von Königsberg gefunden hätte, dann wäre diese Tour jetzt sicher sehr berühmt, würde in allen Reiseführern stehen und ständig abgelaufen werden. Wer Souvenirs oder Erfrischungen verkaufen möchte, sollte sich dann am besten dort aufstellen, wo die Touristen am häufigsten vorbeikommen.




Weil das bei einem Rundgang egal ist, können wir nicht wissen, wo er begonnen wird, aber da man beim Ablauf jede Brücke genau einmal überquert, ist zumindest klar, dass jeder Knotenpunkt gerade halb so oft besucht wird, wie Brücken an ihn anschließen (jeweils eine Brücke wird fürs Hinkommen benötigt und eine fürs Weitergehen). Die attraktivsten Verkaufsstandorte wären daher dort, wo die meisten Brücken angrenzen; in Königsberg also auf dem Kneiphof (Standort $A$).

Nun gibt es aber ja keinen solchen Rundgang. Nehmen wir daher an, die Touristen irrten ziellos durch die Stadt, und liefen von einer Brücke zufällig, d. h. gleich häufig, zu jeder möglichen (einschließlich der, über die sie gekommen sind). Wie oft kommen sie dann an einen Knotenpunkt?

Die Anzahl $b$ der Besuche bei Knotenpunkt $B$ können wir dadurch beschreiben, dass man vorher bei einem anderen Knotenpunkt gewesen sein muss, der durch eine Brücke mit $B$ verbunden ist, in unserem Fall also bei $A$ oder $D$. Wenn von $A$ aus jede Brücke gleich oft zum Weiterlaufen gewählt wird, dann kommt man in zwei von fünf Fällen von $A$ nach $B$ und entsprechend in einem von drei Fällen von $D$ nach $B$. Das unbekannte $b$ kann also in den analogen, aber ebenfalls unbekannten Anzahlen $a$ und $d$ ausgedrückt werden:

$$b = \frac{2}{5}a + \frac{1}{3}d \ .$$

Entsprechende Gleichungen können wir für alle Unbekannten aufstellen:

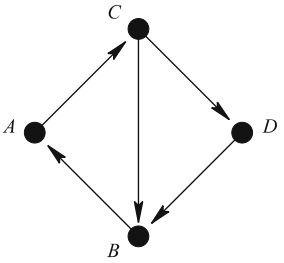
$$a = \frac{2}{3}b + \frac{2}{3}c + \frac{1}{3}d \ , \qquad c = \frac{1}{5}a + \frac{1}{3}d \ , \qquad d = \frac{1}{5}a + \frac{1}{3}b \ .$$

Interessanterweise sind alle Lösungen dieses Gleichungssystems von der Form $a = 5$ und $b = c = d = 3$ bzw. Vielfache davon; die Verhältnisse der Anzahlen sind damit die gleichen, die wir bei Existenz eines Rundgangs erhalten hätten. Ob Touristen systematisch oder ziellos durch Königsberg laufen, wäre für die Standortsuche von Händler(innen) also völlig egal. Mehr noch, es wäre in jeder anderen Stadt genauso, unabhängig von der Art, wie die Stadtteile verbunden sind.

## Trampelpfade im Netz

Interpretiert man einen Verweis im WWW als Empfehlung, auf der Suche nach weiterer Information die verlinkte Seite zu besuchen, können wir genau wie bei den Knotenpunkten fragen, bei welchen Seiten man am häufigsten landet, wenn auf jeder Seite alle Links gleich häufig verfolgt werden, das Web also ziellos durchlaufen wird. Ist dann die Relevanz einer Seite einfach die Anzahl der auf sie zeigenden Links?

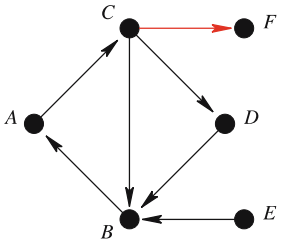
Im Unterschied zu den Brücken von Königsberg besteht das WWW aus lauter Einbahnstraßen, denn Links können nur in einer Richtung verfolgt werden (den „Zurück"-Knopf ignorieren wir zunächst). Das folgende Beispiel zeigt, dass die Situation dadurch erheblich komplizierter wird.




$$a = b$$
$$b = \frac{1}{2}c + d$$
$$c = a$$
$$d = \frac{1}{2}c$$

Die Begründung für die Aufstellung des Gleichungssystems der unbekannten Aufenthaltshäufigkeiten bleibt gültig, aber die Lösungen dieses Gleichungssystems sind Vielfache von $a = b = c = 2$ und $d = 1$. Der Zusammenhang mit der Anzahl von ein- und ausgehenden Verbindungen ist damit verloren gegangen (sonst müssten $a$ und $d$ gleich und von $b$ und $c$ verschieden sein).

In tatsächlich auftretenden Netzwerken kommt zu den Einbahnstraßen noch mindestens ein weiteres Problem hinzu, nämlich Sackgassen. In diesem Netzwerk




steckt man bei Verfolgung des roten Links fest. Solche einfachen Fälle sind zwar leicht zu finden (und werden von Google wohl auch herausgenommen), die Sackgassen können aber auch weniger offensichtlich sein, weil es in größeren Netzwerken möglich ist, dass man von einem Knoten wie $F$ noch weiterkommt, aber nicht wieder zurück zu den Knoten $A, \ldots, E$. Seiten, zu denen man irgendwann nicht mehr zurückkehren kann, führen zu Lösungen des Gleichungssystems, die für die Sortierung nach Relevanz ungeeignet sind (warum wohl?).

Das bisher nachgeahmte Surf-Verhalten entspricht aber ja sowieso nicht dem realen. Wer auf einer Seite keinen oder keinen interessanten Link findet, wird irgend eine andere Seite aufrufen (z. B. durch den „Zurück"-Knopf, Favoriten oder direkte Eingabe einer neuen Web-Adresse).

Nehmen wir solche unvermittelten Sprünge zu anderen Knoten in das Modell auf, wird das Gleichungssystem dadurch nur wenig komplizierter. Wir legen einfach fest, dass z. B. in einem von fünf Fällen nicht durch Verfolgung eines Links, sondern direkt zu einer Seite gesprungen und dass beim Springen keine Seite bevorzugt wird (d. h., bei den Sprüngen ist jede der sechs Seiten langfristig gleich oft das Ziel). Damit bleibt jede Seite zu jedem Zeitpunkt erreichbar, und wir kommen von jeder Seite auch wieder weg.

$$
\begin{aligned}
a &= \frac{4}{5} \cdot b + \frac{1}{5} \cdot \frac{1}{6} & \qquad d &= \frac{4}{5} \cdot \left( \frac{1}{3} \cdot c \right) + \frac{1}{5} \cdot \frac{1}{6} \\
b &= \frac{4}{5} \cdot \left( \frac{1}{3} \cdot c + 1 \cdot d + e \right) + \frac{1}{5} \cdot \frac{1}{6} & \qquad e &= \frac{4}{5} \cdot 0 + \frac{1}{5} \cdot \frac{1}{6} \\
c &= \frac{4}{5} \cdot a + \frac{1}{5} \cdot \frac{1}{6} & \qquad f &= \frac{4}{5} \cdot \left( \frac{1}{3} \cdot c \right) + \frac{1}{5} \cdot \frac{1}{6}
\end{aligned}
$$

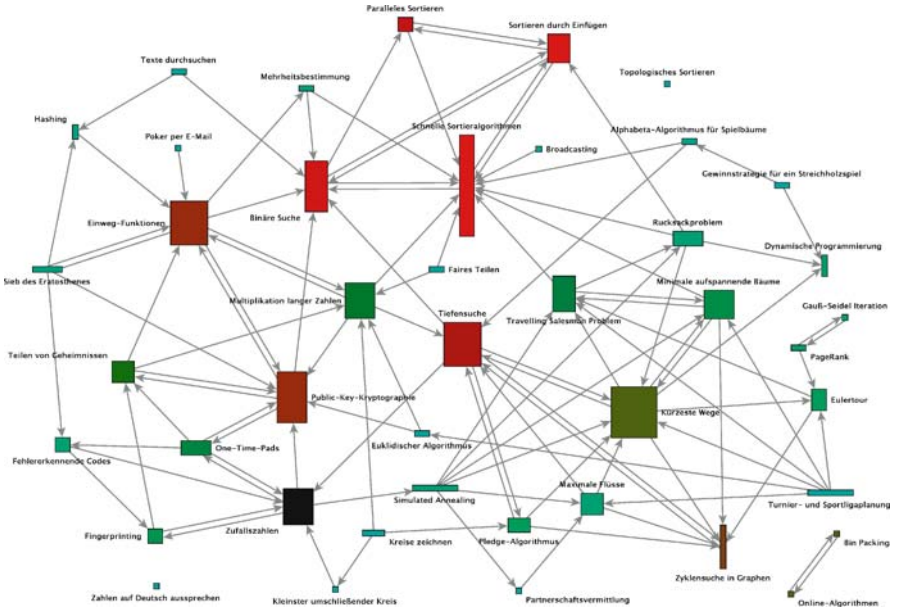
Dieses Gleichungssystem ist nur wenig umfangreicher und auch nicht schwieriger zu lösen als das System im vorigen Abschnitt. Ähneln die Lösungen dem Ergebnis des folgenden Experiments?

**Experiment (10 oder mehr Personen, z. B. eine Schulklasse)**

Jede Person sucht sich im obigen Beispielnetzwerk eine beliebige Seite aus. Von da bewegen sich alle durch das Netzwerk, indem sie den Links folgen und wenn nötig (oder gelegentlich auch einfach so) zu irgendeiner jeweils zufällig gewählten anderen Seite springen.
Nach einer Minute stoppen alle auf Kommando und merken sich die zuletzt besuchte Seite. Zum Schluss wird festgehalten, wie viele Personen auf jeder der sechs Seiten stehen geblieben sind.

Ein weniger konstruiertes und etwas größeres Beispiel für ein Netzwerk aus Verweisen ist dieses Buch selbst, denn wenn man die Kapitel als einzelne Lese-Aufenthaltsorte versteht, entsprechen die Hinweise auf andere Kapitel genau den Links zwischen Web-Seiten.

In der Netzwerkgrafik




sind alle Kapitel dieses Buches als Rechtecke dargestellt, deren Breite und Höhe der Anzahl der Verweise auf andere Kapitel bzw. aus anderen Kapiteln entspricht. Ein schlankes hohes Rechteck stellt somit ein Kapitel dar, das wenige Verweise enthält, auf das aber oft verwiesen wird. Die Farbe zeigt an, wie groß der PageRank (also die durch das Gleichungssystem bestimmte Zahl) im Verweisnetzwerk ist: Türkisene Rechtecke entsprechen den Kapiteln

mit niedrigstem PageRank, orangene solchen mit höchstem PageRank, und für Werte dazwischen sind entsprechende Mischfarben verwendet worden.

Tatsächlich sind – ganz gemäß ihrer zentralen Rolle in der Algorithmik – gerade die Kapitel über Sortieralgorithmen diejenigen, bei denen man mit Abstand am häufigsten landet, wenn man einfach zufällig den Querverweisen folgt und gelegentlich zu irgendeinem anderen Kapitel springt; so, als würde man im Web surfen.

## Lösungen

Das oben beschriebene Modell lässt sich in einer Relevanzbewertung aber trotzdem nicht ohne Weiteres verwenden, weil das Netzwerk, das eine Internet-Suchmaschine auswerten muss, zu einem Gleichungssystem mit Millarden von Unbekannten und Gleichungen führt. Das schaffen auch schnelle Computer nicht durch Auflösen und Einsetzen, wie man es in der Schule lernt.

Glücklicherweise hat das Gleichungssystem einige Eigenschaften, die man gut ausnutzen kann, wenn man die exakte Lösung gar nicht benötigt. Ein ganz einfacher Algorithmus kann nämlich sehr schnell eine ausreichend genaue Näherungslösung berechnen. Wir haben für jede Unbekannte eine Gleichung; wären alle anderen Unbekannten schon bestimmt, bräuchten wir sie nur noch in die eine übrige Gleichung einzusetzen. Der Algorithmus beginnt daher mit beliebigen Werten (z. B. dem gleichen für alle) für eine Lösung und rechnet für jede Unbekannte aus, was ihr richtiger Wert wäre, wenn alle anderen schon stimmten. Mit den so erhaltenen neuen Werten wird das Gleiche nochmal gemacht. Und nochmal. Und nochmal. Und so weiter und so fort.

PageRank-Algorithmus (im Prinzip)

```
1  Initialisiere die Relevanz aller Seiten mit 1
2  Solange sich das Ergebnis noch nennenswert ändert
3      Berechne für jede Seite S:
4          neue Relevanz von S ist
               4/5 · Σ_{über alle Seiten R mit einem Link R → S}  Relevanz von R / Anzahl Links von R  +  1/5 · 1 / Anzahl Seiten
```

Für das Beispiel mit sechs Knotenpunkte ergibt sich folgende Rechnung (die Werte sind auf die 5. Nachkommastelle gerundet):

| | Start | 1. Schritt | 2. Schritt | ... | 11. Schritt | 12. Schritt | ... | Lösung |
|---|---|---|---|---|---|---|---|---|
| $a$ | 1.00000 | 0.83333 | 0.29467 | ... | 0.10758 | 0.10740 | ... | 0.10665 |
| $b$ | 1.00000 | 0.32667 | 0.28222 | ... | 0.09259 | 0.09241 | ... | 0.09164 |
| $c$ | 1.00000 | 0.83333 | 0.70000 | ... | 0.12154 | 0.11940 | ... | 0.11865 |
| $d$ | 1.00000 | 0.30000 | 0.25556 | ... | 0.06592 | 0.06574 | ... | 0.06497 |
| $e$ | 1.00000 | 0.03333 | 0.03333 | ... | 0.03333 | 0.03333 | ... | 0.03333 |
| $f$ | 1.00000 | 0.30000 | 0.25556 | ... | 0.06592 | 0.06574 | ... | 0.06497 |

Mit jedem Schritt werden die Werte ein bisschen besser, und wenn sich kaum noch etwas ändert, ist das ein Zeichen dafür, dass man schon nahe an der exakten Lösung ist. Warum das so ist, wird genauer in Kap. 31 erklärt.

## Schlussbemerkungen

Zum Ende dieses Kapitels müsste die folgende Frage eigentlich leicht zu beantworten sein.

Frage: Wenn man seine eigene *Homepage* von vielen Freund(inn)en verlinken lässt, erscheint sie dann bei Google ganz oben?

Antwort: Das funktioniert nur, wenn deren Seiten selbst große Wichtigkeit im WWW haben – also eher nein.

Im Forschungszweig der Netzwerkanalyse werden noch viele andere Möglichkeiten untersucht, Wichtigkeit in einem Netzwerk aus verlinkten Seiten zu definieren, und die Ergebnisse können in vielfältiger Weise beim Reihen der Treffer berücksichtigt werden. Schon die Änderung von Details kann die Ergebnisse allerdings erheblich beeinflussen. So lässt sich beispielsweise die Häufigkeit der Sprünge im Verhältnis zur Linkverfolgung unterschiedlich festlegen oder die Auswahl der direkt angesprungenen Seiten steuern, indem man nicht jede mit gleicher Wahrscheinlichkeit berücksichtigt. Die spezifischen Festlegungen und viele weitere Details sind natürlich ein Betriebsgeheimnis, man gewinnt aber zumindest den Eindruck, dass Googles Einstellungen die schlechtesten nicht sein können.

## Zum Weiterlesen

1. Kapitel 31 (Gauß-Seidel Iteration zur Berechnung physikalischer Probleme)

   Hier wird die Lösung von Gleichungssystemen durch iteriertes Lösen einzelner Gleichungen ausführlicher erklärt.
2. Kapitel 29 (Die Eulertour)

   Euler-Touren gibt es zwar nicht in Königsberg, aber beim Nikolaus.
3. Der Wikipedia-Artikel
   `http://de.wikipedia.org/wiki/PageRank`

   Eine etwas abstraktere Formulierung, Hintergründe und weitere Verweise.

# Teil II

## Rechnen, Verschlüsseln und Codieren

# Übersicht

Berthold Vöcking

RWTH Aachen

Schon in den ersten Schuljahren wurden wir alle mit Algorithmen konfrontiert. Wir haben gelernt, wie man Zahlen, die aus mehreren Dezimalziffern bestehen, addiert, indem man sie untereinander schreibt und dann ziffernweise von rechts nach links aufaddiert, wobei jeweils ein Übertrag entstehen kann, der von Ziffer zu Ziffer mitgeführt wird. Aufbauend auf diesem einfachen Verfahren für die Addition haben wir dann gelernt, wie man Zahlen multipliziert, indem man erst Teilprodukte bildet, die man dann in einem Stufenschema aufaddiert. Diese *Schulmethoden* für die Addition und die Multiplikation folgen einfachen Regeln und können auch von einem Computer z. B. in Form eines Taschenrechners durchgeführt werden, der in der Regel zum Rechnen allerdings Bits an Stelle der Dezimalziffern verwendet. Der Computer ist dabei wesentlich schneller und zuverlässiger als wir, weshalb wir es heute kaum noch gewohnt sind, diese an sich einfachen Rechnungen mit Papier und Stift durchzuführen.

In diesem Teil des Buches geht es um eine Reihe von Algorithmen, die etwas mit dem Rechnen mit Zahlen zu tun haben. Er beginnt mit Algorithmen für einfache Berechnungsprobleme, die sich direkt auf Zahlen beziehen. In Kapitel 11 wird ein Verfahren für die schnelle Multiplikation vorgestellt, das große Zahlen wesentlich schneller multiplizieren kann als die uns allen bekannte Schulmethode. In Kapitel 12 wird der *Euklidische Algorithmus* erörtert, der auf sehr geschickte Art den größten gemeinsamen Teiler zweier Zahlen bestimmt. Dieses Verfahren ist schon seit der Antike bekannt und wird noch heute in verschiedenen Varianten eingesetzt. In der Antike kannte man auch schon Algorithmen zur Bestimmung von Primzahlen. In Kapitel 13 wird das *Sieb des Eratosthenes* erklärt, das eine Tabelle mit allen Primzahlen bis zu einer vorgegebenen Grenze erstellt. Es wird gezeigt, wie man mit einigen cleveren Tricks das einfache antike Verfahren gewaltig beschleunigen kann.

Nachdem geklärt ist, wie man mit Zahlen rechnen kann, geht es im Rest des Kapitels um Anwendungen des Rechnens mit Zahlen in ganz unterschiedlichen Bereichen und zu unterschiedlichen Zwecken.

Die Kryptographie beschäftigt sich mit dem Verschlüsseln von Nachrichten, damit diese vor Unbefugten geheim bleiben. In den Kapiteln 15 und 16 werden zwei unterschiedliche Ansätze zum Verschlüsseln von Nachrichten erörtert. Zunächst wird ein einfaches *symmetrisches Verfahren* erklärt, das *One-Time-Pad*. Dieses Verfahren ermöglicht es, eine Nachricht mit einem geheimen Kennwort zu ver- und entschlüsseln. Nur wer dieses Kennwort kennt, kann den Inhalt der Nachricht entschlüsseln. Anschließend wird das Prinzip der *asymmetrischen Verschlüsselung* erläutert, auf dem die meisten heute im Internet eingesetzten Verschlüsselungsverfahren beruhen. Hierbei kommen zwei zueinander passende Kennwörter zum Einsatz: ein öffentlicher Schlüssel, mit dem Nachrichten verschlüsselt werden, und ein geheimer Schlüssel, mit dem Nachrichten entschlüsselt werden können. Kapitel 17 stellt ein kryptographisches Verfahren vor, mit dem Geheimnisse zwischen verschiedenen Personen aufgeteilt werden können. So können z. B. Seeräuber eine geheime Schatzkarte oder Bankangestellte einen geheimen Code für einen Safe derart untereinander aufteilen, dass alle Teilnehmer zusammenkommen müssen, um den Schatz zu bergen bzw. den Safe zu öffnen. Kapitel 18 beschreibt eine weitere interessante Anwendung von kryptographischen Verfahren. Es wird erklärt, wie man über das Internet Karten spielen kann, ohne dass einer der beteiligten Spieler sich beim Verteilen der Karten einen Vorteil erschleichen kann.

Viele Verfahren aus der Kryptographie basieren auf Erkenntnissen aus der Zahlentheorie. Eine wichtige Rolle spielen dabei so genannte *Einwegfunktionen*, die zuvor in Kapitel 14 erläutert werden. Dabei handelt es sich um Funktionen, die man zwar relativ leicht mit dem Computer berechnen kann, aber deren Umkehrfunktion sehr schwierig zu berechnen ist. Das asymmetrische Verschlüsselungsverfahren aus Kapitel 16 basiert beispielsweise darauf, dass man Zahlen mit jeweils mehreren hundert Dezimalziffern sehr schnell miteinander multiplizieren kann, z. B. mit dem Algorithmus, der in Kapitel 11 beschrieben wurde. Durch wiederholtes Multiplizieren kann man effizient die Exponentialfunktion berechnen. Für die Umkehrung der Exponentiation, der Berechnung des sogenannten diskreten Logarithmus, gibt es jedoch bis heute keinen Algorithmus mit einer Laufzeit, auf die ein Mensch warten könnte, selbst wenn man alle Computer der Welt an diesem Problem rechnen lassen würde.

Die letzten drei Kapitel dieses Buchteiles setzen sich mit verschiedenen Methoden zur Codierung von Daten auseinander. In den Kapiteln 19 und 20 werden so genannte *Fingerprinting-* und *Hashing*-Verfahren vorgestellt, die sehr große Datenmengen derart verdichten bzw. komprimieren, dass diese durch nur ein paar wenige Bits repräsentiert werden können. Durch diese extreme Form der Komprimierung geht natürlich Information verloren. Die erzeugten Bitfolgen können aber beispielsweise wie Fingerabdrücke verwendet werden, um die Gleichheit zweier Datenbestände zu testen, indem man nur ein paar Bits austauscht. Kapitel 21 beschäftigt sich mit einer völlig anderen Art der Codierung. Bei *fehlerkennenden Codes* findet keine Komprimierung statt, sondern der eigentlichen Information wird sogar zusätzliche, eigentlich

überflüssige Information (Redundanz) hinzugefügt, die dem Empfänger zur Bestimmung von Fehlern und Fehlerpositionen dient.

Im Kern aller hier vorgestellten Methoden steckt das Rechnen mit Zahlen. Erstaunlich ist, für wie vielfältige Aufgaben man die einfachen Operationen Addition, Subtraktion, Multiplikation und Division auf den ganzen Zahlen benutzen kann!

# 11

# Multiplikation langer Zahlen (schneller als in der Schule)

Arno Eigenwillig und Kurt Mehlhorn

Max-Planck-Institut für Informatik, Saarbrücken

Multiplizieren haben wir alle schon in der Grundschule gelernt. Um zwei ganze Zahlen $a$ und $b$ miteinander zu multiplizieren, multipliziert man $a$ mit jeder Ziffer von $b$ und arrangiert diese Teilprodukte in einem Stufenschema. Dann addiert man die Teilprodukte spaltenweise.

$$\begin{array}{r} 5678 \cdot 4321 \\ \hline 22712 \\ 17034 \\ 11356 \\ 5678 \\ \hline 24534638 \end{array}$$

Wir nennen diese Methode die *Schulmethode der Multiplikation*. Wenn die Zahlen $a$ und $b$ lang sind, also aus vielen Ziffern bestehen, dann ist diese Methode recht aufwändig. Wir wollen nachfolgend untersuchen:

1. Was genau heißt hier „aufwändig“?
2. Wie können wir mit weniger Aufwand multiplizieren?

Das ist aus zweierlei Gründen interessant:

1. **Praxis:** Schnelles Rechnen mit langen Zahlen braucht man in vielen Anwendungsgebieten der Informatik, z. B. beim Verschlüsseln von Nachrichten (siehe Kap. 14 und 16) und bei der zuverlässigen Lösung geometrischer und rechnerischer Probleme.
2. **Theorie:** Die Schulmethode der Multiplikation erscheint uns so vertraut und natürlich, dass jede wesentliche Verbesserung – und wir werden eine solche kennen lernen – eine bemerkenswerte Überraschung ist.

Aber was heißt nun, die Schulmethode „ist aufwändig“? Informatiker messen Rechenaufwand nicht in Sekunden (denn nächstes Jahr wird es schon wieder schnellere Computer zu kaufen geben), sondern in der Anzahl der *Grundoperationen*, die eine Methode ausführt. Eine Grundoperation ist etwas, was

ein Computer oder ein Mensch in einem einzelnen gedanklichen Schritt tun kann. Die Grundoperationen, die wir hier brauchen, sind Rechnungen mit den Ziffern $0, 1, 2, \ldots, 8, 9$, aus denen die Zahlen zusammengesetzt sind:

1. **Multiplikation von zwei Ziffern:** Wir kennen das kleine Einmaleins, also können wir zu zwei Ziffern $x$ und $y$, die man uns gibt, in einem Schritt die zwei Ziffern $u$ und $v$ ihres Produktes bestimmen: $x \cdot y = 10 \cdot u + v$.
   **Beispiel:** Für die Ziffern $x = 3$ und $y = 7$ wissen wir $x \cdot y = 3 \cdot 7 = 21 = 10 \cdot 2 + 1$, also haben wir die Ergebnisziffern $u = 2$ und $v = 1$. Für $x = 3$ und $y = 2$ haben wir $u = 0$ und $v = 6$.
2. **Addition von drei Ziffern:** Wir können zu drei Ziffern $x, y, z$ in einem Schritt die zwei Ziffern ihrer Summe ausgeben: $x + y + z = 10 \cdot u + v$. (Wir sehen gleich, warum wir hier drei statt zwei Ziffern haben wollen.)
   **Beispiel:** Für $x = 3$, $y = 5$ und $z = 4$ haben wir $u = 1$ und $v = 2$, weil $3 + 5 + 4 = 12 = 10 \cdot 1 + 2$.

Wie viele Grundoperationen braucht nun die Schulmethode der Multiplikation? Bevor wir das beantworten können, müssen wir erst einmal über die Addition von zwei Zahlen reden, denn diese brauchen wir ja auch fürs Multiplizieren.

### Die Addition langer Zahlen

Wie aufwändig ist es, zwei Zahlen $a$ und $b$ zu addieren? Das hängt natürlich von ihrer Länge ab. Sagen wir, $a$ und $b$ bestehen beide aus $n$ Ziffern. (Wenn eine von beiden kürzer ist, fügen wir einfach vorne Nullen an, dann stimmt's wieder.) Um sie zu addieren, schreiben wir sie untereinander und addieren von rechts nach links die Ziffern in jeder Spalte mit der oben eingeführten Ziffern-Addition. Vom Ergebnis $10 \cdot u + v$ schreiben wir die Ziffer $v$ als Ergebnisziffer hin; die Ziffer $u$ schreiben wir als dritte Ziffer (Übertrag) bei der nächsten Spalte dazu. Hier ist ein Beispiel mit vier Ziffern, also für $n = 4$:

$$\begin{array}{r} 6917 \\ 4269 \\ \hline 1101 \\ \hline 11186 \end{array}$$

So gehen wir von rechts nach links alle $n$ Spalten durch. Den letzten Übertrag schreiben wir ohne weitere Rechnung als linkeste Ziffer des Ergebnisses hin. Insgesamt haben wir dann $n$ Grundoperationen durchgeführt, nämlich eine Ziffern-Addition für jede Spalte.

### Die Multiplikation einer Zahl mit einer Ziffer

Erinnern wir uns wieder an die Schulmethode der Multiplikation. Die einzelnen Teilprodukte (Zeilen), die bei ihr auftreten, entstehen durch Multiplikation einer langen Zahl $a$ (nämlich dem linken Faktor) mit einer einzelnen Ziffer $y$

(die kommt aus dem rechten Faktor). Diese Multiplikation „Zahl mal Ziffer“ schauen wir uns jetzt genauer an. Dazu schreiben wir ihre Zwischenergebnisse etwas ausführlicher hin als sonst: Wir gehen von rechts nach links die Ziffern von $a$ durch. Jede Ziffer $x$ von $a$ multiplizieren wir mit $y$. Das Ergebnis $10 \cdot u + v$ schreiben wir in eine neue Zeile, und zwar so, dass $v$ in derselben Spalte wie $x$ steht und $u$ links daneben. Anschließend addieren wir alle diese zweistelligen Zwischenergebnisse. Das liefert uns das Teilprodukt, das wir gewöhnlich direkt in eine einzelne Zeile schreiben. Für das erste Teilprodukt aus unserem Anfangsbeispiel sieht das so aus:

$$
\begin{array}{r}
5\,6\,7\,8 \cdot 4 \\
\hline
\quad 3\,2 \\
\quad 2\,8 \\
\quad 2\,4 \\
2\,0 \\
0\,0\,1\,0 \\
\hline
2\,2\,7\,1\,2
\end{array}
$$

Wie viele Grundoperationen haben wir durchgeführt? Für jede der $n$ Ziffern von $a$ haben wir eine Ziffern-Multiplikation durchgeführt. (Im obigen Beispiel: vier Ziffern-Multiplikationen für die vier Ziffern der Zahl 5678.) Anschließend mussten wir in $n+1$ Spalten die Zwischenergebnisse addieren. In der rechtesten Spalte steht nur eine einzelne Ziffer, da müssen wir nichts rechnen. Aber in den anderen $n$ Spalten stehen zwei Ziffern und eventuell ein Übertrag aus der vorigen Spalte, so dass wir pro Spalte eine einzelne Ziffern-Addition brauchen. Das macht $n$ Ziffern-Additionen. Zusammen mit den Ziffern-Multiplikationen sind das $2 \cdot n$ Grundoperationen für die Multiplikation „Zahl mal Ziffer“.

## Die Schulmethode der Multiplikation: Analyse

Jetzt untersuchen wir die Anzahl der Grundoperationen, die die Schulmethode der Multiplikation für zwei lange Zahlen $a$ und $b$ benötigt, die beide aus $n$ Ziffern bestehen. (Wenn eine kürzer ist, schreiben wir einfach so lange Nullen davor, bis sie genau so lang wie die andere ist.)

Für jede Ziffer $y$ von $b$ müssen wir ein Teilprodukt $a \cdot y$ ausrechnen. Das ist eine Multiplikation „Zahl mal Ziffer“, benötigt also $2 \cdot n$ Grundoperationen, wie wir uns oben überlegt haben. Weil $b$ insgesamt $n$ Ziffern hat, müssen wir $n \cdot (2 \cdot n) = 2 \cdot n^2$ Grundoperationen durchführen, um alle Teilprodukte zu errechnen.

$$
\begin{array}{r}
5\,6\,7\,8 \cdot 4\,3\,2\,1 \\
\hline
2\,2\,7\,1\,2\,0\,0\,0 \\
0\,1\,7\,0\,3\,4\,0\,0 \\
0\,0\,1\,1\,3\,5\,6\,0 \\
0\,0\,0\,0\,5\,6\,7\,8 \\
\hline
2\,4\,5\,3\,4\,6\,3\,8
\end{array}
$$

Als nächstes müssen wir die Teilprodukte so zusammenzählen, wie sie schräg untereinanderstehen. Um uns die Rechnung zu vereinfachen, denken wir uns an den leeren Stellen Nullen hingeschrieben; dann haben wir einfach $n$ Zahlen, die gerade untereinander stehen und die wir alle addieren müssen. Dazu verwenden wir mehrmals die oben beschriebene Methode zur Addition langer Zahlen: Erst addieren wir die erste Zeile zur zweiten Zeile, zu dieser Zwischensumme addieren wir die dritte Zeile usw., bis wir schließlich alle $n$ Teilprodukte (Zeilen) addiert haben. Dazu brauchen wir $n-1$ Additionen langer Zahlen. (Im Beispiel ist $n = 4$, und wir brauchen drei Additionen zum Zusammenzählen der vier Teilprodukte, nämlich die folgenden: $22712000+1703400 = 24415400$, $24415400 + 113560 = 24528960$ und $24528960 + 5678 = 24534638$.)

Aber wie viele Grundoperationen sind das? Um das sagen zu können, müssen wir die Länge aller Zwischensummen kennen, die während dieser Kette von Additionen auftreten. Dafür denken wir ein bisschen um die Ecke: Das Endergebnis $a \cdot b$ unserer Rechnung kann höchstens $2 \cdot n$ Ziffern haben. (Man überlege sich, warum!) Während wir Zeile um Zeile addieren, werden die Zahlen immer nur größer, nie wieder kleiner. Deswegen können auch alle Zwischenergebnisse nur eine Länge von höchstens $2 \cdot n$ Ziffern haben. Nach der obigen Untersuchung über die Addition heißt das: Jede einzelne Addition von Teilprodukten braucht höchstens $2 \cdot n$ Grundoperationen. Für unsere $n-1$ Additionen dieser Zahlen brauchen wir also höchstens $(n-1) \cdot (2 \cdot n) = 2 \cdot n^2 - 2 \cdot n$ Grundoperationen. Zusammen mit den $2 \cdot n^2$ Grundoperationen für das Ausrechnen der Teilprodukte sind das also insgesamt höchstens $4 \cdot n^2 - 2 \cdot n$ Grundoperationen, um mit der Schulmethode zwei Zahlen aus jeweils $n$ Ziffern miteinander zu multiplizieren; darunter sind $n^2$ Ziffern-Multiplikationen.

Was bedeutet das konkret? Wenn wir mit wirklich langen Zahlen rechnen wollen, sagen wir: mit 100.000 Ziffern, dann brauchen wir fast 40 Milliarden Grundoperationen für eine einzige Multiplikation, darunter 10 Milliarden Ziffern-Multiplikationen. Für jede einzelne Ausgabeziffer eines solchen Produkts brauchen wir im Mittel ca. 200.000 Rechenoperationen. Das ist ein offensichtlich sehr ungünstiges Verhältnis, und es wird noch schlechter, wenn die Zahlen länger werden: Bei 1 Million Ziffern brauchen wir fast 4 Billionen Grundoperationen (davon eine Billion Ziffern-Multiplikationen). Das sind im Mittel ca. 2 Millionen Grundoperationen für eine einzelne Ergebnisziffer.

## Die Methode von Karazuba

Jetzt besprechen wir eine Methode, die mit wesentlich weniger Grundoperationen zwei Zahlen aus $n$ Ziffern multiplizieren kann. Sie ist nach dem russischen Mathematiker Anatolij Alexejewitsch Karazuba (in englischer Transkription: Karatsuba) benannt, von dem ihre zentrale Idee stammt (veröffentlicht 1962

mit Yu. Ofman[1]). Wir beschreiben diese Methode zunächst für Zahlen mit ein, zwei oder vier Ziffern, dann für Zahlen jeder Länge.

Der einfachste Fall ist die Multiplikation zweier einstelliger Zahlen ($n = 1$); beispielsweise $8 \cdot 4 = 32$. Dazu braucht man nur eine Grundoperation, nämlich eine einzelne Ziffern-Multiplikation, die direkt das Ergebnis liefert.

Der nächste Fall, den wir uns ansehen wollen, ist der Fall $n = 2$, also die Multiplikation von zweistelligen Zahlen $a$ und $b$. Wir schreiben sie zerlegt in ihre Ziffern hin:

$$a = p \cdot 10 + q \quad \text{und} \quad b = r \cdot 10 + s.$$

Ist beispielsweise $a = 78$ und $b = 21$, so lautet die Zerlegung in Ziffern

$$p = 7 \text{ und } q = 8 \quad \text{sowie} \quad r = 2 \text{ und } s = 1.$$

Jetzt überlegen wir uns, wie das Produkt $a \cdot b$ in Ziffern ausgedrückt aussieht:

$$\begin{aligned} a \cdot b &= (p \cdot 10 + q) \cdot (r \cdot 10 + s) \\ &= (p \cdot r) \cdot 100 + (p \cdot s + q \cdot r) \cdot 10 + q \cdot s. \end{aligned}$$

Für das obige Beispiel ($a = 78$ und $b = 21$) ist das

$$78 \cdot 21 = (7 \cdot 2) \cdot 100 + (7 \cdot 1 + 8 \cdot 2) \cdot 10 + 8 \cdot 1 = 1638.$$

So wie es dasteht, sieht man direkt, dass man das Produkt der zweistelligen Zahlen $a$ und $b$ ausrechnen kann, indem man **vier** Multiplikationen einstelliger Zahlen durchführt und die Ergebnisse (gegeneinander verschoben) addiert. Das ist genau das, was die Schulmethode der Multiplikation auch macht. Karazuba verdanken wir die Entdeckung, dass aber auch schon **drei** Multiplikationen einstelliger Zahlen ausreichen, mit denen man die folgenden Werte berechnet:

$$\begin{aligned} u &= p \cdot r, \\ v &= (q - p) \cdot (s - r), \\ w &= q \cdot s. \end{aligned}$$

Beim Berechnen von $v$ muss man etwas genauer hinschauen, weil dabei Subtraktionen von zwei Ziffern auftreten. Wir brauchen also Ziffern-Subtraktion als eine weitere Grundoperation, die wir hier zweimal anwenden müssen. Ihre Ergebnisse $(q-p)$ und $(s-r)$ haben zwar nur eine Ziffer, aber möglicherweise ein negatives Vorzeichen. Wenn man sie multipliziert, um $v$ zu erhalten, muss

[1] A. Karazuba, Yu. Ofman, „Multiplikation vielstelliger Zahlen auf Automaten“ (russisch), *Doklady Akad. Nauk SSSR* **145** (1962), S. 293–294. Englische Übersetzung: A. Karatsuba, Yu. Ofman, “Multiplication of multidigit numbers on automata”, *Soviet Physics Doklady* **7** (1963), pp. 595–596. In dieser Arbeit beschreibt Karazuba seinen Trick, um lange Zahlen effizient zu *quadrieren*. Die *Multiplikation* der Zahlen $a$ und $b$ führt er mit der Formel $ab = \frac{1}{4}((a+b)^2 - (a-b)^2)$ auf zweimaliges Quadrieren zurück.

man zunächst für die beiden Ziffern eine Ziffern-Multiplikation durchführen und dann nach den üblichen Regeln („minus mal minus ist plus“ usw.) das Vorzeichen bestimmen.

Aber warum hilft das alles nun beim Multiplizieren? Weil folgende Gleichung gilt:

$$u + w - v = p \cdot r + q \cdot s - (q - p) \cdot (s - r) = p \cdot s + q \cdot r.$$

Der Karazuba-Trick besteht nun darin, mit Hilfe dieser Gleichung das Produkt $a \cdot b$ folgendermaßen auszudrücken:

$$a \cdot b = u \cdot 10^2 + (u + w - v) \cdot 10 + w.$$

Rechnen wir das einmal für unser Beispiel $a = 78$ und $b = 21$ durch. Es ist

$$\begin{aligned} u &= 7 \cdot 2 &&= 14, \\ v &= (8 - 7) \cdot (1 - 2) &&= -1, \\ w &= 8 \cdot 1 &&= 8. \end{aligned}$$

Damit finden wir

$$\begin{aligned} 78 \cdot 21 &= 14 \cdot 100 + (14 + 8 - (-1)) \cdot 10 + 8 \\ &= 1400 + 230 + 8 \\ &= 1638. \end{aligned}$$

Jetzt haben wir nur noch drei statt vier Ziffern-Multiplikationen benutzt; hinzu kommen mehrere Additionen und Subtraktionen für das Zusammensetzen der Teilergebnisse.

### Die Methode von Karazuba für vierstellige Zahlen

Nach dem Fall $n = 2$ wenden wir uns jetzt dem Fall $n = 4$ zu, also der Multiplikation von zwei vierstelligen Zahlen $a$ und $b$. Genau wie oben können wir sie in je zwei Hälften $p$ und $q$ bzw. $r$ und $s$ teilen; diese Hälften vierstelliger Zahlen sind jetzt aber keine Ziffern mehr, sondern zweistellige Zahlen:

$$a = p \cdot 10^2 + q \quad \text{und} \quad b = r \cdot 10^2 + s.$$

Aus diesen vier Hälften berechnen wir mit der Karazuba-Multiplikation zweistelliger Zahlen wieder die drei Hilfsprodukte

$$\begin{aligned} u &= p \cdot r, \\ v &= (q - p) \cdot (s - r), \\ w &= q \cdot s \end{aligned}$$

und ganz genau wie vorher erhalten wir das Produkt von $a$ und $b$ als

$$a \cdot b = u \cdot 10^4 + (u + w - v) \cdot 10^2 + w.$$

**Beispiel:** Nehmen wir uns noch einmal die Aufgabe $a \cdot b$ für $a = 5678$ und $b = 4321$ vor. Zuerst spalten wir $a$ und $b$ in die Hälften $p = 56$ und $q = 78$ sowie $r = 43$ und $s = 21$. Dann berechnen wir mit Hilfe der Karazuba-Methode für zweistellige Zahlen die Hilfsprodukte

$$\begin{aligned} u &= 56 \cdot 43 &&= 2408, \\ v &= (78 - 56) \cdot (21 - 43) &&= -484, \\ w &= 78 \cdot 21 &&= 1638. \end{aligned}$$

Damit ergibt sich

$$\begin{aligned} 5678 \cdot 4321 &= 2408 \cdot 10000 + (2408 + 1638 - (-484)) \cdot 100 + 1638 \\ &= 24080000 + 453000 + 1638 \\ &= 24534638. \end{aligned}$$

Für diese Rechnung haben wir drei Hilfsprodukte zweistelliger Zahlen bilden müssen, und wir haben uns ja gerade im vorigen Abschnitt überlegt, wie das mit der Karazuba-Methode geht. Für jedes Hilfsprodukt braucht man drei Ziffern-Multiplikationen. Insgesamt brauchen wir also $3 \cdot 3 = 9$ Ziffern-Multiplikationen und mehrere Additionen und Subtraktionen, um zwei vierstellige Zahlen nach der Karazuba-Methode zu multiplizieren. Mit der Schulmethode hätten wir 16 Ziffern-Multiplikationen und mehrere Additionen benötigt.

### Die Methode von Karazuba für beliebig lange Zahlen

Nach dem gleichen Prinzip fortfahrend können wir die Multiplikation 8-stelliger Zahlen auf drei Multiplikationen 4-stelliger Zahlen zurückführen, die Multiplikation 16-stelliger Zahlen auf drei Multiplikationen 8-stelliger Zahlen usw. Mit anderen Worten, für die Karazuba-Methode kann die Länge $n$ der beiden Zahlen $a$ und $b$ irgendeine Potenz von 2 sein, wie nämlich $2 = 2^1$, $4 = 2 \cdot 2 = 2^2$, $8 = 2 \cdot 2 \cdot 2 = 2^3$, $16 = 2 \cdot 2 \cdot 2 \cdot 2 = 2^4$ usw.

In allgemeiner Form können wir die Karazuba-Methode so beschreiben: Zwei Zahlen $a$ und $b$ der Länge $n = 2 \cdot 2 \cdot 2 \cdots 2 = 2^k$ teilen wir auf als

$$a = p \cdot 10^{n/2} + q \quad \text{und} \quad b = r \cdot 10^{n/2} + s$$

und berechnen dann ihr Produkt mit drei Multiplikationen von Zahlen der Länge $\frac{n}{2} = 2^{k-1}$ in der Art

$$a \cdot b = p \cdot r \cdot 10^n + (p \cdot r + q \cdot s - (q - p) \cdot (s - r)) \cdot 10^{n/2} + q \cdot s$$

Auf diese Weise brauchen wir für die Multiplikation von Zahlen der Länge $2^k$ nur dreimal (statt viermal) so viele Ziffern-Multiplikationen wie für Zahlen der Länge $2^{k-1}$. Damit ergibt sich die folgende Tabelle für die Anzahl der Ziffern-Multiplikationen, die die Karazuba-Methode und die Schulmethode jeweils benötigen, um Zahlen einer bestimmten Länge $n$ zu multiplizieren:

| Länge | Karazuba | Schulmethode |
|---:|---:|---:|
| $1 = 2^0$ | 1 | 1 |
| $2 = 2^1$ | 3 | 4 |
| $4 = 2^2$ | 9 | 16 |
| $8 = 2^3$ | 27 | 64 |
| $16 = 2^4$ | 81 | 256 |
| $32 = 2^5$ | 243 | 1.024 |
| $64 = 2^6$ | 729 | 4.096 |
| $128 = 2^7$ | 2.187 | 16.384 |
| $256 = 2^8$ | 6.561 | 65.536 |
| $512 = 2^9$ | 19.638 | 262.144 |
| $1.024 = 2^{10}$ | 59.049 | 1.048.576 |
| $1.048.576 = 2^{20}$ | 3.486.784.401 | 1.099.511.627.776 |
| ... | ... | ... |
| $n = 2^k$ | $3^k$ | $4^k$ |

Wer das Rechnen mit Logarithmen beherrscht, kann die Tabelleneinträge auch leicht als Funktion von $n$ ausdrücken:
In der Spalte für die Schulmethode steht für $n = 2^k$ der Wert $4^k$. Wir schreiben log für den Logarithmus zur Basis 2. Damit gilt $k = \log(n)$ und

$$4^k = 4^{\log(n)} = (2^{\log(4)})^{\log(n)} = n^{\log(4)} = n^2.$$

Bei der Karazuba-Methode hingegen steht

$$3^k = 3^{\log(n)} = (2^{\log(3)})^{\log(n)} = n^{\log(3)} = n^{1{,}58\ldots}.$$

Vergleichen wir nun einmal die Tabelle mit unserer Analyse der Schulmethode für die Multiplikation von Zahlen mit einer Million Ziffern. Die Schulmethode benötigte fast 4 Billionen Grundoperationen, davon eine Billion Ziffern-Multiplikationen. Um statt dessen die Karazuba-Methode anwenden zu können, müssen wir zunächst vor die eine Million Ziffern noch eine Kette von Nullen schreiben, um auf die nächsthöhere Zweierpotenz zu kommen, nämlich $2^{20} = 1.048.576$. (Andernfalls könnten wir die Länge nicht immer weiter durch 2 teilen, bis wir bei 1 angekommen sind.) Dann können wir mit der Karazuba-Methode multiplizieren und brauchen dafür „nur“ etwa dreieinhalb Milliarden Ziffern-Multiplikationen (siehe Tabelle). Das ist gerade noch ein 287stel des Aufwandes der Schulmethode. (Zum Vergleich: Das Verhältnis von einer Sekunde zu den sprichwörtlichen „fünf Minuten“ ist ein 300stel.) Wir sehen also: Die Karazuba-Methode ist im Rechenaufwand erheblich sparsamer; zumindest dann, wenn man – so wie wir – nur die Ziffern-Multiplikationen zählt. Für eine präzise Untersuchung muss man auch noch beachten, welchen Aufwand das Addieren und Subtrahieren der Zwischenergebnisse verursacht. Unsere Beispiele mit zwei und vier Ziffern möchte man dann vielleicht lieber nach der Schulmethode ausrechnen. Wenn die Zahlen ziemlich lang werden, gewinnt die Karazuba-Methode aber, weil sie viel weniger Zwischenergebnisse als die Schulmethode produziert. Ab welcher Länge genau sie schneller ist, hängt von den Eigenheiten des benutzten Computers ab.

## Zusammenfassung

Was ist das Erfolgsrezept hinter der Karazuba-Methode? Es besteht aus zwei entscheidenden Ideen.

Die erste Idee ist eine ganz allgemeine: Die vorgelegte Aufgabe „multipliziere zwei Zahlen der Länge $n$“ wird zurückgeführt auf mehrere Aufgaben von der gleichen Art, aber von kleinerer Größe, nämlich: „multipliziere zwei Zahlen der Länge $\frac{n}{2}$“. Damit verkleinern wir das Problem so lange, bis es ganz einfach geworden ist („multipliziere zwei Ziffern“). Dieses Prinzip nennt sich *divide and conquer* (dt.: teile und herrsche), und wir haben es schon in früheren Kapiteln in Aktion gesehen (z. B. beim schnellen Sortieren in Kap. 3). Für die verschiedenen Größen des Problems programmiert man natürlich nicht jeweils eine neue Prozedur, sondern man schreibt eine Prozedur für allgemeine Länge $n$, die sich für die verringerte Problemgröße $\frac{n}{2}$ mehrfach selbst aufruft. Dies nennt man *Rekursion*, und das ist eine der wichtigsten Techniken überhaupt in der Informatik. Rekursion haben wir auch schon in vorangehenden Kapiteln kennengelernt (etwa bei der Tiefensuche in Kap. 7).

Die zweite Idee, speziell für die Multiplikation, ist der Karazuba-Trick, mit dem es gelingt, bei jeder rekursiven Aufteilung des Problems nur drei statt vier Unterprobleme lösen zu müssen. Dieser scheinbar winzige Unterschied ergibt über die ganze Rekursion hinweg eine enorme Ersparnis und macht den Vorteil der Karazuba-Methode gegenüber der Schulmethode aus.

## Zum Weiterlesen

1. A. K. Dewdney: *The (New) Turing Omnibus*. 2. Auflage, 1993; Nachdruck (Taschenbuch) 2001.
   Von der englischen 1. Auflage erschien eine deutsche Übersetzung unter dem Titel *Der Turing Omnibus* bei Springer, 1995.

   Diese schöne „Reise durch die Informatik mit 66 Stationen“ führt im Kapitel *Fast Multiplication: Divide and Conquer* zu den Multiplikationsverfahren von Karazuba (für lange Zahlen) und von Strassen (eine ähnliche Idee für Matrizen).

2. Wolfram Koepf: *Computeralgebra*. Springer, 2006.

   Dieses einführende, deutschsprachige Lehrbuch für Studenten der Mathematik und Informatik bespricht Algorithmen für alle Grundrechenarten mit Zahlen und Polynomen, einschließlich des Karazuba-Algorithmus, und enthält Beispiel-Programmcode für das Computeralgebra-System *Mathematica*.

3. Joachim von zur Gathen, Jürgen Gerhard: *Modern Computer Algebra*. Cambridge University Press, 2. Auflage, 2003.

   Dieses liebevoll gestaltete, englischsprachige Lehrbuch für fortgeschrittene Studenten der Informatik und Mathematik bespricht in Kap. 8 den Algorithmus von Karazuba für die Multiplikation von Polynomen sowie noch effizientere Verfahren, die auf der „schnellen Fourier-Transformation“ beruhen.

4. Donald E. Knuth: *Arithmetik*. Springer, 2001.
Deutsche Übersetzung von Kap. 4 aus *The Art of Computer Programming*, Band 2: *Seminumerical Algorithms*. Addison-Wesley, 3. Auflage, 1998.

   Dieser schwergewichtige Klassiker der theoretischen Informatik behandelt in Abschn. 4.3 neben dem hier dargestellten Algorithmus von Karazuba auch noch weitere, insbesondere das anspruchsvolle Verfahren von Schönhage und Strassen mit einer zu $n \cdot \log(n) \cdot \log(\log(n))$ proportionalen Laufzeitschranke.

5. Aus Wikipedia:
`http://de.wikipedia.org/wiki/Karatsuba-Algorithmus`
`http://de.wikipedia.org/wiki/Strassen-Algorithmus`

## Danksagung

Die Autoren danken H. Alt, M. Dietzfelbinger und C. Klost für hilfreiche Anmerkungen.

# 12

# Der Euklidische Algorithmus

Friedrich Eisenbrand

Universität Paderborn

In diesem Kapitel behandeln wir einen der ältesten, bereits aus Aufzeichnungen aus der Antike bekannten Algorithmus. Er wurde ca. 300 v. Chr. von *Euklid* in seinem Buch *Die Elemente* beschrieben. Heute ist dieser Algorithmus von fundamentaler Bedeutung in vielen Bereichen der Informatik. Insbesondere im Bereich der Kryptographie (siehe Kap. 16) muss man sich immer wieder auf den *Euklidischen Algorithmus* stützen.

Stell dir vor, du hast zwei Stäbe der Länge $a$ und $b$, wobei $a$ und $b$ natürliche Zahlen sind. Du möchtest die Stäbe in gleich große Stücke zersägen und zwar so, dass die gemeinsame Länge $d$ der Stücke so groß ist wie möglich. Am Ende soll kein Verschnittrest verbleiben. Wir könnten beide Stäbe beispielsweise in Stücke der Länge 1 zersägen. Sind größere Stücke möglich?
Unser Algorithmus berechnet die gesuchte maximale Länge $d$ der Stücke. Der Algorithmus hat zwei Versionen. Die erste Version ist langsam, die zweite ist schnell.

Die Zahlen $a/d$ und $b/d$ sind ebenfalls natürliche Zahlen. Sie sind jeweils die Anzahl der Stücke, in die die Stäbe zerschnitten werden. In dem Bild unten

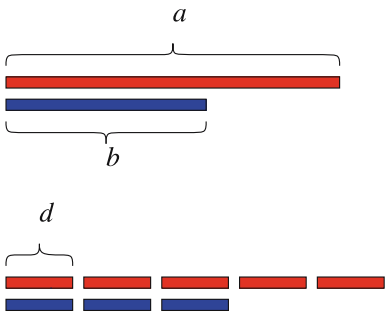



**Abb. 12.1.** Zerteilen zweier Stäbe in möglichst große Stücke

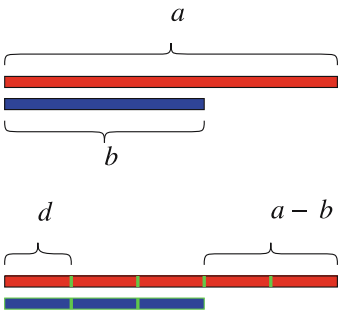



**Abb. 12.2.** Die gesuchte Länge für $a$ und $b$ ist die gesuchte Länge für $a - b$ und $b$

lässt sich der längere Stab in 5 Stücke der Länge $d$ zerteilen und der kürzere in 3.

Wie finden wir nun die gesuchte Länge $d$? Wenn beide Stäbe gleich lang sind, dann ist die gesuchte Länge $d$ sofort klar. Es ist die gemeinsame Länge beider Stäbe, wir müssen also die Stäbe nicht zersägen. Nehmen wir also an, dass die Stäbe unterschiedliche Länge haben, wobei $a$ größer ist als $b$.
Beim Nebeneinanderlegen beider Stäbe (siehe obige Abbildung) fällt dir etwas auf! Wenn beide Stäbe sich in Stücke der Länge $d$ aufteilen lassen, dann können wir doch von dem längeren Stab ein Stück abschneiden, das genauso lang ist wie der kürzere Stab.
Der neue Stab hat jetzt die Länge $a - b$ und lässt sich ebenfalls exakt in Stücke der Länge $d$ zersägen. Umgekehrt gilt: Wenn sich der neue Stab der

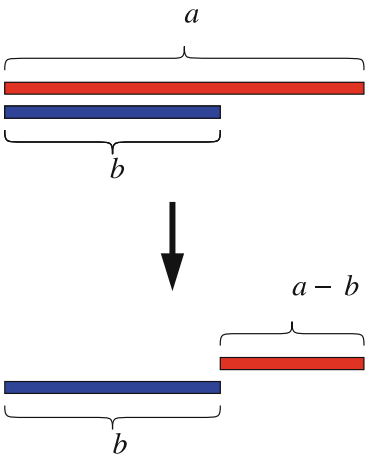



**Abb. 12.3.** Ein Schritt des Algorithmus

Länge $a - b$ und der Stab der Länge $b$ in Stücke der Länge $d$ zerteilen lassen, dann lässt sich auch der alte Stab der Länge $a$ in Stücke der Länge $d$ zerteilen.

Diese Erkenntnis formulieren wir gesondert. Es ist das wichtige Prinzip, welches unserem Algorithmus nun zugrunde liegen wird.

**Prinzip (P)**

Wenn $a = b$, dann ist die gesuchte Länge $a$.
Wenn $a$ größer ist als $b$, dann ist die gesuchte Länge für $a$ und $b$ dieselbe wie die gesuchte Länge für $a - b$ und $b$.

Wir können jetzt einen Algorithmus aufschreiben, der uns die gesuchte Länge berechnet.

**Bestimmung der größten gemeinsamen Stücklänge**

Solange nicht beide Stäbe gleich groß sind:

> Säge von dem längeren Stab ein Stück ab, das so lang ist wie der kürzere der beiden Stäbe. Lege dieses Stück beiseite.

Jetzt sind beide Stäbe gleich groß. Die gesuchte Länge ist die gemeinsame Länge der Stäbe.

Wir müssen uns an dieser Stelle fragen, ob unser Algorithmus fertig wird und nicht immer weiter in kleinere Stücke zersägt und nie aufhört. Hierzu kann man Folgendes beobachten. Wir beginnen mit zwei Stäben, deren Längen jeweils natürliche Zahlen $a$ und $b$ sind. Während der Algorithmus läuft, bleiben die Längen natürliche Zahlen, da die Differenz zweier natürlicher Zahlen eine ganze Zahl ist. Insbesondere ist die Länge der beiden Stäbe immer mindestens 1. Da einer der Stäbe in jeder Runde um mindestens eine Längeneinheit gekürzt wird, endet der Algorithmus schließlich nach höchstens $a + b$ Runden.

### Der größte gemeinsame Teiler

Die gesuchte Länge $d$ ist, so wie $a$ und $b$, auch eine natürliche Zahl. Es ist eine natürliche Zahl, die sowohl $a$ als auch $b$ *teilt*. Mathematisch ausgedrückt bedeutet dies, dass es natürliche Zahlen $x$ und $y$ gibt mit $d \cdot x = a$ und $d \cdot y = b$. In unserer Anschauung ist $x$ die Anzahl der Stücke, die wir aus dem Stab der Länge $a$ schneiden, und $y$ ist die Anzahl der Stücke, die wir aus dem Stab der Länge $b$ schneiden.

Die Zahl $d$ ist der *größte gemeinsame Teiler* von $a$ und $b$, denn nach dem Prinzip (P) ist der größte gemeinsame Teiler der Stablängen nach jeder Säge-Runde derselbe wie vor dieser Runde.

Wir können den Algorithmus auch abstrakter aufschreiben. Dann kommen keine Stäbe mehr vor. Die Eingabe sind zwei natürliche positive Zahlen $a$ und $b$ und die Ausgabe ist der größte gemeinsame Teiler, der ggT von $a$ und $b$. Wir

nennen den Algorithmus aus Gründen, die gleich beleuchtet werden, LangsamEuklid.

**LangsamEuklid**

Solange $a \neq b$

Falls $a$ größer ist als $b$, dann ersetze $a$ durch $a - b$
Falls $b$ größer ist als $a$, dann ersetze $b$ durch $b - a$

Gib den gemeinsamen Wert der beiden Zahlen aus

Schauen wir uns ein konkretes Beispiel an:

Die Eingabe sollen die Zahlen 15 und 9 sein. Im ersten Schritt subtrahieren wir 9 von 15 und erhalten die Zahlen 6 und 9. Im zweiten Schritt erhalten wir die Zahlen 6 und 3. Im dritten Schritt erhalten wir die Zahlen 3 und 3 und der Algorithmus gibt die Zahl 3 aus.

Das nächste Beispiel zeigt, warum wir den Algorithmus LangsamEuklid genannt haben. Betrachte die Eingaben $a = 1001$ und $b = 2$. Die beiden Zahlen während der Iterationen des Algorithmus sind

1001 und 2
999 und 2,
997 und 2
995 und 2
... (ziemlich viele Runden)
3 und 2
1 und 2
1 und 1.

Dass es so lange dauert, liegt daran, dass die zweite Zahl, im Vergleich zur ersten Zahl, sehr klein ist.

### Eine Beobachtung, die den Algorithmus erheblich beschleunigt

Wie oft zieht man die Zahl 2 von der ersten Zahl insgesamt ab? Es gilt $1001 = 2 \cdot 500 + 1$. Die Zahl 2 wird genau 500-mal von der Zahl 1001 subtrahiert, bis der Wert unter den Wert 2 fällt.

Wie kamen wir auf die Zahl 500? Einfach indem wir 1001 durch 2 geteilt und abgerundet haben. Wir wussten dann, dass $1001 = 500 \cdot 2 + 1$ gilt und konnten dann sofort die Zahlen 1001 und 2 durch die Zahlen 1 und 2 ersetzen.

Ein Computer kann sehr schnell eine *Division mit Rest* ausführen. Diese Operation berechnet aus $a$ und $b$ zwei natürliche Zahlen $q$ und $r$ mit $a = q \cdot b + r$, wobei $r$ kleiner ist als $b$. Die Zahl $q$ ist das Ergebnis des *Abrundens* der Zahl

$a/b$, und $r$ ist der *Rest* der Division von $a$ durch $b$. In unserem Beispiel ist $a = 1001$, $b = 2$, $q = 500$ und $r = 1$.

Wenn $a$ und $b$ die Eingabe des Algorithmus LangsamEuklid sind und $a$ größer ist als $b$, dann wird $b$ von $a$ entweder $q$-mal subtrahiert (wenn tatsächlich ein Rest bleibt) oder $q - 1$-mal (wenn $a$ durch $b$ teilbar ist) und schließlich zwei Stäbe der Länge $b$ entstehen. Wir können den Algorithmus also beschleunigen, indem wir $a$ durch $b$ mit Rest teilen und $a$ sofort durch den Rest $r$ ersetzen. Dabei kann es passieren, dass der Rest $r$ Null ist. Das bedeutet, dass $b$ der gesuchte größte gemeinsame Teiler ist, und der Algorithmus ist fertig.

Dies ist die Idee hinter dem folgenden Algorithmus, den wir jetzt Euklid nennen.

Euklid

```
1  if a < b: vertausche a und b.
2  while b > 0:
3      berechne q, r mit a = q · b + r, wobei 0 ≤ r < b;
4      a := b; b := r;
5  return a.
```

### Analyse

Wir können erwarten, dass dieser Algorithmus viel schneller ist als der Algorithmus LangsamEuklid. Warum das so ist, kann man genau begründen. Nehmen wir an, dass $a$ die größere der beiden Zahlen ist. Wie groß ist der Rest $r$, der bei der Division von $a$ durch $b$ übrig bleibt? In Abb. 12.4 sieht man, dass der Rest immer eine Länge von höchstens $a/2$ hat. Das liegt daran, dass $a$ mindestens so groß ist wie $b + r$ und $b$ größer ist als $r$. Daraus folgt, dass $a$ größer ist als $2 \cdot r$, also $r$ durch $a/2$ beschränkt ist.

Nach einer Runde wird $a$ durch den Rest $r$ ersetzt, welcher höchstens so groß ist wie $a/2$. In der zweiten Runde wird $b$ durch den Rest der Division

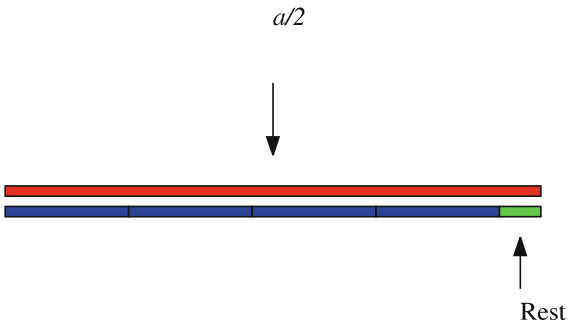



**Abb. 12.4.** Der Rest ist klein

von $b$ durch $r$ ersetzt. Diese Zahl ist auch höchstens so groß wie $a/2$. Also sind nach zwei Runden beide Zahlen höchstens halb so groß wie $a$, welche die größere Zahl der Eingabe war.

Wenn wir jetzt $2 \cdot k$ Runden hintereinander betrachten, dann haben beide Zahlen höchstens den Wert $a/2^k$. Wenn $k > \log_2 a$, dann hätten beide Zahlen den Wert Null. Das kann aber nicht passieren, da der Algorithmus fertig ist, sobald eine Zahl Null ist. Daher ist die Anzahl der Schleifendurchläufe des Algorithmus höchstens $2 \cdot \log_2 a$, wobei wir wieder den Logarithmus zur Basis 2 meinen.

Die *Anzahl der Ziffern*, die wir in unserem Dezimalsystem brauchen, um die Zahl $a$ aufzuschreiben, ist proportional zu $\log_2 a$. Während der Algorithmus LangsamEuklid eine Laufzeit hat, die proportional zu den *Werten* von $a$ und $b$ ist, hat der Algorithmus Euklid eine Laufzeit, die proportional zur *Anzahl der Ziffern* von $a$ und $b$ ist. Das ist ein wesentlicher Unterschied.

### Ein Beispiel

Zum Schluss berechnen wir noch von Hand den größten gemeinsamen Teiler von $a = 1324$ und $b = 145$. Die erste Division mit Rest ergibt $1324 = 9 \cdot 145 + 19$. Jetzt wird $a$ auf den Wert 145 gesetzt und $b$ auf den Wert 19.

Die zweite Division mit Rest ist $145 = 7 \cdot 19 + 12$. Die nächste Division mit Rest ergibt $19 = 1 \cdot 12 + 7$, dann folgen $12 = 7 + 5$, $7 = 5 + 2$, $5 = 2 \cdot 2 + 1$ und $2 = 2 \cdot 1 + 0$, woraus wir schließen können, dass der größte gemeinsame Teiler von 1324 und 145 die Zahl 1 ist. Man sagt dann auch, die Zahlen sind *teilerfremd*.

## Zum Weiterlesen

1. Donald E. Knuth: *Arithmetik*. Springer, 2001.
   Deutsche Übersetzung von Kapitel 4 aus *The Art of Computer Programming*, Band 2: *Seminumerical Algorithms*. Addison-Wesley, 3. Auflage, 1998.

   In diesem klassischen Lehrbuch wird der Euklidische Algorithmus in Kapitel 4.5.2 behandelt. Unter anderem wird dort gezeigt, dass unsere Analyse der Laufzeit scharf ist. Genauer wird dort gezeigt, dass es eine Folge $F_0, F_1, F_2, \ldots$ natürlicher Zahlen gibt, bei der die Anzahl der Ziffern, die man benötigt um $F_i$ darzustellen, proportional zu $i$ ist und der Euklidische Algorithmus bei Eingabe $F_n$ und $F_{n-1}$ mindestens $n$ Schritte durchführt.

2. Joachim von zur Gathen, Jürgen Gerhard: *Modern Computer Algebra*. Cambridge University Press, 2. Auflage, 2003.

   Dieses liebevoll gestaltete, englischsprachige Lehrbuch für fortgeschrittene Studenten der Informatik und Mathematik bespricht in Kapitel 6 den Euklidischen Algorithmus. Auch wird die Komplexität im Sinne Anzahl der Grundoperationen (siehe Kap. 11 (Multiplikation langer Zahlen)) analysiert. In dem Buch sowie

in dem oben aufgeführten Buch von Knuth werden auch Algorithmen beschrieben, die den größten gemeinsamen Teiler zweier Zahlen in Zeit proportional zu $M(n) \log n$ berechnen. Hier ist $n$ die Anzahl der Ziffern in der Eingabe und $M(n)$ bezeichnet die Anzahl an Grundoperationen, die man benötigt um zwei Zahlen, die jeweils höchstens $n$ Ziffern haben, miteinander zu multiplizieren.

3. Aus Wikipedia:
   `http://de.wikipedia.org/wiki/Euklidischer_Algorithmus`

## Danksagung

Der Autor dankt M. Dietzfelbinger für viele hilfreiche Kommentare und Anregungen.

# 13

# Das Sieb des Eratosthenes: Wie schnell kann man eine Primzahlentabelle berechnen?

Rolf Möhring und Martin Oellrich

Technische Universität Berlin

Eine **Primzahl** ist eine natürliche Zahl mit der Eigenschaft, dass sie durch keine andere natürliche Zahl außer 1 und sich selbst ohne Rest teilbar ist. Primzahlen sind in der Menge aller natürlichen Zahlen unregelmäßig verteilt und haben dadurch Mathematiker seit Tausenden von Jahren fasziniert und beschäftigt.

Eine **Primzahlentabelle bis n** ist eine Liste aller Primzahlen zwischen den Zahlen 1 und $n$. Sie beginnt folgendermaßen:

$$2 \quad 3 \quad 5 \quad 7 \quad 11 \quad 13 \quad 17 \quad 19 \quad 23 \quad 29 \quad 31 \quad 37 \quad 41 \quad \ldots$$

Im Laufe der Zeit sind viele Problemstellungen gefunden worden, in denen Primzahlen eine Rolle spielen. Nicht alle konnten bisher restlos geklärt werden. Hier zwei Beispiele.

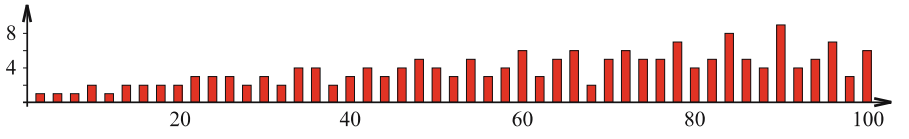
*Christian Goldbach* (1694–1764) formulierte 1742 eine interessante Beobachtung:

> *Jede gerade Zahl größer oder gleich 4 ist darstellbar als Summe zweier Primzahlen.*

Beispielsweise finden wir:

$$4 = 2 + 2\,, \quad 6 = 3 + 3\,, \quad 8 = 3 + 5\,, \quad 10 = 3 + 7 = 5 + 5 \quad \text{etc.}$$

Diese Behauptung verlangt nur, dass es mindestens eine solche Darstellung gibt. Tatsächlich gibt es für die meisten Zahlen sogar mehrere. Das folgende Diagramm wurde mit Hilfe einer Primzahltabelle erstellt und zeigt die Anzahlen solcher Darstellungsmöglichkeiten. Auf der $x$-Achse stehen die zerlegten (geraden) Zahlen.

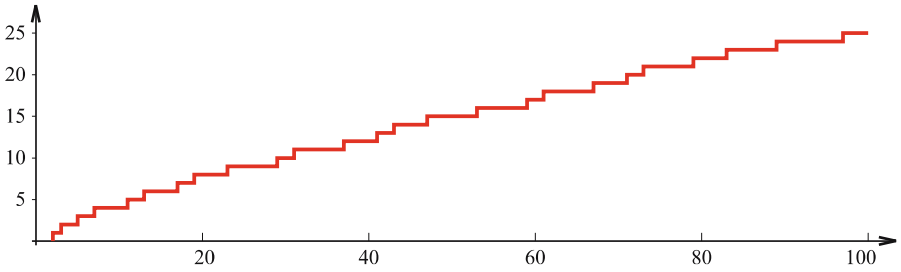


Der leichte Aufwärtstrend in den Säulen setzt sich bei wachsendem $n$ immer weiter fort und es wurde keine gerade Zahl gefunden, für die diese Behauptung nicht gilt. Dennoch konnte bis heute kein Beweis gefunden werden, dass sie *für alle* gilt.

*Carl Friedrich Gauß* (1777–1855) untersuchte die Verteilung der Primzahlen, indem er sie zählte. Er betrachtete die Funktion

$$\pi(n) := \text{ Anzahl aller Primzahlen zwischen 1 und } n.$$

Ein Diagramm dieser Funktion sieht so aus:




Man nennt $\pi(n)$ aus offensichtlichen Gründen eine *Treppenfunktion*. Gauß konstruierte dazu eine „glatte“ Kurve, die so nah wie möglich bei $\pi(n)$ bleibt, egal wie groß $n$ wird. Um sich ein Bild von seinem Vorhaben zu machen und sein Ergebnis später zu kontrollieren, brauchte er eine Primzahltabelle.

(Dieses Problem ist seitdem gelöst, ein tieferes Eingehen würde jedoch den Rahmen dieses Buchs sprengen.)

Heute sind Primzahlen nicht mehr nur eine Herausforderung für Mathematiker, sondern von ganz praktischem Wert. So spielen etwa 100-stellige Primzahlen in der elektronischen Verschlüsselung von Daten eine zentrale Rolle.

## Von der Idee zum Verfahren

Soweit wir heute wissen, hat ein Grieche den ersten Algorithmus zur Berechnung von Primzahltabellen vorgestellt: **Eratosthenes von Kyrene** (ca. 276–194 v. Chr.). Er war ein hochrangiger Gelehrter im antiken Alexandria und ein Leiter der berühmten Bibliothek, in der das gesamte Wissen der damaligen Zeit gesammelt war. Er arbeitete mit an den damals wesentlichen Fragen der Astronomie, Geologie und Mathematik: Welchen Umfang hat die Erde? Woher kommt der Nil? Wie kann man aus einem Würfel einen zweiten konstruieren, der das doppelte Volumen hat?

Wir wollen im Weiteren nachvollziehen, wie er aus einer einfachen Grundidee ein praktikables Verfahren entwickelt hat, das schon zu seiner Zeit gut auf Papyrus oder Sand auszuführen war. Dabei wollen wir untersuchen, wie schnell dieser Algorithmus in der Praxis ist, wenn wir eine recht große Primzahltabelle berechnen wollen. Nehmen wir als Maßstab für „groß“ die Zahl *eine Milliarde*, also $n = 10^9$.

**Eine einfache Idee**

Gemäß der Definition von Primzahlen gilt für jede Zahl $m$, die *keine* Primzahl ist: es gibt zwei Zahlen $i, k$ mit den Eigenschaften

$$2 \leq i, k \leq m \quad \text{und} \quad i \cdot k = m \, .$$

Diese Gesetzmäßigkeit können wir benutzen, um einen sehr einfachen Algorithmus für eine Primzahltabelle zu formulieren:

- schreibe alle Zahlen von 2 bis $n$ in eine Liste,
- bilde alle Produkte $i \cdot k$, wobei $i$ und $k$ Zahlen zwischen 2 und $n$ sind und
- streiche alle Ergebnisse, die vorkommen, aus der Liste.

Dass diese einfache Vorschrift das Gewünschte leistet, ist sofort zu sehen: Alle Zahlen, die am Schluss noch in der Liste stehen, kamen nicht als Produktergebnis vor. Sie können also nicht als ein solches Produkt geschrieben werden und sind demnach Primzahlen.

## Wie schnell läuft die Berechnung?

Um nun zu untersuchen, was der Algorithmus an welcher Stelle genau tut, schreiben wir die in der Grundidee enthaltenen Handlungsvorschriften etwas formaler und geben den Zeilen Nummern:

PRIMZAHLTABELLE (Grundversion)

```
1  procedure PRIMZAHLTABELLE
2  begin
3      schreibe alle Zahlen von 2 bis n in eine Liste
4      for i := 2 to n do
5          for k := 2 to n do
6              streiche die Zahl i · k aus der Liste
7          endfor
8      endfor
9  end
```

Steht bei Ausführung von Schritt 6 die Zahl $i \cdot k$ nicht in der Liste, so passiert nichts.

Dieser Algorithmus kann ohne Schwierigkeiten auf einem Computer programmiert werden und wir können seinen Zeitverbrauch untersuchen. Auf einem LINUX PC (3.2 GHz) ergeben sich folgende Laufzeiten:

| $n$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
|---|---|---|---|---|
| *Zeit* | 0.00 $s$ | 0.20 $s$ | 19.4 $s$ | 1943.4 $s$ |

Es ist gut zu erkennen, dass eine Erhöhung von $n$ um den Faktor 10 eine Verlängerung der Rechenzeit um einen Faktor von ca. 100 mit sich bringt. Das ist auch zu erwarten, denn sowohl $i$ als auch $k$ laufen über einen ca. 10-mal so großen Bereich. Es werden also ca. 100-mal soviele Produkte $i \cdot k$ gebildet.

Daraus können wir schon jetzt sehen: Um auf $n = 10^9$ zu kommen, müssten wir die Zeit bei $n = 10^6$ um den Faktor $(10^9/10^6)^2 = 10^6$ erhöhen und würden dafür $1943 \cdot 10^6$ Sekunden = *61 Jahre und 7 Monate* brauchen. Das ist natürlich untauglich für die Praxis.

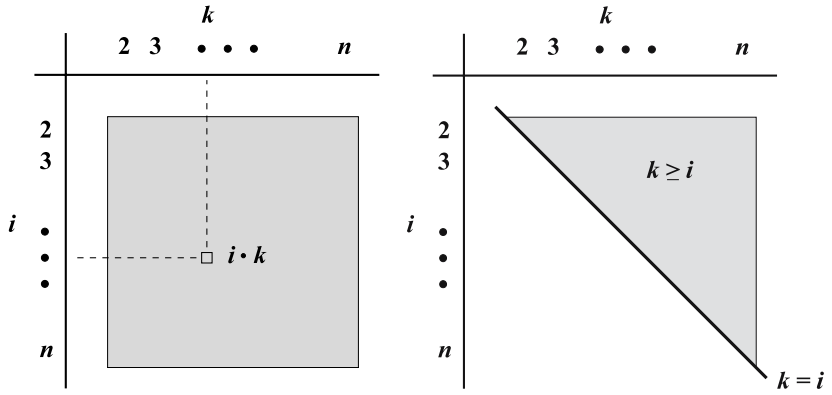
## Womit verbringt der Algorithmus seine Zeit?

Er erzeugt alle Produkte $i \cdot k$ in einem gewissen Bereich (Abb. 13.1(a)).

Jedoch wird jedes einzelne Ergebnis für $i \cdot k$ nur einmal benötigt. Danach ist es aus der Liste entfernt und der Algorithmus bräuchte es eigentlich nie wieder zu erzeugen. Wo macht er Arbeit zuviel? Zum Beispiel dort, wo $i$ und $k$ genau vertauschte Werte haben, etwa $i = 3, k = 5$ und später $i = 5, k = 3$. In beiden Fällen ist das Ergebnis des Produkts dasselbe, wie uns die Vertauschungsregel der Multiplikation lehrt: es ist immer $i \cdot k = k \cdot i$. Deshalb können wir $k \geq i$ festlegen, um diese Doppelungen zu vermeiden (Abb. 13.1(b)).

Diese Idee spart auf Anhieb die halbe Arbeit! Jedoch sind auch 30 Jahre und 10 Monate noch zu lange für unsere Tabelle. Wo können wir noch Arbeit sparen? Dort, wo in Schritt 6 sowieso nichts passiert, wenn nämlich $i \cdot k > n$ ist. Die Liste enthält ja nur Zahlen bis $n$, jenseits von $n$ ist nichts zu streichen.

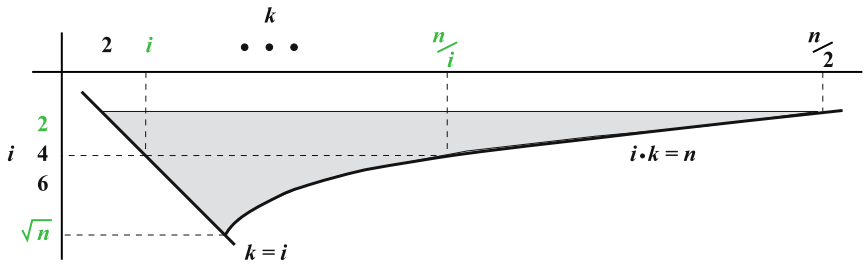
Um das zu erreichen, brauchen wir die $k$-Schleife (Zeile 5) nur für solche Werte auszuführen, für die $i \cdot k \leq n$ ist. Diese Bedingung selbst sagt uns, welche $k$ das sind: $k \leq n/i$. Als willkommenen Nebeneffekt können wir auch




(a) $i \cdot k$ für alle $i, k \in \{2, \ldots, n\}$ (b) $i \cdot k$ nur für $k \geq i$

**Abb. 13.1.** Berechnung der Produkte $i \cdot k$ in einem gewissen Bereich

den Laufbereich für $i$ begrenzen. Aus den beiden Einschränkungen $i \le k \le n/i$ erhalten wir $i^2 \le n$, also $i \le \sqrt{n}$. Für höhere $i$ ist der $k$-Bereich leer. Der jetzt erzeugte Zahlenbereich sieht wie folgt aus:




Der Algorithmus hat bis hierher das folgende Aussehen bekommen:

PRIMZAHLTABELLE (besser)

```
1  procedure PRIMZAHLTABELLE
2  begin
3      schreibe alle Zahlen von 2 bis n in eine Liste
4      for i := 2 to ⌊√n⌋ do
5          for k := i to ⌊n/i⌋ do
6              streiche die Zahl i · k aus der Liste
7          endfor
8      endfor
9  end
```

(Das Zeichen $\lfloor \cdot \rfloor$ bedeutet Abrundung, denn $i$ und $k$ können nur ganzzahlige Werte annehmen.)

Wie schnell sind wir jetzt geworden? Die neuen Laufzeiten:

| $n$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ | $10^8$ | $\mathbf{10^9}$ |
|---|---|---|---|---|---|---|
| *Zeit* | 0.00 *s* | 0.01 *s* | 0.01 *s* | 2.3 *s* | 32.7 *s* | **452.9 s** |

Der Effekt ist recht spürbar, denn das Ziel $10^9$ ist schon in Reichweite: nur noch siebeneinhalb Minuten entfernt. Lassen wir den Computer in Gedanken laufen und nutzen diese Zeit, um es vielleicht noch besser zu machen!

### Brauchen wir jeden *i*-Wert?

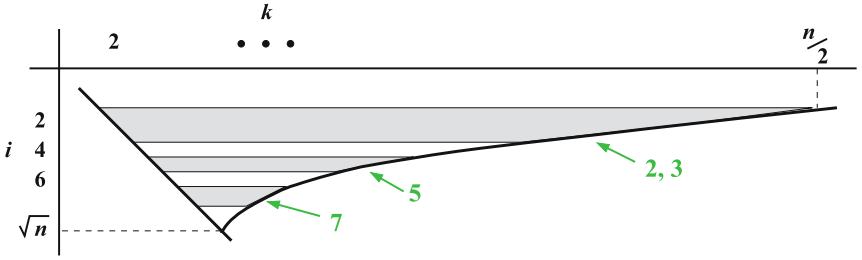
Schauen wir an, was innerhalb der $i$-Schleife (Zeile 4) genau passiert: $i$ bleibt fest und $k$ durchläuft seine Schleife (Zeile 5). Dabei erhält das Produkt $i \cdot k$ die Werte

$$i^2,\ i(i+1),\ i(i+2),\ \ldots.$$

Wenn die $k$-Schleife zu Ende gelaufen ist, stehen in der Liste keine echten Vielfachen mehr von $i$. Ebenso nicht von allen Zahlen kleiner als $i$, denn sie wurden auf dieselbe Weise schon vorher gestrichen.

Was passiert, wenn $i$ keine Primzahl ist? Beispiel $i = 4$: Das Produkt $i \cdot k$ durchläuft die Werte $16, 20, 24, \ldots$. Jedoch sind diese Zahlen alle auch Vielfache von 2, da 4 selbst Vielfaches von 2 ist. Es ist im Grunde für $i = 4$ gar nichts zu tun. Ebenso ist das mit jeder anderen geraden Zahl $i > 4$.

Beispiel $i = 9$: das Produkt $i \cdot k$ durchläuft nur Vielfache von 9. Die sind aber als Vielfache von 3 schon durchlaufen und ebenfalls unnötig. So ist das mit allen Nichtprimzahlen, denn sie haben einen kleineren Primteiler, der vor ihnen schon ein Wert für $i$ war. Wir brauchen also die $k$-Schleife nur für Primzahlen $i$ auszuführen, siehe folgende Abbildung:




Ob nun $i$ eine Primzahl ist oder nicht, könnte uns die Liste selbst sagen – wenn sie schon fertig wäre. Wir können aber erst nach dem Ende des Algorithmus sicher sein, dass nur noch Primzahlen in der Liste stehen. Oder?

Ja und nein. Im allgemeinen *Ja*, denn sonst könnten wir den Algorithmus abkürzen. Das ist nicht immer der Fall, nehmen wir z. B. $n = 100$: Die Nichtprimzahl 91 muss irgendwann aus der Liste gestrichen werden. Sie wird aber erst ganz kurz vor dem Ende erzeugt, nämlich für $i = 7$, $k = 13$.

In unserem speziellen Fall aber *Nein*, denn wir wollen ja nicht jede beliebige Zahl als Primzahl erkennen, sondern eine ganz bestimmte, die Zahl $i$. Und das auch nicht zu beliebiger Zeit, sondern erst zum Anfang der $k$-Schleife für den Wert $i$. Hier gibt uns die Liste eine korrekte Auskunft! Warum?

Wir hatten oben beobachtet, dass für jedes feste $i$ alle gestrichenen Werte $i \cdot k \geq i^2$ sind. Anders ausgedrückt: Im Zahlenbereich $2, \ldots, i^2 - 1$ wird nichts verändert. Da $i$ im weiteren Verlauf nur steigt, wächst auch dieser Bereich und enthält alle Bereiche vor ihm. Im folgenden Bild sind diese Bereiche blau dargestellt. Die erste „falsche“ Zahl in der jeweiligen Tabelle ist rot.

| | $i^2-1=3$ | | $=8$ | | | | | $=15$ | | | | | | | | | | | | | $=24$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $i=2$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| $i=3$ | 2 | 3 | | 5 | | 7 | | 9 | | 11 | | 13 | | 15 | | 17 | | 19 | | 21 | | 23 | | 25 | | 27 |
| $i=4$ | 2 | 3 | | 5 | | 7 | | | | 11 | | 13 | | | | 17 | | 19 | | | | 23 | | 25 | | |
| $i=5$ | 2 | 3 | | 5 | | 7 | | | | 11 | | 13 | | | | 17 | | 19 | | | | 23 | | 25 | | |
| • • • | | | | | | | | | | | | | | | | | | | | | | | | | | |

Da innerhalb dieser Bereiche bis zum Ende des Algorithmus keine Veränderungen mehr auftreten, müssen sie bereits *vor* dem Durchlauf der $k$-Schleife

für das jeweilige $i$ korrekt sein. Die Tabelle wird sozusagen in quadratischen Sprüngen fertig. Grün eingezeichnet ist der Wert $i$ selbst, den wir auf seine Primeigenschaft prüfen wollen. Es ist gut zu erkennen, dass er immer in einem blauen Bereich liegt. Um zu entscheiden, ob $i$ eine Primzahl ist, dürfen wir also einfach in der aktuellen Liste nachschauen.

Wir können im Algorithmus die $i$-Schleife jetzt wie folgt ergänzen:

PRIMZAHLTABELLE (Eratosthenes)

```
1   procedure PRIMZAHLTABELLE
2   begin
3       schreibe alle Zahlen von 2 bis n in eine Liste
4       for i := 2 to ⌊√n⌋ do
5           if i steht in der Liste then
6               for k := i to ⌊n/i⌋ do
7                   streiche die Zahl i · k aus der Liste
8               endfor
9           endif
10      endfor
11  end
```

Diese Version des Verfahrens hatte der kluge Grieche vorgestellt und es heißt nach seinem Erfinder das **Sieb des Eratosthenes**. *Sieb* deswegen, weil es die gewünschten Objekte, die Primzahlen, nicht gezielt konstruiert, sondern im Gegenteil alle Nichtprimzahlen aussondert.

Unsere Zeitmessung sagt zu seinem Algorithmus:

| $n$ | $10^6$ | $10^7$ | $10^8$ | $\mathbf{10^9}$ |
|---|---|---|---|---|
| *Zeit* | 0.02 *s* | 0.43 *s* | 5.4 *s* | **66.5 s** |

Bei $n = 10^9$ nur noch gut eine Minute!

### Geht es noch schneller?

Mit einem ähnlichen Argument wie für die $i$-Werte können wir auch die Werte der $k$-Schleife weiter einschränken: Wir brauchen nur diejenigen zu betrachten, die in der Liste gefunden werden! Steht $k$ nämlich nicht mehr dort, ist $k$ als Nichtprimzahl gestrichen worden und besitzt einen Primteiler $p < k$. Im Durchgang der $i$-Schleife mit $i = p$ wurden alle Vielfachen von $p$ gestrichen, insbesondere $k$ und seine Vielfachen. Es ist nichts mehr zu tun.

Es wäre nun nahe liegend, den Algorithmus wie folgt zu ergänzen:

```
6               for k := i to ⌊n/i⌋ do
7                   if k steht in der Liste then
8                       streiche die Zahl i · k aus der Liste
9                   endif
```

*Doch Achtung!* Diese Formulierung hat ihre Tücken. Lassen wir den Algorithmus so laufen, erzeugt er die folgende Tabelle:

2 3 5 7 *8* 11 *12* 13 17 19 *20* 23 *27* *28* 29 31 *32* 37 ...

Was läuft falsch? Sehen wir uns die ersten Schritte des Algorithmus genau an. Nach der Initialisierung der Liste mit allen Zahlen bis $n$ (Zeile 3) steht darin:

2 3 4 5 6 7 8 9 10 11 ...

Zunächst wird $i = 2$ gesetzt und dann $k = 2$. Die 2 steht in der Liste, also wird $i \cdot k = 4$ gestrichen:

2 3 – 5 6 7 8 9 10 11 ...

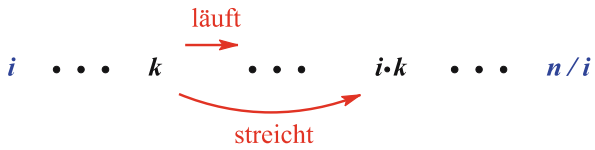
Im nächsten Schritt ist $k = 3$. Die 3 steht ebenfalls in der Liste und so wird $i \cdot k = 6$ gestrichen:

2 3 – 5 – 7 8 9 10 11 ...

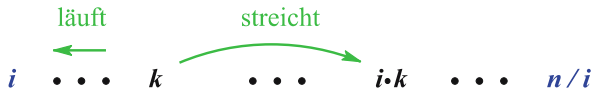
Jetzt passiert's: $k = 4$ steht nicht mehr in der Liste, denn sie wurde ja als erste gestrichen. Entsprechend wird für $k = 4$ wegen der neuen Bedingung nichts gemacht und der Algorithmus fährt mit $k = 5$ fort:

2 3 – 5 – 7 *8* 9 – 11 ...

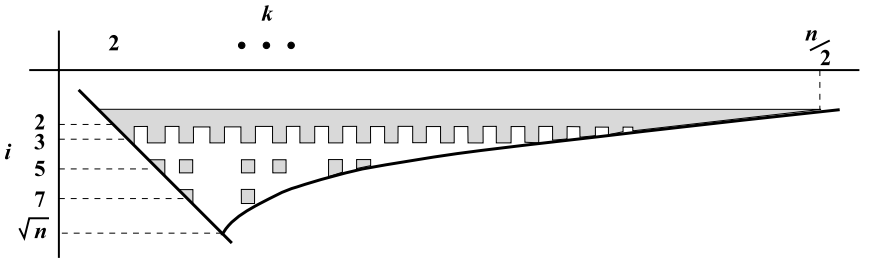
So bleibt $2 \cdot 4 = 8$ irrtümlich in der Liste stehen. Das Problem ist, dass $k$ beim Schleifendurchlauf stets größere Zahlen $i \cdot k > k$ streicht und dann selbst erhöht wird. Irgendwann bekommt $k$ den Wert eines vormaligen Produktes $i \cdot k$ und das Verfahren wirkt ungünstig auf sich selbst zurück:




Die Lösung besteht darin, $k$ seinen Schleifenbereich *rückwärts* durchlaufen zu lassen. Dadurch wird diese Rückwirkung vermieden:




Mit dieser Überlegung werden nur noch die folgenden Produkte $i \cdot k$ gebildet:




Insgesamt ergibt sich nun die folgende Version des Algorithmus:

PRIMZAHLTABELLE (Endversion)

```
1   procedure PRIMZAHLTABELLE
2   begin
3       schreibe alle Zahlen von 2 bis n in eine Liste
4       for i := 2 to ⌊√n⌋ do
5           if i steht in der Liste then
6               for k := ⌊n/i⌋ to i step -1 do
7                   if k steht in der Liste then
8                       streiche die Zahl i · k aus der Liste
9                   endif
10              endfor
11          endif
12      endfor
13  end
```

Seine Laufzeiten:

| $n$ | $10^6$ | $10^7$ | $10^8$ | $\mathbf{10^9}$ |
|---|---|---|---|---|
| *Zeit* | 0.01 *s* | 0.15 *s* | 1.6 *s* | **17.6 s** |

Dieses Ergebnis ist für heutige Standards durchaus akzeptabel. Wir haben ausgehend von der naiven Grundversion mit wenigen gezielten Überlegungen den Algorithmus für $n = 10^9$ um den *Faktor 254.5 Millionen* beschleunigt!

**Was können wir aus diesem Beispiel lernen?**

1. Einfache Rechenverfahren sind nicht automatisch *effizient*.
2. Zu ihrer Beschleunigung muss man sie *gut verstehen*.
3. Es sind oft *viele Verbesserungen* möglich.
4. *Mathematische Ideen* können sehr weitreichend sein!

## Weitere Betrachtungen

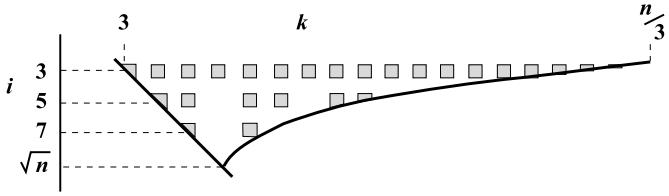
17.6 Sekunden sind natürlich ein toller Erfolg für ein wenig Nachdenken. Doch wie gut ist das denn? Haben wir schon den „Anschlag“ erreicht?

Überlegen wir, was der Algorithmus – egal in welcher Variante – letztlich leisten muss. Er muss alle Nichtprimzahlen bis $n$ mindestens einmal erzeugen, um sie aus der Liste zu streichen. Davon gibt es unterhalb von $n = 10^9$ genau 949.152.466 Stück. Wenn wir nun mitzählen, wie viele Produkte $i \cdot k$ gebildet werden, erhalten wir bei den oben betrachteten Varianten folgende Werte:

| | Grundversion | besser | Eratosthenes | Endversion |
|---|---|---|---|---|
| Anz. Produkte | $10^{18}$ | $9.44 \cdot 10^9$ | $2.55 \cdot 10^9$ | $9.49 \cdot 10^8$ |
| Verhältnis zu Nichtprimzahlen | $1.1 \cdot 10^9$ | 9.9 | 2.7 | **1.0** |

Tatsächlich macht die letzte Variante nicht mehr Arbeit als nötig, sie ist in diesem Sinne optimal!

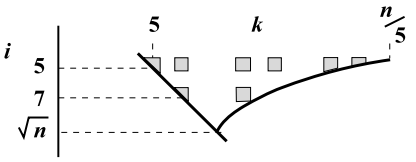
Interessanterweise bedeutet das aber nicht, dass man den Algorithmus nicht weiter verbessern könnte. Der Vergleich mit den zu streichenden Nichtprimzahlen ist nichts Absolutes, denn man kann die Nichtprimzahlen noch verringern. Das geht mit folgendem Trick: Die Tabelle wird nicht mit allen Zahlen ab 2 angelegt, sondern mit der 2 und allen ungeraden Zahlen ab 3. Alle geraden Zahlen $\geq 4$ sind sowieso nicht prim, also wozu sie zeitraubend erzeugen? Das Verfahren muss weniger arbeiten auf einer Tabelle, die nur ungerade Primkandidaten enthält, denn die Iteration $i = 2$ fällt weg. Sie ist die zeitlich längste $i$-Iteration. Es werden jetzt nur noch folgende Produkte erzeugt:



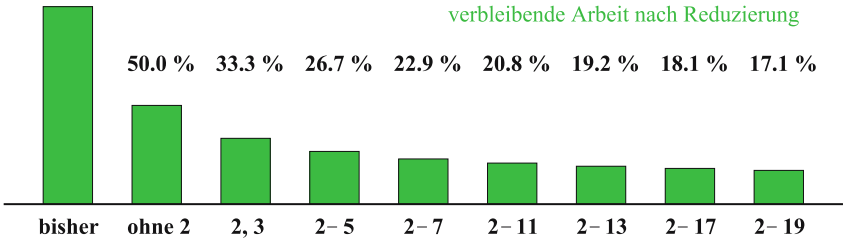

Diese Idee lässt sich weiter denken: Wir verzichten von vornherein auch auf alle echten Vielfachen von 3. Die Tabelle enthält nach der Initialisierung die Zahlen

2 3 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 ...

und die Streicharbeit geht mit der Iteration $i = 5$ los:




Auf diese Weise kann man immer kürzere Tabellen für denselben Zahlenbereich bis $n$ erhalten, indem man die echten Vielfachen von 5, 7, 11, 13 usw. weg lässt:




Diese Charakteristik zeigt sich in den Laufzeiten und, da die Tabelle selbst verkleinert wird, den Speichergrößen:

| | | ohne | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Reduzierung | keine | 2 | 2, 3 | 2–5 | 2–7 | 2–11 | 2–13 | 2–17 | 2–19 |
| Laufzeit [s] | 17.6 | 33.0 | 22.6 | 17.8 | 14.7 | 13.3 | **12.6** | 24.0 | 25.9 |
| Speicher [MByte] | 119.2 | 59.6 | 39.7 | 31.8 | 27.3 | 24.8 | 22.9 | 21.6 | 20.4 |

Hierbei wurde die fertige Tabelle dargestellt durch ein Array von Bits. An der Stelle $i$ im Array steht 1, wenn $i$ eine Primzahl ist, sonst 0.

Stellt man die Tabelle im Speicher als Liste der Primzahlen dar, so hat man dem gegenüber zwei Nachteile: Will man eine bestimmte Zahl auf ihre Primeigenschaft prüfen, muss man erst suchen, ob sie an „ihrem“ Platz in der Liste steht. Zum anderen werden die Zahlen bis zu 9-stellig und man braucht für alle 50.847.534 Primzahlen bis 1 Milliarde 1551.7 MByte Speicher.

Dass sich die Laufzeit nach dem Entfernen der geraden Zahlen gegenüber unserem Vorergebnis von 17.6 s fast verdoppelt, liegt daran, dass wir jetzt eine Umrechnung der Tabellenindizes $(1, 2, 3, 4, \ldots)$ auf die jeweils gemeinten Zahlen (im obigen Beispiel: $2, 3, 5, 7, 9, \ldots$) haben müssen, die etwas Zeit beansprucht. Insgesamt kommen wir jedoch auf ganz hervorragende 12.6 s bei einer reduzierten Tabelle ohne die Vielfachen von 2 bis 13. Danach steigt die Arbeit für die Vorbereitung der Umrechnungsdaten so stark an, dass sich weitere Reduzierung nicht mehr lohnt.

## Zum Weiterlesen

1. `http://de.wikipedia.org/wiki/Sieb_des_Eratosthenes`

   Der Wikipedia-Artikel bietet eine ganz konzentrierte Einführung in das Thema. Er wird im Lauf der Zeit möglicherweise ergänzt.

2. Auf der Homepage einer der Autoren steht ein Demoprogramm in `C` zur Verfügung, mit dem die tabellierten Laufzeiten gemessen wurden:
   `http://www.math.tu-berlin.de/~oellrich/schueler.html`

3. Kapitel 14 (Einwegfunktionen) und 16 (Public-Key-Kryptographie)

   In den Kapiteln 14 und 16 geht es nicht in erster Linie um Primzahlen. Jedoch reduzieren sich die behandelten Probleme im Wesentlichen darauf, wie schnell man die Teiler von sehr großen natürlichen Zahlen finden kann. Da Primzahlen keine echten Teiler besitzen, kann man keine solchen schnell finden und etwa durch Herausteilen das restliche Faktorisierungsproblem verkleinern. Große Primzahlen sind schwer als Teiler zu erkennen, daher eignen sie sich gut als Bausteine für Verschlüsselungsverfahren. Allerdings sind Primzahlen mit 100 oder mehr Stellen nicht mehr praktikabel mit Primzahltabellen zu handhaben. Hier kommen andere Methoden zum Einsatz, die gezielt Zahlen dieser Größenordnung erzeugen.

4. Kapitel 20 (Hashing)

   In Kap. 20 geht es auch nicht unmittelbar um Primzahlen. Jedoch sind Hash-Tabellen vorteilhaft, die die Länge einer Primzahl besitzen. Sie haben bei der Hashfunktion *Schlüssel modulo Tabellenlänge* i. A. recht gute Streueigenschaften, wenn die Schlüssel gleichverteilt sind. Bei einer bestimmten Suchmethode, dem so genannten *doppelten Hashing*, kann dadurch ein (evtl. gut „versteckter“) freier Platz sicher gefunden werden. Um eine passende Primzahl für eine Hash-Tabelle auszuwählen, ist eine Primzahlentabelle sehr nützlich.

5. Kapitel 21 (Fehlererkennende Codes)

   In Kap. 21 wird der Aufbau der Buchnummern ISBN erklärt. Da die Prüfsumme modulo 11 genommen wird und 11 eine Primzahl ist, können alle Einzelfehler und Ziffernvertauschungen erkannt werden. Solche Sicherungssysteme kann man prinzipiell mit beliebigen Primzahlen konstruieren, wobei eine Primzahltabelle helfen kann.

6. Ein *Primzahlenzwilling* ist ein Paar von Primzahlen, deren Differenz genau zwei ist. Die ersten Paare lauten: (3,5) (5,7) (11,13) (17,19) (29,31) (41,43) (59,61) (71,73) ... Das erste Paar (3,5) ist eine Ausnahme, denn dadurch steht zum einen die 5 als einzige Zahl in zwei Paaren, zum anderen ist die Zahl in der Mitte sonst immer durch 6 teilbar. Das muss so sein, weil von drei benachbarten Zahlen immer genau eine durch 3 teilbar ist und mindestens eine durch 2. Da die beiden Primzahlen (außer bei (3,5)) weder durch 2 noch 3 teilbar sind, besitzt die Zahl in der Mitte diese beiden Faktoren. Primzahlzwillinge wurden in höchsten untersuchten Höhen gefunden. Jedoch gibt es bis heute keinen Beweis, ob es endlich oder unendlich viele davon gibt. In einer Primzahltabelle bis $10^9$ kann man sich 3.424.506 Zwillinge ansehen.

# 14

# Einweg-Funktionen: Vorsicht Falle – Rückweg nur für Eingeweihte!

Rüdiger Reischuk und Markus Hinkelmann

Universität zu Lübeck

Die bisherigen Beiträge haben ein Problem schnell oder, wie man sagt, *effizient* gelöst. Wenn man nun für ein Problem keinen schnellen Algorithmus findet, ist dies eigentlich ein Grund, unzufrieden zu sein. Dieses Mal überlegen wir, wieso es manchmal von Vorteil ist, wenn ein Problem *keinen* schnellen Algorithmus besitzt. Unser heutiges Motto könnte also lauten:

*Auch schlechte Nachrichten können ihre guten Seiten haben.*

## Die Umkehrung des Multiplizierens: das Faktorisieren

In Kap. 11 haben wir gesehen, dass das Produkt von Zahlen recht schnell berechnet werden kann, auch wenn die Zahlen sehr groß sind und damit viele Ziffern für ihre Darstellung im Dezimal- oder Binärsystem benötigen. Selbst mit dem simplen Algorithmus, der in der Grundschule gelehrt wird, kann ein Mensch zwei größere Dezimalzahlen auf einem großen Blatt Papier in ein paar Minuten multiplizieren, auch wenn dies mühsam ist und Konzentration bedarf, um Rechenfehler zu vermeiden. Für einen Computer ist es überhaupt kein Problem, hundert- oder tausendziffrige Zahlen innerhalb von Sekundenbruchteilen zu multiplizieren oder zu dividieren.

Betrachten wir nun das umgekehrte Problem, ein Produkt wieder in seine Faktoren zu zerlegen. Man lernt in der Schule, dass es Primzahlen gibt, die man nicht mehr in Faktoren zerlegen kann. Jeder kennt die ersten Primzahlen: 2, 3, 5, 7, 11, 13, 17, 19, 23, ... Die Erfahrung zeigt, dass jede natürliche Zahl auf eindeutige Weise in Primfaktoren zerfällt wie beispielsweise

$$20518260 = 2 \cdot 2 \cdot 3 \cdot 5 \cdot 7 \cdot 7 \cdot 7 \cdot 997 \,,$$

und das kann man auch beweisen. Bei Primzahlen selber besteht dies Produkt nur aus einem Faktor. Hier ergibt sich in natürlicher Weise die Frage nach einem Algorithmus für das Problem, zu einer gegebenen großen Zahl

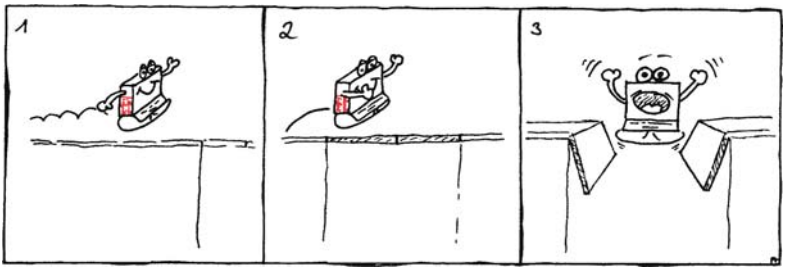



**Abb. 14.1.** Eine Einweg-Funktion

ihre Primfaktoren zu bestimmen, das ***Faktorisierungsproblem***. Geht das vielleicht auch so schnell wie das Multiplizieren?

Ob eine Dezimalzahl $n$ durch 2 bzw. 5 teilbar ist, erkennt man mit einem Blick an der letzten Ziffer; Teilbarkeit durch 3 überprüft man mit dem Quersummentest. Einen ähnlich einfachen Test gibt es auch noch für die Teilbarkeit durch 11. Das Faktorisieren von vielziffrigen Zahlen, die nur große Teiler besitzen, scheint dagegen sehr schwer zu sein. Man könnte $n$ durch jede Primzahl $p$ mit $p^2 \leq n$ zu teilen versuchen. Es ist bekannt, dass dies bei einer 100-stelligen Zahl ungefähr $8{,}5 \cdot 10^{48}$ Versuchs-Divisionen benötigt – das ist in etwa eine 1 mit 49 Nullen, eine absolut unvorstellbar große Zahl, so dass hieran nicht zu denken ist.

Als eine einfachere Variante des Faktorisierungsproblems können wir das ***Primzahlproblem*** ansehen, wo nur festzustellen ist, ob eine Zahl $n$ eine Primzahl oder Nichtprimzahl ist – man muß im zweiten Fall also nicht auch noch die Faktoren herausfinden. Für das Primzahlproblem gibt es effiziente Algorithmen, aber die sollen nicht unser heutiges Thema sein – siehe auch Kap. 13.

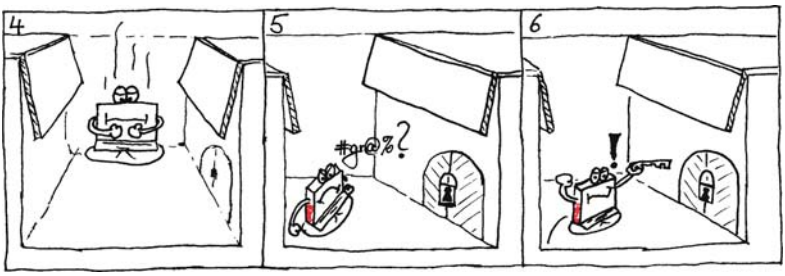



**Abb. 14.2.** Wie kommen wir wieder zurück?

In den 70er Jahren haben Ron Rivest, Adi Shamir und Leonard Adleman ihr inzwischen weit verbreitetes ***RSA-Kryptosystem*** erfunden, das Texte verschlüsselt, die man dann versenden kann – auch über so ein unsicheres Medium wie das Internet, so dass ein unbefugter Mithörer die Nachricht nicht verstehen kann. In dem System wird eine Zahl $n$ benutzt, die Produkt zweier sehr großer Primzahlen $p$ und $q$ ist. Man weiß, dass das *Knacken* dieses Verschlüsselungsverfahrens sehr eng zusammenhängt mit dem Problem, die beiden Faktoren $p$ und $q$ von $n$ herauszufinden.

Um zu demonstrieren, wie sicher ihr System ist, haben die Erfinder 1977 eine Nachricht mit einer 129-stelligen Dezimalzahl $n$ verschlüsselt und $n$ gemeinsam mit der verschlüsselten Nachricht veröffentlicht.

$$n = 114381.625757.888867.669235.779976.146612.010218.295721.242362.562561.842935.$$
$$706935.235733.897830.597123.563958.705058.989075.147599.290026.879543.541$$

Es dauerte unglaubliche 17 Jahre, bis 1994 sehr komplizierte Algorithmen entwickelt worden waren, die Intelligenteres tun als einfach alle möglichen Primteiler auszuprobieren, und dann noch 8 Monate Rechenzeit auf einem weltweiten Netz von Hunderten von Computern benötigten, um die beiden Faktoren

$$n = 3490.529510.847650.949147.849619.903898.133417.764638.493387.843990.820577$$
$$\times\, 32769.132993.266709.549961.998190.834461.413177.642967.992942.539798.288533$$

herauszufinden. Insgesamt wurden dabei etwa 160 Billiarden Computerbefehle ausgeführt, das sind $1,6 \cdot 10^{17}$. Wollt ihr wissen, wie die verschlüsselte Nachricht hieß? *"The magic words are squeamish ossifrage."*

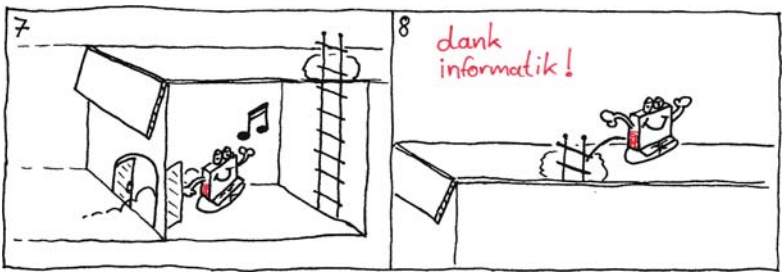



**Abb. 14.3.** Rückweg durch eine Geheimtür

## Einweg-Funktionen

Auch heutzutage, weitere 12 Jahre später, gilt weiterhin

- *die schlechte Nachricht:* Selbst mit den besten momentan verfügbaren Algorithmen und den schnellsten zur Zeit oder in den nächsten Jahren verfügbaren Superrechnern, auch mit Tausenden oder Zigtausenden von Computern kann man für Dezimalzahlen mit mehreren hundert Stellen, die nur sehr große Primfaktoren besitzen, diese Faktoren nicht herausfinden.
- *die gute Nachricht:* Solange keine dramatisch schnelleren Faktorisierungsalgorithmen zur Verfügung stehen, können Daten wie z.B. beim Online-Banking, die mit modernen kryptografischen Verfahren basierend auf dem RSA-Schema oder Ähnlichem verschlüsselt werden, als sicher angesehen werden.

Halten wir also fest:

*1.* Die Multiplikation natürlicher Zahlen, insbesondere von Primzahlen, läßt sich effizient berechnen,

*2.* ihre Umkehrung, die Zerlegung einer Zahl in ihre Primteiler, dagegen nicht – zumindest nach dem heutigen Erkenntnisstand der Informatik und Mathematik.

In der Mathematik versteht man unter dem Begriff Funktion eine Operation, die mathematische Objekte in andere derartige Objekte transformiert. Die Faktorisierung kann man als die zur Multiplikation umgekehrte Operation ansehen. Eine Operation, die leicht zu berechnen ist, aber eine schwierige Umkehroperation besitzt, wollen wir eine ***Einweg-Funktion*** nennen. Einweg-Funktionen haben für die Verschlüsselung von Texten eine wichtige Bedeutung. Die Verschlüsselung könnte mit Hilfe einer Einweg-Funktion geschehen. Denn diese Operation sollte schnell ausführbar sein; die Entschlüsselung, die Umkehroperation, dagegen sollte schwierig sein. Dies sind aber gerade die Eigenschaften, die von einer Einweg-Funktion verlangt werden. Ein unberechtigter Angreifer auf die verschlüsselten Daten hat damit praktisch keine Chance, aus dem verschlüsselten Text das ursprüngliche Dokument zu rekonstruieren.

Betrachten wir folgendes Beispiel. *Alice* möchte an *Bob* den Text „Ziemlich geheim“ schicken. Dabei sollen zur Vereinfachung nur Großbuchstaben verwendet werden. Ein Zwischenraum zwischen zwei Worten wird durch den Buchstaben „X“ gekennzeichnet. Nun verschlüsseln wir folgendermaßen: Die Buchstaben werden mit 01 bis 26 durchnummeriert, und jeder Buchstabe wird durch seine Nummer ersetzt. Aus

ZIEMLICHXGEHEIM

wird also

$$T = 26090513120903082407050805091 3.$$

Nun ist $T$ im Allgemeinen keine Primzahl, aber man kann immer durch Anhängen von ein paar zusätzlichen Ziffern – in diesem Fall z.B. 000257 – eine Primzahl

$$p = 260905131209030824070508050913\,000257$$

erhalten. Dann nimmt *Alice* eine weitere Primzahl, mit etwa ebenso vielen Ziffern, etwa

$$q = 673460865034106389254736738902736012787923$$

und bildet das Produkt

$$n = p \cdot q = 175709395355870926553760540843251469965 \\ 236378018754818950543391958678985496211\,.$$

Diese Zahl $n$ kann *Alice* an *Bob* schicken – niemand wird jetzt in der Lage sein, die Nachricht $p$ bzw. $T$ zu entschlüsseln. (In Wirklichkeit nimmt man 150 stellige Primzahlen, um den Grad der Sicherheit zu erhöhen.)

Aber halt ... Jetzt kann ja auch *Bob* die Nachricht nicht entschlüsseln! So ein Pech! Ganz so einfach ist die Verschlüsselung mit Einwegfunktionen also auch wieder nicht. Wir brauchen einen zusätzlichen Trick.

Eine **Einweg-Funktion mit geheimer Zusatzinformation**, in der Fachsprache spricht man auch von einer **Falltür-** oder **Geheimtür-Funktion**, besitzt die folgende zusätzliche Eigenschaft:

*3.* Die Umkehr-Funktion von $f$ kann effizient berechnet werden, wenn man im Besitz eines geheimen Schlüssels $S$ ist.

Hierfür gibt es ein ganz unmathematisches Beispiel aus dem täglichen Leben.

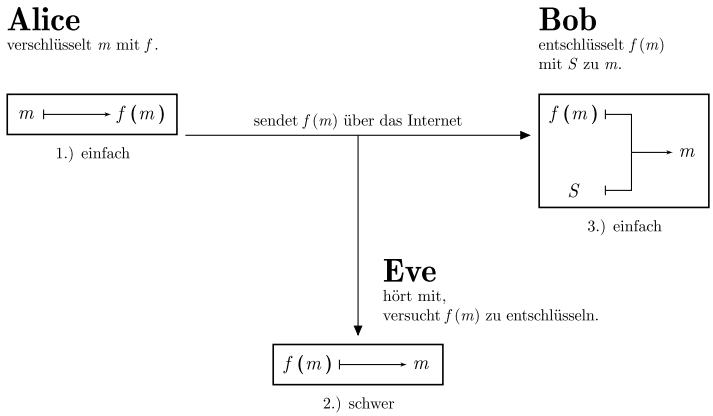


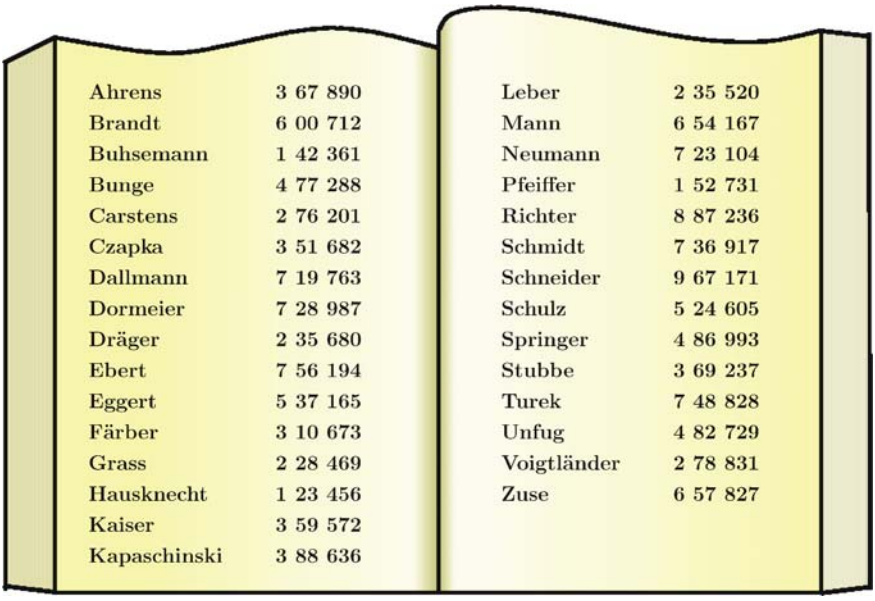

**Abb. 14.4.** Verwendung einer Falltür-Funktion in der Kryptologie

## Ein Problem der täglichen Praxis: Suchen im Telefonbuch

Gehen wir einen Schritt zurück in die Vergangenheit, als es noch keine elektronischen Dateien über Telefonanschlüsse gab, sondern nur das gute alte gedruckte Telefonbuch. Wenn *Alice* ihren alten Freund *Bob* nach langer Zeit wieder einmal anrufen möchte und seine Telefonnummer nicht mehr zur Hand hat, braucht sie nur im Telefonbuch nach *Bobs* Namen zu suchen und findet dahinter seine Telefonnummer. Das geht einfach und schnell – warum?

Erinnern wir uns an den ersten Beitrag in Kap. 1, die *binäre Suche.* Im aktuellen Telefonbuch der Hansestadt Lübeck mit circa $N = 250.000$ Einträgen (verteilt auf über 700 Seiten) beispielsweise kann man jeden Namen mit höchstens $18 \approx \log_2 N$ vielen Namensvergleichen finden, da die Teilnehmer in alphabetischer Reihenfolge aufgelistet werden. Wären sie dagegen ungeordnet, müßte man jedesmal das Telefonbuch solange durchlesen, bis man auf den gesuchten Namen gestoßen ist, was im Mittel die halbe Buchgröße ausmachen würde und bei 250.000 Einträgen absolut nicht praktikabel wäre. In der Realität wird allerdings die binäre Suche selten in Originalform angewandt – auch von Informatikern nicht, jeder macht es ein bisschen anders. Wir wissen jedenfalls alle aus Erfahrung: Die Suche vom Namen zur Telefonnummer ist einfach – aufgrund der alphabetischen Anordnung.

Wie sieht es mit der umgekehrten Suche aus, zu einer gegebenen Nummer den Teilnehmer zu finden?



| | | | |
|---|---|---|---|
| Ahrens | 3 67 890 | Leber | 2 35 520 |
| Brandt | 6 00 712 | Mann | 6 54 167 |
| Buhsemann | 1 42 361 | Neumann | 7 23 104 |
| Bunge | 4 77 288 | Pfeiffer | 1 52 731 |
| Carstens | 2 76 201 | Richter | 8 87 236 |
| Czapka | 3 51 682 | Schmidt | 7 36 917 |
| Dallmann | 7 19 763 | Schneider | 9 67 171 |
| Dormeier | 7 28 987 | Schulz | 5 24 605 |
| Dräger | 2 35 680 | Springer | 4 86 993 |
| Ebert | 7 56 194 | Stubbe | 3 69 237 |
| Eggert | 5 37 165 | Turek | 7 48 828 |
| Färber | 3 10 673 | Unfug | 4 82 729 |
| Grass | 2 28 469 | Voigtländer | 2 78 831 |
| Hausknecht | 1 23 456 | Zuse | 6 57 827 |
| Kaiser | 3 59 572 | | |
| Kapaschinski | 3 88 636 | | |

**Abb. 14.5.** Normales Telefonbuch

> *Alice* findet auf dem Display ihres Telefons die Nachricht, dass jemand mit der Nummer 123456 versucht hat, sie anzurufen. Sie kennt diese Nummer nicht, will aber nicht einfach zurückrufen, sondern erst wissen, wer sich hinter dieser Nummer verbirgt.

Mit einem gedruckten Telefonbuch bleibt ihr nichts anderes übrig, als das gesamte Telefonbuch Eintrag für Eintrag durchzukämmen, bis sie die Nummer entdeckt hat. Damit ist die Zuordnung *Name* $\rightarrow$ *Nummer* eine Art Einweg-Funktion, zumindest für Menschen, denen nur gedruckte Telefonbücher zur Verfügung stehen.

Computer dagegen können mit cleveren Algorithmen eine Folge von Datenpaaren sehr schnell sortieren, wie in vorangegangenen Beiträgen erläutert worden ist, so dass heutzutage manche Informationsanbieter elektronische Telefonbücher nach Nummern geordnet umsortieren. Mit Hilfe solch eines zusätzlichen *umgekehrten Telefonbuches* führt eine binäre Suche schnell zu dem gesuchten Namen. Im Computerzeitalter könnte man somit auch die Suche nach einem Namen effizient durchführen. Wir wollen jedoch noch einen kleinen Augenblick in der Situation verweilen, dass es nur gedruckte Telefonbücher gibt.

Eine kundenfreundliche Telefongesellschaft könnte *Alices* Problem dadurch lösen, dass sie Telefonnummern gemäß der alphabetischen Reihenfolge der Namen der Telefonbesitzer vergibt, d.h., *Alice* erhält eine Nummer $m_A$, die kleiner ist als *Bobs* Nummer $m_B$, da ihr Name alphabetisch vor dem von *Bob* kommt. In einem derartigen geordneten Telefonnummervergabesystem kann auch ein Mensch zu jeder Nummer den passenden Teilnehmer in wenigen Sekunden finden – vorausgesetzt er wendet die binäre Suche auf die geordnete Folge der Telefonnummern an.

Das Telefonbuch für ein geordnetes Telefonnummervergabesystem hat damit die Einweg-Eigenschaft verloren. Dies gefällt dem Datenschützer *Bob* jedoch nicht. Er besteht auf einer chaotischen Verteilung der Telefonnummern, um seine Telefonnummer nicht für jedermann so leicht zugänglich zu machen, und wendet sich deshalb an die GDK (Gesellschaft deutscher Kryptologen). Die GDK löst das Problem, indem jeder Teilnehmer eine neue Nummer bekommt, die sich durch eine geheim gehaltene Transformation $f$ errechnet, so dass die vormals geordnete Folge der Nummern

$$m_A \quad m_{\text{Andreas}} \quad m_{\text{Axel}} \ \ldots \qquad \text{von } \textit{Alice}, \textit{Andreas, Axel}, \ldots$$

nach der Transformation zu

$$f(m_A)\ f(m_{\text{Andreas}})\ f(m_{\text{Axel}})\ \ldots \qquad \text{vollständig chaotisch erscheint.}$$

Was kann man über dieses neue Telefonnummervergabesystem sagen? Im zugehörigen Telefonbuch sind die Teilnehmer alphabetisch geordnet aufgelistet und dies mit ihren neuen Nummern $f(m)$, die, wenn man $f$ nicht kennt, vollkommen willkürlich angeordnet zu sein scheinen. *Alice* bleibt also wiederum nichts anderes übrig, als das Telefonbuch linear zu durchsuchen. Die GDK dagegen kann mit Kenntnis der Transformation $f$ aus $f(m)$ die ursprüngliche

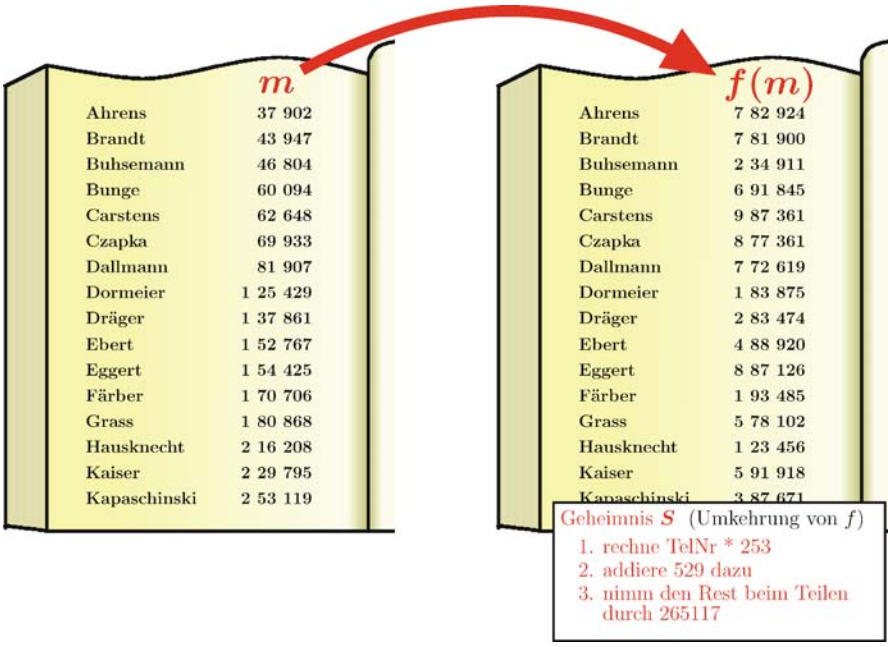



**Abb. 14.6.** Telefonbuch bei geordneter Nummernvergabe und Telefonbuch mit transformierten Nummern

Nummer im geordneten System zurückberechnen und dann eine binäre Suche auf den zurücktransformierten Nummern ausführen.

Die Zuordnung *Name* → *Nummer* im transformierten Telefonbuch ist somit eine Einweg-Funktion mit Geheimtür. Jeder kann zu einem Namen schnell die passende Nummer finden. Die Umkehrung kann nur derjenige effizient ausführen, der das Geheimnis $S$ kennt.

## Sicherheit und Googles

Kann man beweisen, dass es für ein konkretes Problem keine effizienten Algorithmen gibt? Intuitiv wird man vermuten, dass es schwieriger sein dürfte, die Nichtexistenz eines Objektes nachzuweisen als dessen Existenz: Dies gilt sicherlich bei der Suche nach schnellen Algorithmen, denn deren Zahl ist unendlich groß (vergleiche hierzu die Diskussion in Kap. 24).

Um letzte Zweifel an der Sicherheit moderner Kryptosysteme zu zerstreuen, müßte jedoch der Nachweis erbracht werden, dass es für die Umkehr-Funktion des Verschlüsselns keine effizienten Verfahren gibt. Seit über 30 Jahren sucht die Informatik nach Methoden, um zu beweisen, dass bestimmte Probleme keine schnellen Algorithmen besitzen. Obwohl dabei schon große

**Tabelle 14.1.** Zahlenwerte geschätzt, gerundet und ohne Gewähr

| **Bezugsgröße** | **Aufwand/Anzahl** |
|---|---|
| Verwendung des momentan schnellsten Algorithmus und aller heutigen Rechner auf der Erde zur | |
| Faktorisierung einer 256-stelligen Dezimalzahl | mehr als 2 Monate |
| Faktorisierung einer 512-stelligen Dezimalzahl | mehr als 10 Mill. Jahre |
| Faktorisierung einer 1024-stelligen Dezimalzahl | mehr als $10^{18}$ Jahre |
| Lebensdauer des Universums | $\approx 10^{11}$ Jahre |
| Taktzyklen eines 5 GHz Prozessors in 1 Jahr | $\approx 1{,}6 \cdot 10^{17}$ |
| Anzahl Elektronen im Universum | $\approx 8 \cdot 10^{77}$ |
| Anzahl 100-stellige Primzahlen | $\approx 1{,}8 \cdot 10^{97}$ |

Meilensteine erreicht worden sind, scheint der Weg bis zu diesem Ziel noch weit und stellt eine der großen Herausforderungen der modernen Informatik dar.

Zum Abschluß wollen wir dem Leser noch ein Gefühl für Größenordnungen vermitteln. Die folgende Tabelle enthält Abschätzungen für den Zeitaufwand, eine Zahl, die sich aus 2 großen Primzahlen zusammensetzt, zu faktorisieren. Im unteren Teil haben wir einige markante riesengroße Zahlen (so genannte Googles) aufgelistet.

## Zum Weiterlesen

1. Kapitel 16 (Public-Key-Kryptographie)

   Kapitel 16 beschäftigt sich mit Public-Key-Kryptosystemen, für die Einweg-Funktionen eine wichtige Voraussetzung sind.

2. Das RSA-System wird näher beschrieben unter anderem in Infoserversecurity: `http://www.infoserversecurity.org/itsec_infoserver_v0.5/sections/links/1074005790/index_html`

   Für weitere Informationen zum Brechen des RSA-Schemas siehe: `http://www.wired.com/wired/archive/4.03/crackers.html`

3. Wer mehr über Kryptographie und ihre Historie erfahren möchte, dem empfehlen wir folgende Bücher:

   - A. Beutelspacher: *Moderne Verfahren der Kryptologie*. Vieweg, 7. Auflage, 2005.

- A. Beutelspacher, H. Neumann, T. Schwarzpaul: *Kryptopgrafie in Theorie und Praxis*. Vieweg, 2005.
- F. Bauer: *Entzifferte Geheimnisse*. Springer, 1997.

4. Interessantes über Primzahlen kann man unter anderem finden bei C. Caldwell: The Prime Pages:
   `http://primes.utm.edu/`
5. Das Faktorisierungsproblem wird diskutiert in R. Weis, S. Lucks: Bigger is better!:
   `http://www.cryptolabs.org/rsa/WeisLucksDatenschleuderFaktor.html`
6. Eine empfehlenswerte Einführung, wo uns im heutigen IT-Zeitalter Algorithmen überall begegnen, gibt J. Gallenbacher in *Abenteuer Informatik – IT zum Anfassen von Routenplaner bis Online-Banking*, Spektrum Akademischer Verlag, 2007. Insbesondere empfehlen wir Kap. 9 über das Thema Sicherheit.

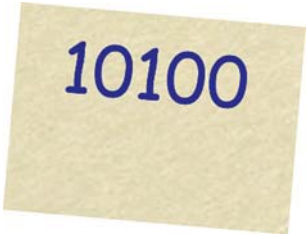
# 15

# Der One-Time-Pad-Algorithmus: Der einfachste und sicherste Verschlüsselungsalgorithmus

Till Tantau

Universität zu Lübeck

Die Schule hat kaum angefangen und schon wieder steht eine Informatikklausur an – und Max hat keine Ahnung vom Stoff, irgendwas über Verschlüsselung. Dass Max keine Ahnung hat, ist auch nicht weiter verwunderlich, da er sich im Unterricht und auch sonst hauptsächlich mit Lisa beschäftigt, seiner neuen Freundin. Lisa hingegen findet nicht nur Max, sondern auch Verschlüsselung faszinierend. Deshalb möchte Max, man ahnt es schon, die Klausur von Lisa abschreiben. Lisa sieht allerdings ein kleines Problem: „Peter wird während der Klausur zwischen uns sitzen. Ich muss also Peter einen Zettel mit den Antworten geben und der gibt ihn dann dir. Das sollte aber kein Problem sein, da die Lehrerin das eh nicht merkt, Peter alles macht, was ich ihm sage, und es nur Ankreuzfragen sind: Ich schreibe eine 1 auf, wenn man ein Kreuz machen muss, und eine 0, wenn man keines machen darf. Wenn du also bei einer Ankreuzaufgabe mit fünf Möglichkeiten beim ersten und dritten Kästchen ein Kreuz machen sollst, dann schicke ich dir folgenden Papierschnipsel:“




Gesagt, getan. Und tatsächlich schreiben Lisa, Max und auch Peter hervorragende Klausuren. Die Lehrerin ist allerdings ob der sonstigen Leistungen von Max nicht ganz sicher, ob alles mit rechten Dingen zugegangen ist. Für die nächste Klausur beschließt sie deshalb, statt Peter die Ex-Freundin von Max zwischen Max und Lisa zu setzen. Die Idee der Lehrerin zeigt Wirkung, denn Max meint aufgeregt zu Lisa: „Lieber falle ich durch die Klausur als dass *sie* die Antworten auch abschreibt!“

Lisa denkt kurz nach und meint dann: „Ok, dann müssen wir eben die Lösung mit einem One-Time-Pad verschlüsseln.“

„One-Time-Pad?“ fragt Max etwas ratlos.

„Das bedeutet wörtlich ‚Einmal-Notizblock‘ und ist ein Verfahren zur Einmalverschlüsselung.“

„Einmalverschlüsseln?“ fragt Max immer noch ratlos.

„Du passt echt nicht auf. . .“ beginnt Lisa, worauf Max sie mit einem „Aber dafür liebe ich dich!“ unterbricht, was Lisa ignoriert. „Verschlüsseln bedeutet, dass wir uns einen *Schlüssel* ausdenken, mit dem ich die Lösung für die Aufgaben mit einem One-Time-Pad abschließe. Du kannst die verschlüsselte Lösung mit dem Schlüssel wieder aufschließen. Und deine Ex kann mit der verschlüsselten Lösung ohne den Schlüssel nichts anfangen.“

„Häh?“ entgegnet Max nun völlig verwirrt.

## Verschlüsselung von Nachrichten

„Gib mal fünf Münzen her,“ bittet Lisa, was Max auch macht, obwohl man ihn undeutlich so etwas wie „Die will ich aber wiederhaben“ grummeln hört. Lisa legt die Münzen auf den Tisch und fährt fort: „Die Zahl-Seite bedeutet, dass du ein Kreuz machen musst in einem Kästchen, die andere Seite bedeutet, dass du kein Kreuz machen sollst. Wenn du also bei einer Ankreuzaufgabe mit fünf Möglichkeiten beim ersten und dritten Kästchen ein Kreuz machen sollst, dann können wir das mit Münzen wie folgt darstellen:“

„Von mir aus, auch wenn ich nicht sehe, was das bringen soll,“ entgegnet Max etwas gelangweilt. „Egal, ob du mir nun 10100 auf einen Zettel schreibst oder die fünf Münzen hinlegst, meine Ex kapiert doch auch, dass das bedeuten soll: Mache Kreuze bei der ersten und dritten Frage.“

„Richtig, aber jetzt kommt die Verschlüsselung ins Spiel. Gib mir mal einen Stapel Zettel – Danke. Auf einige schreibe ich ‚umdrehen‘, auf die anderen ‚nicht umdrehen‘. Jetzt darfst du fünf Zettel zufällig wählen und nebeneinander legen.“

Max tut wie ihm geheißen und legt unter die Münzen Folgendes:

„Gut," meint Lisa. „Diese Zettel bilden unseren ‚Schlüssel', den wir vor der Klausur festlegen und auswendig lernen."

„Ah," entgegnet Max, der ja auch nicht blöd ist, „du schickst dann statt der ursprünglichen Münzen die entsprechend unseres Schlüssels umgedrehten Münzen, also Folgendes:

Wenn meine Ex das sieht, dann kann sie damit herzlich wenig anfangen – wenn sie zum Beispiel die erste Münze betrachtet, dann kann das sowohl ‚mache ein Kreuz' bedeuten oder eben gerade ‚mache kein Kreuz'. Die Chance, dass wir in unserem Schlüssel gerade ‚umdrehen' stehen haben, ist ja fifty-fifty.

Aber – warum sollte sie mir dann überhaupt den Zettel weiterreichen und nicht gleich zerreißen oder runterschlucken oder verbrennen oder ..."

„Sie will doch auch, dass du versetzt wirst – damit sie dich auch nächstes Jahr durch ihre Anwesenheit ärgern kann."

„Na toll... Jetzt aber nochmal der Schlachtplan: In der Klausur löst du die erste Aufgabe und möchtest mir mitteilen, dass ich im ersten und dritten Kästchen ein Kreuz machen soll. Das entspricht 10100. Da aber unser Schlüssel sagt, dass die erste und vierte ‚Münze' umgedreht werden sollen, schickst du mir stattdessen


Ich habe mir auch den Schlüssel gemerkt und ‚drehe die Münzen wieder zurück' und erhalte 10100. Schließlich mache ich meine Kreuze im ersten und dritten Kästchen."

„Ich wusste doch, dass ich einen schlauen Freund habe. Wie du siehst ist dieses One-Time-Pad-Verfahren nicht schwierig und trotzdem sehr sicher. Wir sind auch nicht die einzigen, die es benutzen – wenn sich Staatsoberhäupter Nachrichten schicken und nicht wollen, dass jemand die Nachrichten mitlesen kann, dann benutzen sie auch dieses Verfahren."

Jetzt muss Max lachen. „Glaubst du wirklich, wenn Herr Bush an Frau Merkel eine Nachricht schicken will, dann dreht er Münzen herum und schreibt Nullen und Einsen auf eine Postkarte?!"

„Natürlich nicht," grummelt Lisa, „das macht ein Computer für ihn. Dazu muss man das Verfahren als Algorithmus aufschreiben, was für dich eine gute Übung wäre."

## Der Algorithmus

„Mal sehen," beginnt Max, der immer noch grinst. „Erstmal scheint mir, dass es tatsächlich nur einen Algorithmus gibt, denn die zur Verschlüsselung und zur Entschlüsselung sind gleich: Jedesmal beginnen wir mit einem Array von Nullen und Einsen und einem Schlüssel; heraus bekommen wir wieder einen Array von Nullen und Einsen. Dann muss einfach im Algorithmus dort jede Null in eine Eins und umgekehrt verwandelt werden, wo im Schlüssel ‚umdrehen' steht."

„Und wie speicherst du den Schlüssel? Du kannst ja keine gelben Zettel im Computerspeicher haben."

„Da nehme ich auch einen Array und in dem stehen eben die Zeichenketten ‚umdrehen' und ‚nicht umdrehen'. Dann lautet der Algorithmus wie folgt:"

Der Algorithmus OneTimePad führt eine Verschlüsselung oder Entschlüsselung des Arrays $A$ mit $n$ Einträgen mittels *Schlüssel* durch.

```
1  procedure OneTimePad (A, Schlüssel)
2  begin
3      for i := 1 to n do
4          if Schlüssel[i] = „umdrehen" then
5              if A[i] = 0 then
6                  A[i] := 1
7              else
8                  A[i] := 0
9      endfor
10 end
```

„Genau," entgegnet Lisa. „So könnte man das machen. Normalerweise ist der Schlüssel allerdings nicht ein Array, in dem Zeichenketten wie ‚umdrehen' oder ‚nicht umdrehen' stehen, sondern ebenfalls ein Array von Nullen und Einsen. Dabei bedeutet eine Eins gerade ‚umdrehen' und eine Null ‚nicht umdrehen'. Damit kann man den Algorithmus auch ganz kurz schreiben:"

Kurze Version von OneTimePad.

```
1  procedure OneTimePad (A, Schlüssel)
2  begin
3      for i := 1 to n do
4          A[i] := A[i] xor Schlüssel[i]
5      endfor
6  end
```

„Moment," unterbricht Max, „ich erinnere mich dunkel, dass ‚xor' für eXklusives OdeR steht. Was war das nochmal?"

„Das exklusive Oder überprüft, ob genau eine von zwei Zahlen eine 1 ist, und liefert eine 1, wenn dies der Fall ist. Damit das exklusive Oder also eine 1 ist, muss $A[i]$ eine 1 sein oder $Schlüssel[i]$ eine 1 sein, aber eine dieser Bedingungen muss ‚exklusiv' gelten – es darf gerade nicht beides gelten. Schau dir folgende Tabelle an, die zeigt, was passiert."

Tabelle mit den Werten von $A[i]$ xor $Schlüssel[i]$.

| | $Schlüssel[i] = 0$ | $Schlüssel[i] = 1$ |
|---|---|---|
| $A[i] = 0$ | 0 | 1 |
| $A[i] = 1$ | 1 | 0 |

„Ich glaube, ich verstehe, was hier passiert. Das ‚xor' macht genau, was wir brauchen: Es dreht den Wert von $A[i]$ um, wenn $Schlüssel[i] = 1$ ist, und lässt ihn so wie er ist, wenn $Schlüssel[i] = 0$ gilt."

## Brechen der Verschlüsselung

„So langsam gefällt mir dieses Verfahren," fährt Max fort. „Und das beste ist, ich brauche mir nur fünf gelbe Zettel zu merken für die gesamte Klausur, denn wir können ja die Verschlüsselung immer wieder verwenden!"

„Nein, so einfach ist das leider nicht," wendet Lisa ein. „Stelle dir mal vor, wir würden für alle zwanzig Aufgaben in der Klausur, jede mit fünf Kästchen, jedesmal denselben Schlüssel verwenden. Was würde denn passieren, wenn deine Ex eine der Aufgaben selber lösen würde und dann meine verschlüsselte Lösung sehen würde? Nehmen wir an, ich finde heraus, dass du die letzten drei Kästchen ankreuzen musst. Dann lautet die Lösung ‚in Münzen geschrieben':

Diesen Wert kennt deine Ex dann auch, wenn sie die Aufgabe lösen kann. Dann sieht sie den Zettel, den sie von mir bekommt, auf dem der verschlüsselte Wert 01010 steht. Dies entspricht den Münzen

Siehst du, was jetzt passiert?“

„Ja, sie kann den Schlüssel ausrechnen. Da du auf den Zettel eine 0 an die erste Stelle geschrieben hast, weiß sie, dass du die erste Münze nicht umgedreht hast. Die zweite hingegen hast du umgedreht und so weiter. Sie weiß dann, dass der Schlüssel lautet:


Und wenn sie erstmal den Schlüssel kennt, dann kann sie alle anderen Aufgabe auch lösen!“

„Deshalb heißt das Verfahren auch One-Time-Pad, also Einmalverschlüsselung. *Man kann einen Schlüssel bei diesem Verfahren nur einmal verwenden.* Wenn man das nicht macht, so kann man das Verfahren schnell umgehen. So benutzt beispielsweise ein älteres Sicherheitsverfahren beim schnurlosen Surfen mit Laptops in Cafés immer wieder denselben Schlüssel – weshalb der Schlüssel schnell herauszubekommen ist und man selbst im Nebenzimmer noch mitlesen kann, welche E-Mails du mir schickst.“ Max wird nacheinander kreidebleich, dann knallrot. Lisa lächelt und fährt fort: „Die hochbezahlten Leute, die sich dieses Verfahren ausgedacht haben, haben offenbar in der Schule nicht sonderlich aufgepasst.

Wir werden uns für die Klausur für jede Aufgabe einen neuen Schlüssel ausdenken müssen.“

„Aber das ist ja schrecklich! Da muss ich mir ja die richtige Reihenfolge von 100 mal ‚umdrehen‘ oder ‚nicht umdrehen‘ merken. Da kann ich ja gleich den Stoff lernen!“

„Hmm, vielleicht wäre das sowieso die beste Lösung. So schwierig ist Verschlüsselung ja nun auch wieder nicht.“

## Zum Weiterlesen

1. Kapitel 16 (Public-Key-Kryptographie)

   Hier geht es um ein Verfahren, das mit Schlüsseln arbeitet, die nicht nur einmal verwendet werden können (wie beim One-Time-Pad) sondern mehrfach. Die Schlüssel werden dabei sogar öffentlich bekannt gegeben!

2. Kapitel 17 (Teilen von Geheimnissen)

   Manche Geheimnisse (z. B. Schlüssel bei Verschlüsselungsverfahren) sind zu wichtig, als dass man sie einfach herumliegen lassen könnte. Dieser Artikel beschreibt Tricks, wie man Geheimnisse so in Teile aufspaltet, dass man schon alle Teile auf einmal finden oder stehlen muss, um das Geheimnis zu lüften.

3. Kapitel 25 (Zufallszahlen)

   Lisa und Max können ihren Schlüssel zufällig aus einem Stapel von Zetteln ziehen. Aber wie macht das ein Computer, wenn er selber einen Schlüssel erzeugen muss? – schließlich ist in einem Algorithmus eigentlich nichts zufällig.

4. Kapitel 21 (Fehlererkennende Codes)

   Das Verfahren von Lisa und Max ist ziemlich störungsanfällig (z. B. wenn Max oder Lisa sich einen Zettel nicht richtig gemerkt hat). In diesem Artikel wird beschrieben, wie sie solche Fehler aufdecken und sogar korrigieren können.

5. `http://de.wikipedia.org/wiki/Verschlüsselung`

   Dieser Wikipedia-Artikel ist ein leicht zugänglicher Anfangspunkt für alle, die mehr erfahren möchten über Verschlüsselung.

6. `http://de.wikipedia.org/wiki/One-Time-Pad`

   In diesem Wikipedia-Artikel werden One-Time-Pads speziell beschrieben, insbesondere deren Geschichte und die vielen möglichen Varianten.

7. Friedrich L. Bauer: *Entzifferte Geheimnisse*. Springer-Verlag, 3. Auflage, 2000.

   Dieses Buch liefert einen Überblick über das weite Feld der Ver- und Entschlüsselung. Es empfiehlt sich für alle, die wirklich in die Materie einsteigen wollen und sich von mathematischen Formeln nicht abschrecken lassen.

# 16

# Public-Key-Kryptographie

Dirk Bongartz und Walter Unger

RWTH Aachen

Wer wollte nicht schon mal eine Geheimnachricht übermitteln? Sogar Cäsar hat das schon gemacht. Angeblich verschob er einfach jeden Buchstaben seiner Nachricht im Alphabet um drei Positionen weiter nach rechts. Aus einem A wird bei diesem Verfahren ein D, aus einem B ein E usw. und schließlich aus einem W ein Z, aus einem X ein A, aus einem Y ein B und aus einem Z ein C.

Wenn man das Verfahren kennt, dann kann man eine abgefangene „Geheimnachricht“ sicherlich leicht entschlüsseln. Was verbirgt sich beispielsweise hinter der folgenden Nachricht?

Verschlüsselung nach Cäsar:

„DOJRULWKPXV GHU ZRFKH“
Welcher Text wurde hier verschlüsselt?

Wenn man es etwas allgemeiner haben möchte, kann man natürlich auch eine Zahl $k$ ($< 26$) wählen und dann die Buchstaben der Nachricht, die man gerne verschlüsseln möchte (man nennt sie auch *Klartext*), jeweils um $k$ Positionen im Alphabet nach rechts verschieben. Auf diese Weise erhalten wir dann wieder eine verschlüsselte Nachricht, den so genannten *Geheimtext*[1]. Das *Verschlüsselungsverfahren* ist hier das Verschieben der Buchstaben im Alphabet, und $k$ wird als *(geheimer) Schlüssel* bezeichnet. Wenn man nun wieder entschlüsseln möchte, dann muss man nur die Buchstaben im Geheimtext jeweils um $k$ Positionen im Alphabet nach links verschieben.

Sobald bei einem solchen Verfahren jemand den Schlüssel kennt, dann kann er sowohl verschlüsseln als auch entschlüsseln. Deshalb nennt man diese Verfahren auch *symmetrische Verfahren*. Das One-Time-Pad aus Kap. 15 ist übrigens auch ein symmetrisches Verfahren.

[1] Dieser wird auch als *Krypttext* oder *Chiffre* bezeichnet

Das heißt, jeder, der eine Nachricht verschlüsseln kann, kann andere Nachrichten, die mit dem gleichen Verfahren und mit dem gleichen Schlüssel verschlüsselt wurden, auch entschlüsseln.

Wir werden im nächsten Abschnitt sehen, dass dies nicht immer gilt. Es bringt sogar Vorteile wenn die Schlüssel nicht gleichwertig sind. Solche Verfahren werden als *asymmetrische Verfahren* bezeichnet.

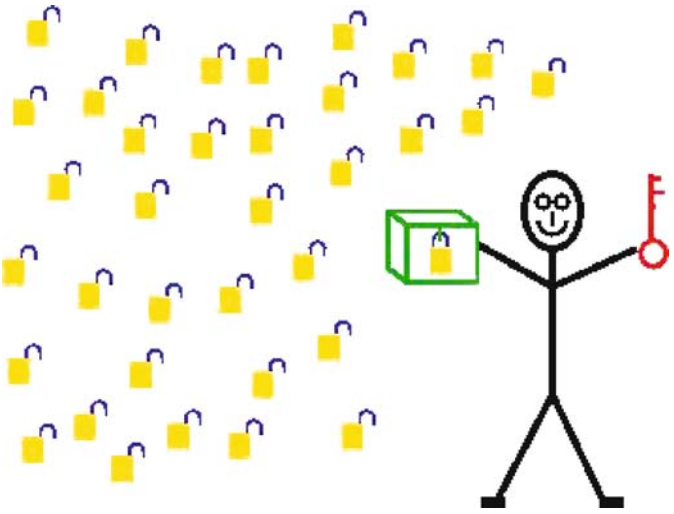
## Öffentliche Schlüssel (Public-Keys)

Wenn man diese Überschrift liest, dann erscheint sie vielleicht zunächst unsinnig. Das kann doch gar nicht klappen – das haben wir doch oben beim Cäsar-Verfahren gesehen, oder?! Denn wie sollte ein Verschlüsselungsverfahren funktionieren, bei dem der Schlüssel *öffentlich* ist?

Aber das erscheint nur im ersten Augenblick so. Es bräuchte ja nur die *Verschlüsselung* mit einem öffentlichen Schlüssel durchgeführt zu werden. Der Schlüssel zur Entschlüsselung könnte weiterhin geheim bleiben.

Wäre es also nicht super, wenn jeder euch eine Geheimnachricht schicken könnte, aber nur ihr selbst sie entschlüsseln könntet.

Eigentlich ist das doch gar nicht so schwer. Stellt euch vor, ihr würdet Tausende von Vorhängeschlössern kaufen, die aber alle mit dem gleichen Schlüssel geöffnet werden können (ok, so etwas gibt es normalerweise nicht im Laden zu kaufen – da gibt es vermutlich eher *ein* Schloss mit *mehreren* Schlüsseln – aber wir können ja mal so tun als ob). Dann verteilt ihr die geöffneten Schlösser in eurer Schule, z. B. im Sekretariat, in der Bibliothek, usw. Zum Schließen der Schlösser braucht man keinen Schlüssel – sie schnappen einfach zu! Den Schlüssel behaltet ihr.



Dann kann jeder, der euch eine geheime Nachricht schicken will, eine Kiste nehmen, die geheime Nachricht dort hineinlegen, die Kiste mit einem eurer Vorhängeschlösser verschließen und sie ruhig dem größten Klatschmaul der ganzen Schule mitgeben, das sie euch überbringt. Keiner außer euch wird die geheime Botschaft lesen können.

Nun ja, es funktioniert also. Aber wir haben schon einen ziemlichen Aufwand betrieben – so viele Schlösser herzustellen und zu verteilen, dürfte wohl ziemlich teuer werden. Kann man das Problem denn nicht ohne Kisten und Vorhängeschlösser lösen? Ja, kann man. Dazu helfen uns die Einwegfunktionen aus Kap. 14. Hierbei handelt es sich um Funktionen, bei denen die Berechnung des Funktionswertes „einfach“, die andere Richtung, also die Umkehrfunktion, aber schwer zu berechnen war (wenn wir das mal so salopp zusammenfassen). Bei uns bräuchten wir also etwas, mit dem Verschlüsseln einfach ist (damit alle eine Nachricht für uns verschlüsseln können) und nur das Entschlüsseln schwer. Allerdings sollten wir selbst natürlich entschlüsseln können – also benötigen wir noch so etwas wie eine *Hintertür*, damit es funktioniert.

Man kann hier beispielsweise auch das inverse Telefonbuch aus Kap. 14 verwenden. Wir benutzen hier aber eine andere Idee.

## Eine eingeschränkte Mathematik

Public-Key-Verfahren, die bei der Verschlüsselung von Nachrichten am Computer verwendet werden, benutzen eine relativ komplexe Mathematik, die wir hier nicht beschreiben möchten. Wir können das Prinzip aber sehr gut mit einer eingeschränkten Mathematik illustrieren.

Diese eingeschränkte Mathematik kennt nur Addition, Subtraktion und Multiplikation auf ganzen Zahlen. In dieser eingeschränkten Mathematik gibt es insbesondere niemanden, der dividieren kann. Versetzt euch einfach in die Zeit zurück, wo ihr gerade die Multiplikation in der Schule gelernt hattet und noch nicht wusstet, wie dividiert wird. Und mit dieser eingeschränkten Mathematik beschreiben wir nun, wie Simone eine Nachricht an Eike schickt. Diese Nachricht wird aus einer Zahl bestehen.

Das Verfahren wird in drei Schritten beschrieben. Das Erzeugen der beiden Schlüssel (öffentlich und privat), das Verschlüsseln und das Entschlüsseln.

### Aufbau der Schlüssel

Zuerst brauchen wir einen Schlüssel. Um genau zu sein, brauchen wir zwei Schlüssel, einen geheimen und einen öffentlichen. Die Nachricht soll von Simone an Eike geschickt werden. Daher braucht Eike den geheimen Schlüssel, um die Nachricht zu entschlüsseln. Dazu denkt sich Eike zwei Zahlen aus und multipliziert diese. Die erste Zahl ist der *private Schlüssel*, die zweite bildet dann zusammen mit dem Produkt den *öffentlichen Schlüssel*. Damit besteht

der private Schlüssel aus einer Zahl und der öffentliche Schlüssel aus zwei Zahlen (nämlich dem öffentlichen Faktor und dem öffentlichen Produkt).

Der öffentliche und private Schlüssel

| p | *privater Schlüssel* |
|---|---|
| 11 | *öffentlicher Faktor* |
| 143 | *öffentliches Produkt* |

Da ihr ja dividieren könnt, ist es für euch nun einfach, den privaten Schlüssel zu bestimmen: Er lautet $p = 143/11 = 13$. Aber wenn man von unserer eingeschränkten Mathematik ausgeht, so ist dieser private Schlüssel weiterhin geheim.

An das schwarze Brett der Schule hängt Eike nun einen Zettel. Jeder kann ihn lesen, da aber keiner dividieren kann, bleibt trotzdem der private Schlüssel $p$ von Eike geheim.


**Das Verschlüsseln**

Simone liest diesen Zettel und will Eike eine Nachricht schicken. Das ist der Termin der nächsten Fete, der 5. Dezember 2006. Dabei besteht die Nachricht nur aus der 5, denn Simone macht immer im Dezember eine Fete und das wissen schon alle.

Simone beginnt nun mit der Verschlüsselung. Sie kennt die öffentlichen Zahlen 11 und 143 von Eike und die zu verschlüsselnde Nachricht 5.

Zusätzlich zu der zu verschlüsselnden Nachricht denkt sich Simone noch eine geheime Zahl aus, diese Zahl nennen wir *Sendegeheimnis*. Damit rechnet sie nun die verschlüsselten Daten aus. Die verschlüsselten Daten bestehen aus zwei Zahlen, der *verschlüsselten Nachricht* und einer *Entschlüsselungshilfe*.

Simone bestimmt das Produkt aus dem *Sendegeheimnis* und dem *öffentlichen Produkt* von Eike. Um ihre Nachricht zu verschlüsseln, addiert Simone

dieses Produkt nun einfach zur Nachricht hinzu. Wenn Simone als Sendegeheimnis die Zahl 3 gewählt hat, dann ergibt sich als *verschlüsselte Nachricht*: $5 + 3 \cdot 143 = 434$. Diese 434 würde veröffentlicht werden, aber die 3 muss geheim bleiben. Ansonsten könnte jeder ausrechnen: $434 - 3 \cdot 143 = 5$.

Da nun die 3 als Sendegeheimnis ja schon bekannt ist, wählt Simone ein anderes Sendegeheimnis, nennen wir es einfach $s$. Simone rechnet die *verschlüsselte Nachricht* aus:

Berechnung der verschlüsselten Nachricht:

$$5 + s \cdot 143 = 1292$$

Die 1292 wird veröffentlicht werden, aber das Sendegeheimnis $s$ wird geheim bleiben.

Wenn nur Simone das *Sendegeheimnis* kennt, ist es keinem möglich, aus der *verschlüsselten Nachricht* die Nachricht zu bestimmen. Wie kann nun aber Eike entschlüsseln? Denn auch Eike kennt das Sendegeheimnis nicht. Damit Eike, und auch nur Eike, entschlüsseln kann, berechnet Simone als *Entschlüsselungshilfe* noch das Produkt aus *Sendegeheimnis* und *öffentlichem Faktor*.

Simone berechnet $11 \cdot s = 99$. Diese Zahl wird auch veröffentlicht.

Berechnung der Entschlüsselungshilfe:

$$11 \cdot s = 99$$

Simone geht zum schwarzen Brett der Schule und hängt da folgenden Zettel aus.


Damit sind die folgenden Zahlen allen bekannt, denn jeder konnte die beiden Zettel am schwarzen Brett lesen.

Die öffentlichen Werte nach dem Verschlüsseln

| | | |
|---|---|---|
| 11 | *öffentlicher Faktor* von Eike | |
| 143 | *öffentliches Produkt* von Eike | $11 \cdot p$ |
| 1292 | *verschlüsselte Nachricht* | $5 + s \cdot 143$ |
| 99 | *Entschlüsselungshilfe* | $s \cdot 11$ |

Selbst wenn jeder weiß, wie Simone gerechnet hat, müsste man wieder dividieren können, um aus den Zahlen das *Sendegeheimnis* $s$ oder den *privaten Schlüssel* $p$ zu bestimmen. Und ohne das *Sendegeheimnis* kann auch niemand Simones geheime Nachricht herausbekommen.

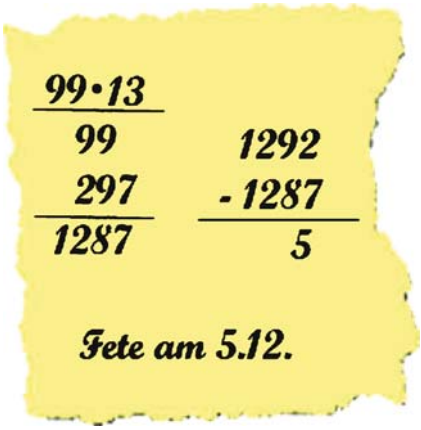
### Das Entschlüsseln

Nun will Eike die Nachricht von Simone entschlüsseln. Eike kann wie alle anderen nicht dividieren. Eike kennt aber ihren *privaten Schlüssel* $p$. Schauen wir uns nun die verschlüsselte Nachricht genauer an.

Die verschlüsselte Nachricht

| | |
|---|---|
| 1292 | *Nachricht + Sendegeheimnis · 143* |
| = | *Nachricht + Sendegeheimnis · öffentliches Produkt* |
| = | *Nachricht + Sendegeheimnis · öffentlicher Faktor · privater Schlüssel* |
| = | *Nachricht + Entschlüsselungshilfe · privater Schlüssel* |

Damit kann nun Eike durch folgende Rechnung an die Nachricht gelangen.



Die Entschlüsselung:

$$1292 - 99 \cdot (\textit{privater Schlüssel } p) = 5$$

Wie man sieht, hat Eike keine Division gebraucht.

### Der Lauscher

In Spionage- oder Agentenfilmen sieht man oft, wie Telefonate und andere Gespräche belauscht werden. Um so etwas zu verhindern, werden in den Filmen sichere Leitungen benutzt. Aber dann muss man sich darauf verlassen, dass genau diese Leitung auch wirklich sicher ist.

Hier haben wir keine sichere Leitung gebraucht. Als Leitung wurde das schwarze Brett verwendet. Wir alle waren die Lauscher. Wir haben alle Nachrichten zwischen Eike und Simone mitgelesen. Aber trotzdem war es uns – in der eingeschränkten Mathematik – nicht möglich, die geheime Nachricht zu entschlüsseln. Eike und Simone mussten nur jeweils eine Zahl geheim halten. Auch brauchten sie kein gemeinsames Geheimnis. Sie mussten sich nie treffen, um einen Schlüssel, wie z. B. beim One-Time-Pad (siehe Kap. 15), auszutauschen.

Wenn also in einem Film eine sichere Leitung benutzt wird, dann stelle man sich wohl besser eine Leitungsverbindung vor, die mit unserem Trick und weiterer Technik verschlüsselt worden ist.

### Ohne eingeschränkte Mathematik

Solange keiner dividieren kann, ist das Verfahren also sicher. Aber spätestens mit dem Schulabschluss weiß man, dass es Menschen und Programme gibt, die dividieren können. Wer auch noch gut aufgepasst hat, sieht, wie man mit Hilfe eines der bereits vorgestellten Verfahren das Ergebnis der Division auch recht schnell bestimmen kann (siehe Kap. 1). Bei diesem Verfahren wird aber ausgenutzt, dass der Bruch $a/b$ kleiner wird, wenn $b$ größer wird. Nur wegen dieser Eigenschaft kann man bei der binären Suche bestimmen, in welchem Bereich weiter gesucht werden muss.

Falls man aber nur mit Resten rechnen würde, dann schlüge die binäre Suche fehl. Wie man mit Resten rechnet, seht ihr im nächsten Abschnitt. Da man aber auch mit diesen Resten gut dividieren kann, wird es nicht ausreichen, nur mit Resten zu rechen, man muss die Division durch etwas anderes ersetzen.

## Das Verfahren von ElGamal

Es gibt aber weitere Operationen in der Mathematik. Und diese benutzt man statt Addition, Subtraktion und Multiplikation. Zum Beispiel wird folgendes gemacht.

- Statt der Addition wird die Modulare Multiplikation benutzt.
  (Die Modulare Multiplikation ist eine Multiplikation, welche auf Restklassen arbeitet. Betrachte dazu auch den Algorithmus aus Kap. 17, in dem Modulare Addition zum Teilen von Geheimnissen verwendet wird.)
- Statt der Subtraktion wird die Modulare Division benutzt.
- Statt der Multiplikation wird die Modulare Exponentiation benutzt.
- Statt der Division wird der Modulare Logarithmus betrachtet.

Dieses Verfahren ist unter dem Namen *ElGamal*-Verschlüsselung bekannt. Bisher kennt man keine Verfahren, die den Modularen Logarithmus einer großen Zahl (mit mehr als 1000 Stellen) schnell bestimmen können. Alle bisher bekannten Verfahren würden dazu auf den schnellsten Rechnern Jahrhunderte benötigen. Selbst wenn diese also irgendwann die geheime Nachricht entschlüsseln, wird die Fete längst vorbei sein. 

### Die Modulare Exponentiation

Wir müssen uns die Modulare Exponentiation genauer betrachten, denn diese muss schnell zu berechnen sein. Zur Einleitung wollen wir uns kurz die Modulare Multiplikation ansehen.

Beteiligt bei diesen Rechnungen ist immer eine Primzahl $p$, d. h. eine Zahl, die nur durch $p$ und 1 teilbar ist (z. B. $2, 3, 5, 7, 11, 13, 17, 19, \ldots$). Wenn man nun $(a \cdot b)$ mod $p$ rechnet, so ist das Ergebnis der Rest der Division von $(a \cdot b)$ geteilt durch $p$. Das Ergebnis von $(5 \cdot 8)$ mod 17 ist daher 6, denn $(5 \cdot 8)/17$ ist 2 Rest 6. Die Menge der Zahlen, die als möglicher Rest bei der Division durch $p$ entstehen können, bezeichnen wir als Restklasse. Mit diesen Restklassen kann man nun fast genau so rechnen wie bei einer normalen Multiplikation.

Nun kommen wir zu der Modularen Exponentiation. Bei der Modularen Exponentiation wird eine Zahl $a$ genau $b$ mal mit sich selbst multipliziert und jeweils der Rest bei der Division durch $p$ als Ergebnis genommen. Diese wird als $(a^b)$ mod $p$ geschrieben.

Das Ergebnis von $(3^9) \bmod 17$ ist 14, denn es gilt:

**Berechnung von $(3^9) \bmod 17$**

$$
\begin{aligned}
3^9 \bmod 17 &= (3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3) \bmod 17 \\
&= ((3^8 \bmod 17) \cdot 3) \bmod 17 \\
3^8 \bmod 17 &= ((3^4 \bmod 17) \cdot (3^4 \bmod 17)) \bmod 17 \\
3^4 \bmod 17 &= ((3^2 \bmod 17) \cdot (3^2 \bmod 17)) \bmod 17 \\
3^2 \bmod 17 &= (3 \cdot 3) \bmod 17 \\
&= 9 \\
3^4 \bmod 17 &= (9 \cdot 9) \bmod 17 \\
&= 13 \\
3^9 \bmod 17 &= (13 \cdot 13 \cdot 3) \bmod 17 \\
&= 14
\end{aligned}
$$

Hier sieht man schon, dass man zum Ausrechnen von $3^9$ nicht 8 Multiplikationen benötigt, sondern nur 4. Dazu versucht man möglichst viele Multiplikationen zusammenzufassen. Damit ergibt sich die folgende Beschreibung.

**Berechnung von $(a^b) \bmod p$**

1. Falls $b = 0$ ist, dann ist das Ergebnis 1.
2. Falls $b = 1$ ist, dann ist das Ergebnis $a$.
3. Falls $b$ ungerade ist, dann ist das Ergebnis $(a^{b-1} \cdot a) \bmod p$.
4. Da keiner der bisherigen Fälle aufgetreten ist, ist $b$ gerade und man rechne:
5. $\quad h = (a^{b/2}) \bmod p$.
6. $\quad$ Das Ergebnis ist $(h \cdot h) \bmod p$.

In dieser Beschreibung benutzen wir zum Berechnen von $(a^b) \bmod p$ weitere Werte von $(c^d) \bmod p$. Dabei waren aber die benutzten Werte $c$ und $d$ immer viel kleiner als die Werte $a$ und $b$. Daher funktioniert dieses Verfahren. Daraus ergibt sich der folgende formale rekursive Algorithmus:

**Rekursiver Algorithmus zur Berechnung von $(a^b) \bmod p$**

```
1   ExpMod(a, b, p)
2     If b = 0 then return 1.
3     If b = 1 then return a.
4     If b is odd then
5       begin
6         h = ExpMod(a, b − 1, p)
7         return (h · a) mod p
8       end
9     h = ExpMod(a, b/2, p)
10    return (h · h) mod p
```

In der folgenden Tabelle geben wir mal alle Werte von $2^b \bmod 59$ für gerade $b$ an. Daran sieht man schon, wie unregelmäßig diese Werte im Vergleich zur Multiplikation sind.

Werte für $2^b \bmod 59$

| $b$ | $2^b \bmod 59$ | $b$ | $2^b \bmod 59$ | $b$ | $2^b \bmod 59$ |
|---|---|---|---|---|---|
| 0 | 1 | 20 | 28 | 40 | 17 |
| 2 | 4 | 22 | 53 | 42 | 9 |
| 4 | 16 | 24 | 35 | 44 | 36 |
| 6 | 5 | 26 | 22 | 46 | 26 |
| 8 | 20 | 28 | 29 | 48 | 45 |
| 10 | 21 | 30 | 57 | 50 | 3 |
| 12 | 25 | 32 | 51 | 52 | 12 |
| 14 | 41 | 34 | 27 | 54 | 48 |
| 16 | 46 | 36 | 49 | 56 | 15 |
| 18 | 7 | 38 | 19 | | |

Beim Modularen Logarithmus ist nun zu einer Zahl, z. B. der 42, das passende $b$ zu finden mit $2^b \bmod 59 = 42$. Da kann man nur ausprobieren. Das ist sehr aufwändig, vor allem wenn die Zahlen sehr groß werden. Aber auch wenn die Zahlen so groß sind, kann man immer noch schnell die Modulare Exponentiation durchführen, denn bei dem obigen Verfahren werden die Zahlen sehr schnell kleiner.

Wenn der Exponent $b$ zehn Stellen hat, dann sind zur Lösung des Modularen Logarithmus mindestens eine Million Möglichkeiten zu betrachten. Aber das Berechnen einer Modularen Exponentiation mit einem Exponenten mit zehn Stellen braucht höchstens 65 Modulare Multiplikationen. Hier sieht man schon den Unterschied in der Komplexität zwischen Modularem Logarithmus und Modularer Exponentiation. Bei den Verfahren von ElGamal werden heutzutage Zahlen mit mindestens Tausend Stellen betrachtet. Da ist dieser Unterschied noch größer.

### Beschreibung des Verfahrens von ElGamal

Damit kann nun direkt das Verfahren von ElGamal angegeben werden. Es ist eine relativ einfache Übertragung des Verfahrens mit der eingeschränkten Mathematik.

Im obigen Verfahren wurde das Assoziativgesetz $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ ausgenutzt. Bei der ElGamal-Verschlüsselung brauchen wir ein ähnliches Gesetz für das Rechnen mit Potenzen. Die Schreibweise $a^x$ bedeutet, dass $a$ $x$ mal multipliziert wird. Ohne auf die genauen Beweise einzugehen, geben wir hier das analoge Gesetz an: $(g^{a \cdot b})^c = g^{(a \cdot b) \cdot c} = g^{a \cdot (b \cdot c)} = (g^a)^{b \cdot c}$. Daher brauchen wir hier noch die Zahl $g$, die gemeinsame Basis. Dabei ist es wichtig, dass alle Zahlen zwischen 1 und $p - 1$ erzeugt werden können. Es muss also für jede

Zahl $i$ zwischen 1 und $p-1$ eine Zahl $j$ geben mit $i = g^j \bmod p$. Daher wird $g$ auch als *Generator* bezeichnet, denn $g$ generiert die ganze Restklasse. In der folgenden Tabelle sieht man, dass 4 kein *Generator* für die Restklasse mod 7 ist, aber 3 und 5.

Generatoren der Restklasse mod 7: 3 und 5

| $i$ | $3^i \bmod 7$ | $i$ | $4^i \bmod 7$ | $i$ | $5^i \bmod 7$ |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 3 | 1 | 4 | 1 | 5 |
| 2 | 2 | 2 | 2 | 2 | 4 |
| 3 | 6 | 3 | 1 | 3 | 6 |
| 4 | 4 | 4 | 4 | 4 | 2 |
| 5 | 5 | 5 | 2 | 5 | 3 |
| 6 | 1 | 6 | 1 | 6 | 1 |

Eike – sie ist in der Zwischenzeit schon eine Klasse höher – bestimmt zuerst eine Primzahl $p$ und eine weitere Zahl $g$, den *Generator*. Für dieses Beispiel sei die Primzahl 59 und 2 der *Generator*. Dann bestimmt Eike ihren *privaten Schlüssel* $x$. Sie berechnet den ersten Teil des *öffentlichen Schlüssels* durch $y = (g^x) \bmod p$, d. h. $42 = (2^x) \bmod 59$. Sie veröffentlicht nun die Zahlen $p = 59$, $g = 3$ und $y = 42$ am schwarzen Brett der Schule. Da die Zahl 42 nicht in der obigen Tabelle als Ergebnis auftritt, hat Eike also einen privaten Schlüssel $x$, der ungerade ist. Ihr könnt ja mal versuchen, diesen zu finden.

Die ElGamal Werte von Eike:

| | |
|---|---|
| 59 | Primzahl von Eike |
| 2 | *Generator* von Eike |
| $x$ | *privates Geheimnis* von Eike |
| 42 | *öffentlicher Schlüssel* von Eike |

Simone liest diese drei Zahlen, um die geheime Nachricht 15 an Eike zu senden. Dieses Jahr ist die Fete 10 Tage später. Als Sendegeheimnis wählt Simone nun 9. Sie rechnet: $a = 2^9 \bmod p$, d. h. $a = 40$. Weiter rechnet sie $b = (15 \cdot 42^9) \bmod 59$, d. h. $b = 38$. Sie veröffentlicht am schwarzen Brett diese beiden Zahlen.

Die ElGamal Werte von Simone:

| | |
|---|---|
| 40 | *Entschlüsselungshilfe* für Eike |
| 38 | *Nachricht* für Eike |

Eike fängt nun an, zu entschlüsseln. Sie rechnet daher nun: $h = 40^x \bmod 59$. Das Ergebnis dieser Rechnung ist 34. Nun gilt $34 \cdot 33 \bmod 59$ ist 1. Das heißt, 33 ist das Inverse von 34 im Restklassenring von 59.

Diese Inversen wollen wir etwas genauer beschreiben. Bei der Addition ist das Inverse einer Zahl $x$ die Zahl $-x$, denn $x + (-x) = 0$. Bei einer Multiplikation ist das Inverse der Zahl $x$ die Zahl $\frac{1}{x}$, denn $x \cdot \frac{1}{x} = 1$. Bei einer Modularen Multiplikation ist dann das Inverse zu einer Zahl $x$ eine Zahl $y$, wenn $(x \cdot y) \bmod p = 1$ gilt.

Nun ergibt sich das Geheimnis als $38 \cdot 33 \bmod 59$. Damit erhält Eike 15 als Ergebnis der Entschlüsselung.

In der bisherigen Beschreibung haben wir es vermieden, den privaten Schlüssel $x$ von Eike zu verraten. Ihr könnt diese Zahl ja mal suchen. Dabei wird man sehen, dass dies sogar bei diesen kleinen Zahlen schon kompliziert ist.

### Weitere Verfahren

Viele Public-Key-Systeme benutzen dieses Verfahren. Dabei werden oft noch andere, kompliziertere Operationen verwendet, z. B. elliptische Kurven oder sogar hyper-elliptische Kurven.

## Sicherheit

Man fragt sich natürlich: Wie sicher sind die Verfahren? Zum einen muss man darauf achten, dass die verwendeten Zahlen groß genug sind. Wenn man zu kleine Zahlen wählt, dann kann es sein, dass ein Programm durch Ausprobieren den Schlüssel findet. Wenn nun der Schlüssel sehr groß ist, dann dauert das Ausprobieren auch sehr lange, z. B. Jahrhunderte. Damit ist das Geheimnis – z. B. der Termin der Fete – ausreichend geschützt.

Kann es sein, dass jemand plötzlich ein Verfahren findet, den Modularen Logarithmus zu bestimmen? Bisher kann man noch nicht beweisen, dass es ein solches Verfahren nicht gibt. Aber schon seit vielen Jahren versuchen Mathematiker und Informatiker für dieses Problem eine Lösung zu finden. Und bisher ist trotz aller Bemühungen keine Lösung gefunden worden. Man geht davon aus, dass es ein solches Verfahren nicht gibt.

Wenn aber plötzlich doch ein schnelles Verfahren zur Lösung des Modularen Logarithmus bekannt wird, dann ist die Verschlüsselung nach ElGamal nicht mehr sicher. Wir hoffen, dass das nicht passiert. Der obige Trick lässt sich aber dann immer noch mit anderen Operationen anwenden.

## Zum Weiterlesen

1. Das Verfahren von ElGamal ist auch auf Wikipedia zu finden:
   `http://de.wikipedia.org/wiki/Elgamal-Kryptosystem`

2. Kapitel 14 (Einweg-Funktionen)

   In diesem Kapitel wird ein weiteres Public-Key Verfahren, das RSA Verfahren, erwähnt.

3. `http://www.matheprisma.uni-wuppertal.de`

   Hier sind Beschreibungen zu RSA, Enigma und Cäsar zu finden.

4. Es gibt viele Bücher, die eine Einführung zur Verschlüsselung und deren Geschichte beschreiben. Einige davon sind:

   - Klaus Schmeh: *Die Welt der geheimen Zeichen. Die faszinierende Geschichte der Verschlüsselung.* W3l, 2. Auflage, 2007.
   - Rudolf Kippenhahn: *Verschlüsselte Botschaften. Geheimschrift, Enigma und Chipkarte.* Rowohlt Tb., 4. Auflage, 1999.
   - Albrecht Beutelspacher: *Geheimsprachen. Geschichte und Techniken.* Beck, 4. Auflage, 2005.
   - Dietmar Wätjen: *Kryptographie. Grundlagen, Algorithmen, Protokolle.* Spektrum Akademischer Verlag, 1. Auflage, 2003.
   - Simon Singh: *Geheime Botschaften. Die Kunst der Verschlüsselung von der Antike bis in die Zeiten des Internet.* Dtv, 1. Auflage, 2001.

# 17

# Teilen von Geheimnissen

Johannes Blömer

Universität Paderborn

In vielen Filmen und Romanen wie „Die Piratenbraut“, „Der Schatz im Silbersee“ oder „Der Schuh des Manitu“ taucht immer wieder das folgende Motiv auf. Ein Teil einer Schatzkarte wird gefunden. Um den Schatz zu bergen, genügt aber dieser Teil der Karte alleine nicht. Vielmehr werden hierzu auch alle anderen Teile der Karte benötigt. Also begibt sich der Finder des Schatzkartenfragments auf die Suche nach den restlichen Teilen. Die Besitzer der anderen Teile sind aber natürlich genauso brennend an den ihnen fehlenden Teilen der Karte interessiert. Und schon kann das Abenteuer beginnen.

Dieses Film- und Romanmotiv ist ein Beispiel für das Problem, das wir in diesem Kapitel vorstellen: Teilen von Geheimnissen. Dabei interessiert uns die Frage, wie man überhaupt Schatzkarten oder beliebige andere Informationen so in Teile zerlegen kann, dass ohne Kenntnis aller Teile der Schatz nicht gefunden werden kann bzw. die Informationen nicht vollständig rekonstruiert werden können. Wir werden dabei Methoden zum Teilen von Geheimnissen kennen lernen, die deutlich besser sind als das Zerschneiden einer Schatzkarte. Denn es ist nicht wirklich überzeugend, dass man nur mit Hilfe aller Teile einer Karte das Versteck des Schatzes finden kann.

Das allgemeine Problem ist leicht beschrieben. Ein Geheimnis, wir nennen es $G$, soll in eine bestimmte Anzahl von Teilen zerlegt werden. Die einzelnen Teile werden dann an unterschiedliche Personen verteilt, wobei auf folgende Punkte geachtet wird:

1. Wenn alle Personen wieder zusammenkommen und ihre Teile des Geheimnisses kombinieren, so können sie das Geheimnis $G$ vollständig rekonstruieren.
2. Wenn jedoch nur einige Personen, aber nicht alle, zusammenkommen, so können sie das Geheimnis $G$ nicht vollständig rekonstruieren. Mehr noch, diese Personen können dann keine oder nur sehr wenige Informationen über $G$ aufdecken.

Neben seiner Rolle als Filmmotiv hat das Teilen von Geheimnissen andere, ernstere und realistische Anwendungen. Stellen wir uns etwa vor, dass

ein wichtiges Dokument eines Staates oder eines Unternehmens in einem Safe gelagert ist. Man hat sich vorher darauf geeinigt, dass dieses Dokument nur dann aus dem Safe genommen und veröffentlicht werden darf, wenn alle Mitglieder einer eigens hierzu eingesetzten Kommission oder eines Gremiums der Veröffentlichung zustimmen. Um dieses zu erreichen, wird der Safe nun mit verschiedenen Schlössern gesichert, genau ein Schloss für jedes Kommissionsmitglied. Jedes Mitglied der Kommission hat den Schlüssel zu genau einem Schloss. Soll der Safe geöffnet werden, muss jedes Kommissionsmitglied sein Schloss öffnen und auf diese Weise der Veröffentlichung des Dokuments zustimmen.

Mit Hilfe des Teilens von Geheimnissen können wir ebenfalls erreichen, dass das Dokument nur mit Zustimmung aller Mitglieder der Kommission veröffentlicht werden kann. Dazu werden wir den Safe nicht mehr durch mehrere Schlösser sichern und jedem Kommissionsmitglied einen Schlüssel übergeben. Stattdessen sichern wir den Safe durch ein einziges Nummernschloss dessen Geheimnummer z. B. 50 Dezimalstellen besitzt. Jetzt wird die Geheimzahl zur Öffnung des Safes einfach in so viele Teile aufgeteilt, wie die Kommission Mitglieder hat. Jedes Kommissionsmitglied bekommt dann genau ein Teilgeheimnis. Wenn alle Kommissionsmitglieder sich einig sind, dass der Safe geöffnet werden und das Dokument veröffentlicht werden soll, können sie ihre Teilgeheimnisse nutzen, um die Geheimzahl zu rekonstruieren. Die Teilgeheimnisse kann man sich also wie Schlüssel für verschiedene Schlösser vorstellen, mit denen der Safe gesichert ist. Mit dem Teilen von Geheimnissen können somit physische Schlüssel eines Safes durch geheime, nur einzelnen Personen zugängliche Informationen ersetzt werden.

Neben dem Verteilen der Geheimzahl eines Safes gibt es noch viele andere Beispiele für Anwendungen des Teilens von Geheimnissen. Es ist sogar so, dass das Teilen von Geheimnissen, wie wir es gleich kennen lernen werden, eines der wichtigsten Hilfsmittel der Kryptographie ist, also der Wissenschaft vom Verschlüsseln von Nachrichten oder, allgemeiner, der Wissenschaft vom Schutz von Informationen vor unbefugtem Zugriff und Änderung. Denn kombiniert man die Methoden zum Teilen von Geheimnissen mit der *Public-Key Kryptographie* (siehe Kap. 16), so kann man nicht nur Schlüssel, sondern auch Safes und Schlösser durch Algorithmen und Informationen ersetzen. Dann können elektronische Daten so verschlüsselt werden, dass die Daten nur dann entschlüsselt werden können, wenn, wie in unserem Beispiel von oben, alle Mitglieder einer Kommission ihr Teilgeheimnis beisteuern. Die Teilgeheimnisse sind dabei die Teile des geheimen Schlüssels eines Public-Key-Verfahrens.

## Eine einfache Methode zum Teilen von Geheimnissen

Wie aber können wir nun Geheimnisse teilen? Wie können wir Schlösser und die dazugehörigen Schlüssel durch Informationen ersetzen, die jeweils nur ein

Mitglied der Kommission kennt? Betrachten wir wieder unser Beispiel des in einem Safe eingeschlossenen Dokuments. Die 50-stellige Geheimzahl des Safes sei etwa

$$G = 65497\ \ 62526\ \ 79759\ \ 79230\ \ 86739\ \ 20671\ \ 67416\ \ 07104\ \ 96409\ \ 84628.$$

Nehmen wir weiter an, dass dieses Geheimnis unter genau 10 Personen so aufgeteilt werden soll, dass nur alle 10 Personen zusammen die Geheimzahl rekonstruieren können. Wie wäre es mit der Idee, wie in der Abbildung unten, jeder Person genau 5 der 50 Stellen zu geben?

Wir sehen schnell, dass das keine gute Idee ist. Denn kommen nur 9 der 10 Personen zusammen, so sollten diese 9 Personen ja eigentlich nichts oder zumindest nicht sehr viel über die Geheimzahl erfahren. Nun kennen sie aber bereits 45 der 50 Stellen der Geheimzahl. Für jede einzelne dieser 9 Personen hat sich die Anzahl der ihr unbekannten Stellen von 45 auf 5 Stellen reduziert. Musste vorher jede Personen $10^{45}$ Möglichkeiten für die ihr unbekannten Stellen ausprobieren, um das gesamte Geheimnis zu bestimmen, reduziert sich die Anzahl Möglichkeiten durch die Zusammenarbeit der 9 Personen auf $10^5 = 10000$. Um uns den Informationsgewinn deutlich zu machen, stellen wir uns vor, man könnte in einer Sekunde überprüfen, ob eine 50-stellige Zahl die gesuchte Geheimzahl ist. Um die $10^5 = 10000$ noch möglichen Geheimzahlen auszuprobieren, benötigen die 9 zusammen arbeitenden Personen dann noch etwa 3 Stunden für die Bestimmung der gesuchten Geheimzahl. Denn in einer Stunden mit 3600 Sekunden können auch 3600 mögliche Geheimzahlen getestet werden. Damit können dann 9 Personen auch ohne die Zustimmung der 10. Person in relativ kurzer Zeit den Safe öffnen und das Dokument veröffentlichen. Muss hingegen eine Person noch $10^{45}$ Geheimnzahlen ausprobieren, so würde dieses bei einem Aufwand von einer Sekunde pro möglicher Geheimzahl über $10^{35}$ *Jahre* dauern, weitaus länger als das Universum bislang existiert und weitaus länger als es vermutlich insgesamt existieren wird.

Der nächste Versuch bringt uns unserem Ziel des Teilens von Geheimnissen schon etwas näher. Um das Geheimnis $G$, die Geheimzahl mit 50 Dezimalstellen, auf die 10 Mitglieder der Kommission aufzuteilen, wählen wir 10 zufällige Zahlen, die alle größer als Null sind und deren Summe genau $G$ ergibt. Die Teilgeheimnisse sind dann die 10 zufällig gewählten Zahlen. Betrachten wir ein kleines Beispiel, in dem $G$ eine ganze Zahl ist und die Teilgeheimnisse Zahlen zwischen 1 und 50 sind. Außerdem soll das Geheimnis nur auf 4 Mitglieder aufgeteilt werden. Sei jetzt $G = 129$. Dann können die Teilgeheimnisse

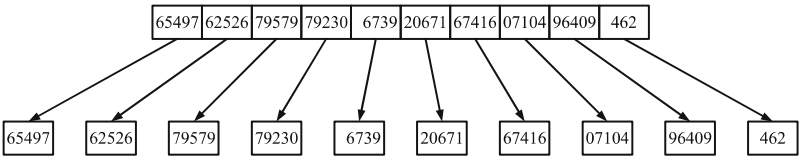



**Abb. 17.1.** Ein einfaches Beispiel für das Teilen von Geheimnissen

z. B. 17, 47, 31 und 34 sein, denn $17+47+31+34 = 129$. In diesem Verfahren ist klar, dass mit Kenntnis aller Teilgeheimnisse das Geheimnis $G$ bestimmt werden kann. Aber es gibt ein etwas größeres Problem: Wie können die Teilgeheimnisse so gewählt werden, dass nur ein Teil der Kommissionsmitglieder nichts oder wenig über das Geheimnis $G$ erfährt? Leider ist dieses Problem in diesem Verfahren auch nicht so ohne weiteres zu lösen. So erfahren in unserem Beispiel die ersten drei Teilnehmer aus ihren Geheimnissen, dass das Geheimnis $G$ zwischen $17 + 47 + 31 = 95$ und 200 liegt. Sie haben also die Möglichkeiten für das Geheimnis fast halbiert. Mit Hilfe eines kleinen Tricks können wir jedoch das Verfahren so ändern, dass nur alle Kommissionsmitglieder zusammen das Geheimnis rekonstruieren können, während nichts über das Geheimnis verraten wird, wenn sich nicht alle Kommissionsmitglieder an der Rekonstruktion beteiligen. Der Trick besteht darin, die *Division mit Rest* zu verwenden.

Das Verfahren zum Teilen von Geheimnissen sieht dann folgendermaßen aus. Nehmen wir an, das zu teilende Geheimnis $G$ ist eine Zahl zwischen 0 und einer großen Zahl $N$. In unserem Beispiel der 50-stelligen Dezimalzahl ist $N = 10^{50}$. Dieses Geheimnis soll weiterhin auf 10 Personen aufgeteilt werden, wobei das Verfahren allerdings für jede beliebige Anzahl von Personen funktioniert. Wir gehen in zwei Schritten vor:

1. Zunächst wählen wir 9 zufällige Zahlen zwischen 0 und $N-1$. Wir nennen diese Zahlen $t_1, t_2, \ldots, t_9$. Diese Zahlen sind die Teilgeheimnisse der ersten 9 Personen.
2. Um das 10. Teilgeheimnis $t_{10}$ zu bestimmen, bilden wir zunächst $t_1+\ldots+t_9$ und dividieren diese Summe durch $N$. Von dieser Division nehmen wir den Rest $R$ und bilden die Differenz $G-R$. Falls $G-R$ positiv ist, so ist dies unser gesuchtes $t_{10}$. Falls $G-R$ negativ ist, so ist $G-R+N$ unser $t_{10}$. Durch diese Vorschrift gilt, dass $t_1+t_2+\ldots+t_{10}$ bei Division durch $N$ den Rest $G$ ergibt.

Betrachten wir ein kleines Beispiel, in dem wir die Werte der Teilgeheimnisse leicht per Hand ausrechnen können. Wir wählen $N = 53$ und $m = 4$. Das zu teilende Geheimnis sei 23.

1. Wir wählen nun zunächst die ersten drei Teile des Geheimnisses. Seien diese 17, 47 und 31.
2. Um das vierte Teilgeheimnis zu bestimmen, berechnen wir zunächst die Summe der ersten drei Teilgeheimnisse, also $17+47+31 = 95$. Wir addieren nun zu 95 noch 34 hinzu und erhalten 129. Bei Division mit Rest von 129 durch 53 erhalten wir 23, das Geheimnis. Als viertes Teilgeheimnis müssen wir daher 34 wählen.

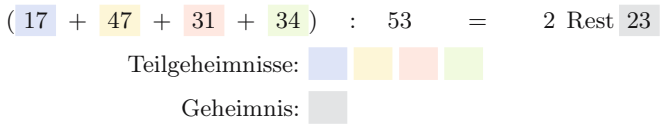



**Abb. 17.2.** Beispiel für das Teilen von Geheimnissen durch Division mit Rest

Dargestellt ist unser Verfahren in Abb. 17.2.

Liefert dieses Verfahren uns auch die gewünschten Eigenschaften? Betrachten wir unser Beispiel. Kommen alle vier Besitzer der Teilgeheimnisse zusammen, so können sie die Summe ihrer Teile berechnen und dann den Rest bei Division mit $N = 53$ bestimmen. Sie erhalten zunächst als Summe den Wert 129 und dann als Rest bei Division mit 53 den Wert 23, also genau das Geheimnis. Dasselbe Verfahren funktioniert auch im allgemeinen Fall. Denn das Geheimnis selber ist immer der Rest bei Division durch $N$ der Summe der Teilgeheimnisse. Damit können alle Personen zusammen das Geheimnis rekonstruieren.

Was passiert nun, wenn nicht alle Personen zusammenkommen? Zunächst einmal sieht es so aus, als ob wir den letzten Teilnehmer etwas anders behandelt haben als die übrigen Teilnehmer, denn sein Teilgeheimnis hängt von den anderen Teilgeheimnissen ab, während dieses bei den ersten Teilgeheimnissen nicht der Fall zu sein scheint. Bei genauerem Hinsehen stellt man aber fest, dass dieser Eindruck täuscht. Gehen wir wieder zu unserem Beispiel und nehmen wir das erste Teilgeheimnis 17. Betrachten wir die Summe der übrigen Teilgeheimnisse, so erhalten wir $47 + 31 + 34 = 112$. Bestimmen wir nun die eindeutige Zahl $x$, so dass $112 + x$ bei Division mit Rest durch 53 den Wert 23 liefert, so erhalten wir $x = 17$. Wir sehen, dass das erste Teilgeheimnis von den übrigen drei Teilgeheimnissen auf die gleiche Art abhängt wie das letzte Teilgeheimnis von den ersten drei Teilgeheimnissen.

Was geschieht aber, wenn nicht alle Personen zusammenkommen? Kann auf einige Teilgeheimnisse bei der Rekonstruktion des Geheimnisses verzichtet werden? Betrachten wir wieder unser Beispiel. Wir nehmen an, die letzten drei Teilnehmer kommen zusammen, um etwas über das Geheimnis zu erfahren. Sie kennen damit die Teilgeheimnisse 47, 31 und 34. Sie kennen auch die Zahl 53. Sie wissen jedoch nicht das Teilgeheimnis des ersten Teilnehmers. Das Geheimnis selber ist ja der Rest bei Division durch 53 der Summe der Teilgeheimnisse. Die Summe der Teilgeheimnisse der letzten drei Teilnehmer ist 112. Bei Division mit Rest durch 53 liefert 112 den Wert 6. Hätte nun der erste Teilnehmer statt der 17 das Teilgeheimnis 0 erhalten, so wäre das Geheimnis 6 und nicht 23 gewesen. Wäre das erste Teilgeheimnis 1 gewesen, so wäre das Geheimnis 7 gewesen. Und so geht es weiter, bis bei den Teilgeheimnissen 51 und 52 für den ersten Teilnehmer das Geheimnis 4 bzw. 5 gewesen wäre. Genauer: Es gibt zu jeder Zahl $g$ zwischen 0 und 53 eine andere Zahl $t$,

so dass die Summe von 112 und $t$ bei Division mit Rest durch 53 die Zahl $g$ ergibt. Das aber heißt, dass bei Teilgeheimnis $t$ für den ersten Teilnehmer und den Teilgeheimnissen 47, 31 und 34 für die anderen Personen das Geheimnis $g$ und nicht $G = 23$ gewesen wäre. Die letzten drei Teilnehmer können also nur mit Kenntnis ihrer drei Teilgeheimnisse keinen der möglichen Werte für das Geheimnis insgesamt ausschließen. Dieses heißt dann aber auch, dass die letzten drei Teilnehmer nur mit Kenntnis ihrer Teilgeheimnisse nichts über das Geheimnis erfahren. Dies gilt auch ganz allgemein.

Allerdings muss man darauf achten, die Zahlen nicht zu klein zu wählen. In unserem Beispiel mit $N = 53$ ist es sicherlich kein Problem, alle Möglichkeiten für ein fehlendes Teilgeheimnis auszuprobieren. Schließlich gibt es ja nur 53 mögliche Werte für jedes Teilgeheimnis. Einfacher noch gibt es ja insgesamt nur 53 Möglichkeiten für das Geheimnis selbst, die man leicht alle ausprobieren kann. In Anwendungen des Geheimnisteilens wird daher auch mit deutlich größeren Werten als 53 für $N$ gearbeitet. Da wird dann $N$ vielleicht als $10^{50}$ gesetzt. Dann gibt es für jedes Teilgeheimnis auch $10^{50}$ Möglichkeiten. In diesem Fall ist es völlig utopisch, ein fehlendes Teilgeheimnis durch Ausprobieren zu bestimmen.

## Allgemeines Geheimnisteilen

Als nächstes wollen wir sehen, ob man Geheimnisse auch so teilen kann, dass nicht unbedingt alle Personen zusammenkommen müssen, um das Geheimnis aufzudecken. Vielmehr soll es jeder genügend großen Anzahl von Teilnehmern bereits möglich sein, das Geheimnis zu rekonstruieren.

Zu Beginn betrachten wir ein Beispiel, bei dem ein Geheimnis so auf drei Personen verteilt werden soll, dass jeweils zwei von ihnen das Geheimnis aufdecken können. Einer alleine soll jedoch nichts oder möglichst wenig aus seinem Teilgeheimnis auf das Geheimnis schließen können. Unsere Idee von oben, das Geheimnis als Summe von Teilgeheimnissen darzustellen, führt nun leider nicht weiter. Wir brauchen eine neue Idee. Etwas Geometrie hilft uns weiter. Das Geheimnis sei nun ein Punkt $P$ in der Ebene. Wir können uns dabei vorstellen, dass die Koordinaten des Punktes $P$ zusammen die Geheimzahl eines Safes bilden. Weiter wählen wir drei Geraden, die sich alle in diesem Punkt $P$ schneiden. Die drei Geraden sind nun die Teilgeheimnisse. Wir haben dieses in dem folgenden Bild dargestellt.

Kommen nun zwei der drei Teilnehmer zusammen, so können sie den Schnittpunkt ihrer beiden Geraden berechnen. Dieses liefert ihnen genau das Geheimnis $P$. Dieses sieht man auch auf den drei Bildern in Abb. 17.4.

Wie viel erfährt ein einziger Teilnehmer durch sein Teilgeheimnis über das Geheimnis $P$? Nun, er lernt schon etwas dazu. Denn zunächst einmal ist ja das Geheimnis ein beliebiger Punkt in der Ebene. Nachdem ein Teilnehmer sein Teilgeheimnis erfahren hat, weiß er jedoch, dass das Geheimnis $P$ auf der

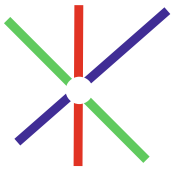
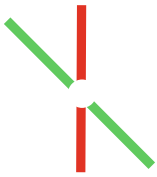
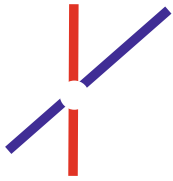


**Abb. 17.3.** Drei Geraden, die sich in einem Punkt schneiden. Der Schnittpunkt ist das Geheimnis

(a) Mit Hilfe der grünen und blauen Geraden kann das Geheimnis bestimmt werden.



(b) Mit Hilfe der grünen und roten Geraden kann das Geheimnis bestimmt werden.



(c) Mit Hilfe der roten und blauen Geraden kann das Geheimnis bestimmt werden.

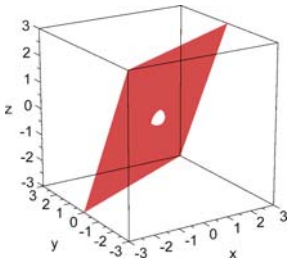
**Abb. 17.4.** Geheimnisteilung mit Geraden in der Ebene

Geraden liegt, die sein Teilgeheimnis ist. Er hat also schon etwas dazu gelernt. Aber er kennt nicht das Geheimnis selber.

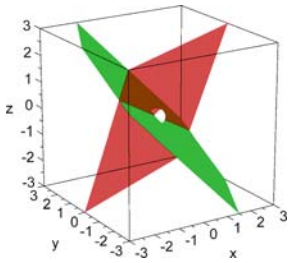
Man kann dieses Verfahren leicht verallgemeinern, so dass je zwei von $m$ beliebigen Teilnehmern aus ihren Teilgeheimnissen ein Geheimnis rekonstruieren können. Hierzu ist das Geheimnis wieder ein Punkt $P$ in der Ebene. Die $m$ Teilgeheimnisse sind dann $m$ Geraden, die sich alle im Punkt $P$ schneiden.

Wie sieht es aus, wenn jeweils drei der Teilnehmer das Geheimnis rekonstruieren können sollen? Nun müssen wir die Ebene verlassen und in den (3-dimensionalen) Raum gehen. Wieder wählt man als Geheimnis einen Punkt $P$, diesmal allerdings im Raum. Als Teilgeheimnisse wählen wir nun Ebenen und zwar so, dass sich je drei dieser Ebenen genau in dem Punkt $P$ schneiden. Wir haben dieses in den Bildern in Abb. 17.5 für 4 Teilnehmer dargestellt.

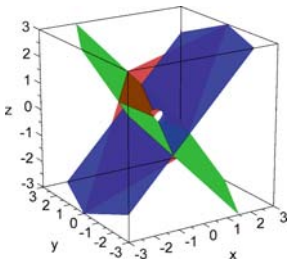
Drei der Teilnehmer können also immer das Geheimnis bestimmen, indem sie den gemeinsamen Schnittpunkt ihrer Ebenen bilden. Man kann dies in Bild (c) der Abb. 17.5 gut erkennen. Allerdings lernen auch weniger als drei Teilnehmer immer etwas über das Geheimnis. Kommen z.B. nur zwei der Teilnehmer zusammen, so schneiden sich ihre Ebenen in einer Geraden. Man kann dieses an der roten und grünen Ebene in Bild (b) der Abb. 17.5 sehen. Kombinieren die Besitzer der roten und grünen Ebene ihre Teilgeheimnisse, so wissen sie, dass das Geheimnis $P$ auf der Schnittgeraden der roten und



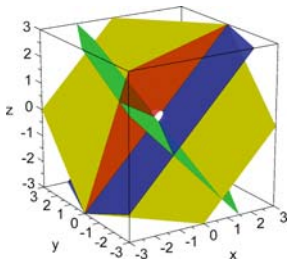
(a) Das Geheimnis $P$ ist der weiße Punkt in der Mitte. Er liegt in der roten Ebene.



(b) Das Geheimnis liegt im Schnitt der grünen und roten Ebene. Damit liegt er auch auf der Schnittgeraden der grünen und roten Ebenen.



(c) Das Geheimnis liegt im Schnitt der grünen, roten und blauen Ebene. Diese drei Ebenen bestimmen das Geheimnis eindeutig.



(d) Das Geheimnis liegt im Schnitt der vier Ebenen (grün, rot, blau, gelb). Je drei dieser Ebenen genügen, um das Geheimnis zu bestimmen.

**Abb. 17.5.** Geheimnisteilung mit Ebenen im Raum

grünen Ebene liegt. Allerdings haben sie keine Ahnung, welcher Punkt auf der Geraden das Geheimnis ist.

## Ausblicke

Ganz allgemein können wir uns nun fragen, ob ein Geheimnis $G$ so auf $m$ Personen aufgeteilt werden kann, dass je $t$ oder mehr das Geheimnis rekonstruieren können, weniger als $t$ Teilnehmer jedoch wenig oder gar nichts über das Geheimnis lernen. Die Antwort ist, dass dieses möglich ist. Eine Realisierung besteht in einer Verallgemeinerung unserer geometrischen Konstruktion. Man muss dann bei $t$ aus $m$ Teilnehmern in den so genannten $t$-dimensionalen Raum gehen.

Es gibt auch Konstruktionen, bei denen weniger als $t$ Teilnehmer absolut nichts über das Geheimnis erfahren. Diese beruhen allerdings nicht auf dem Schnitt von Ebenen, sondern auf so genannten Polynomen. Das Verfahren stammt von A. Shamir, einem berühmten Kryptologen, und wird daher auch *Shamir's secret sharing scheme* genannt.

Wer teilt eigentlich das Geheimnis auf? Diese Frage haben wir bislang völlig ausgeklammert. Sie ist aber natürlich wichtig, denn diejenige Person, die das Geheimnis aufteilt, wird dieses auch kennen. Wenn man das Teilen von Geheimnissen in der Kryptographie anwenden will, bleibt häufig nichts anderes übrig als anzunehmen, dass es eine besonders vertrauenswürdige Person gibt, die man mit dem Aufteilen des Geheimnisses beauftragen kann, ohne dass diese Person versuchen wird, daraus einen Nutzen zu ziehen. Stellt euch diese Person als vollkommen uneigennützigen und unbestechlichen Schiedsrichter vor.

Was heißt es eigentlich, Informationen zu gewinnen? Was ist Information eigentlich? Irgendwie wissen wir das sicherlich alle. Aber wenn man Informationen mathematisch betrachten will, muss man genauer sein. Beim Teilen von Geheimnissen z. B., wie wir es vorgestellt haben, will und muss man präzise sagen, was es eigentlich heißt, nichts erfahren zu haben. Aber wenn man einmal verstanden hat, warum unsere oben vorgestellten Methoden zum Teilen von Geheimnissen funktionieren, ist es nicht mehr schwer, Begriffe wie Information oder Informationsgewinn mathematisch präzise zu definieren. So liefern Teilgeheimnisse eben keinerlei zusätzliche Informationen über das Geheimnis, wenn man die Möglichkeiten für die Werte des Geheimnisses überhaupt nicht durch die Kenntnis der Teilgeheimnisse einschränken kann. Es war der bedeutende Mathematiker Claude Shannon, der schon um 1948 auf diesem Wege die so genannte Informationstheorie begründete.

Wir können aber auch noch etwas weiter gehen. Was nützt es uns, wenn wir Informationen zwar prinzipiell gewinnen können, aber dieses nur mit extrem hohem Zeitaufwand möglich ist. Das Teilen von Geheimnissen bietet hier wieder ein gutes Beispiel. Egal wie wir die 50-stellige Geheimzahl eines Safes auf die 10 Mitglieder einer Kommission verteilen, generell ist es natürlich möglich, die Geheimzahl zu bestimmen. Es gibt $10^{50}$ Möglichkeiten für die Geheimzahl. Diese Möglichkeiten können prinzipiell natürlich alle ausprobiert werden, um die Geheimzahl zu bestimmen. Aber die Betonung liegt hierbei auf prinzipiell. Denn $10^{50}$ ist eine so unvorstellbar große Zahl, dass niemand, nicht einmal mit Hilfe eines Computers, alle $10^{50}$ Möglichkeiten wirklich schnell ausprobieren kann. Wir können also sagen, dass ein Geheimnis bereits dann nicht aufgedeckt werden kann, wenn der Zeitaufwand, der für das Aufdecken benötigt wird, zu groß ist, als dass man das Geheimnis praktisch wirklich berechnen kann. Diese Überlegungen führen über die oben erwähnte Informationstheorie hinaus. Sie führen zu der Frage, wie viele Ressourcen eigentlich benötigt werden, um etwas zu berechnen oder Informationen zu gewinnen. Für viele interessante Probleme wie die Multiplikation großer Zahlen (Kap. 11) kann in diesem Buch nachgelesen werden, wie schnell sie gelöst werden können. Die Kapitel über

Public-Key-Kryptographie (Kap. 16) oder über Einwegfunktionen (Kap. 14) zeigen andererseits, dass es nützlich sein kann, wenn man weiß (oder vermutet), dass Probleme nicht gut gelöst werden können oder Informationen nicht oder nur schwer zugänglich sind.

## Zum Weiterlesen

1. Kapitel 16 (Public-Key-Kryptographie)

   Geheime Schlüssel von Public-Key-Verfahren, wie sie in diesem Kapitel beschrieben sind, werden in vielen Anwendungen nicht einer Person zugeordnet. Vielmehr werden sie mit Verfahren zum Teilen von Geheimnissen auf viele Personen verteilt. So wird verhindert, dass etwa der geheime Schlüssel eines großen Konzerns in den Händen einer einzigen Person liegt.

2. `http://en.wikipedia.org/wiki/Secret_sharing`
   `http://de.wikipedia.org/wiki/Secret_Sharing`

   Hier werden einige der vorgestellten Verfahren ebenfalls erläutert. Leider ist die deutsche Version des Artikels sehr viel kürzer und kaum brauchbar.

3. `http://de.wikipedia.org/wiki/Shamirs_Secret_Sharing`

   In diesem Wikipedia-Artikel wird das im Beitrag angesproche Verfahren von Shamir erläutert.

4. Wade Trapp und Lawrence Washington: *Introduction to Cryptography*. Pearson Education International, Second Edition, 2006.

   Wer es genauer wissen will, sollte in dieses Buch schauen. Nicht nur werden hier die wesentlichen Verfahren des Geheimnisteilens erläutert, es werden auch noch Erweiterungen und Varianten des Teilens von Geheimnissen beschrieben. Leider gibt es kein deutsches Buch, das das Teilen von Geheimnissen gut und ausführlich beschreibt.

# 18

# Poker per E-Mail

Detlef Sieling

Universität Dortmund

In diesem Kapitel wollen wir untersuchen, ob es möglich ist, Kartenspiele, beispielsweise Poker, zu spielen, ohne dass sich die Spieler dazu treffen. Stattdessen sollen die Karten mit Hilfe von E-Mails oder Briefen verteilt werden. Anders als bei kommerziellen Online-Poker-Systemen sollen die Spieler dabei selbst die Karten mischen und verteilen; es soll also keinen vertrauenswürdigen Kartengeber geben. Hierbei gibt es einige offensichtliche Schwierigkeiten: Wenn ein Spieler die Karten verteilt, muss er dies tun, ohne etwas über die Karten zu erfahren, die er verteilt. Er soll dazu E-Mails an die anderen Spieler senden, aus denen diese ihre Karten entnehmen können, der Kartengeber aber nicht. Weiterhin soll kein Spieler die bereits vergebenen Karten kennen. Andererseits dürfen Karten nicht mehrfach vergeben werden. Schließlich soll es in jedem Fall auffallen, wenn ein Spieler nicht fair spielt.

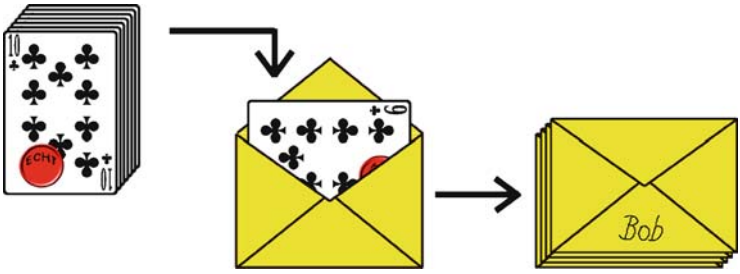
## Pokern mit Briefpost

Um Ideen zu sammeln, wie man Kartenspiele per E-Mail realisieren kann, überlegen wir zunächst, wie dies mit normaler Briefpost gehen kann. Wir betrachten nur die Situation mit den zwei Spielern Alice und Bob. Diese halten sich an verschiedenen Orten auf und können somit nicht beobachten, was ihr Gegenspieler macht. Die einfachste Möglichkeit zu mogeln besteht darin, dass einer der Spieler Karten aus einem zweiten identischen Kartenspiel verwendet, aus dem er bei Bedarf gute Karten, beispielsweise einen Royal Flush, auswählt. Um dies zu verhindern, bekommt jede Karte des verwendeten Kartenspiels ein Siegel, das die Karte als „echt“ kennzeichnet, mit dem sie sich also von Karten aus anderen Kartenspielen unterscheidet.

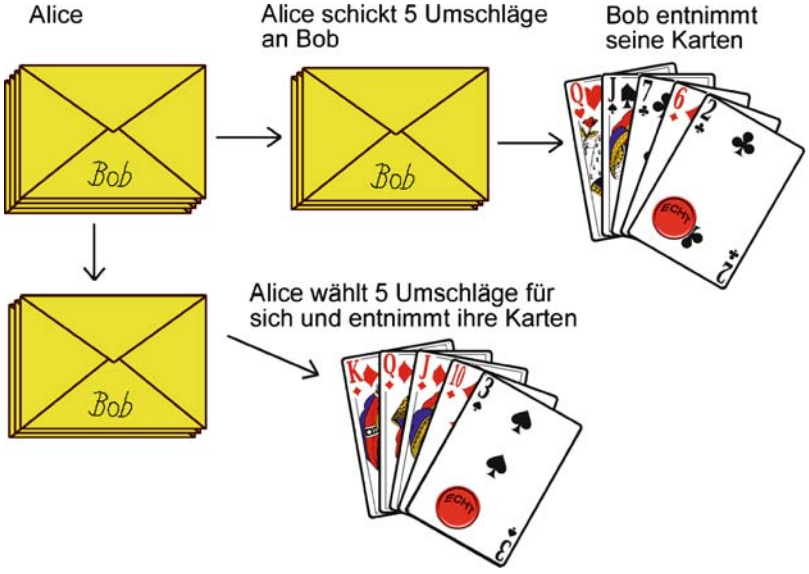
### Mischen und Verteilen der Karten

Beim Pokern verwenden wir ein Kartenspiel mit den 52 Karten Kreuz-As, Kreuz-Zwei, ..., Karo-König. Nach dem Mischen soll jeder Spieler fünf Kar-

ten erhalten. Wie kann ein Spieler die Karten mischen und verteilen, ohne dass er dabei die Möglichkeit hat, die Karten zu sehen? Wir verwenden dazu Briefumschläge. Bob legt jede der 52 Karten in einen gelben Umschlag. Damit später klar ist, wer die Karte in den Umschlag gesteckt hat, versieht er jeden Umschlag mit seiner Unterschrift.




Dann mischt er den Stapel von 52 Umschlägen und schickt ihn an Alice. Für Alice sehen die Umschläge alle gleich aus, sodass sie nicht in der Lage ist, für sich selbst bessere Karten und für Bob schlechtere Karten auszuwählen. Also kann Alice nur die Umschläge mischen und für sich selbst und für Bob jeweils fünf Umschläge auswählen. Dies entspricht einer zufälligen Auswahl der Karten. Alice sendet Bob seine fünf Umschläge zu, und er kann seine Karten einfach entnehmen. Ebenso kann Alice ihre fünf Karten entnehmen.




Welche Möglichkeiten gibt es hier zu mogeln? Alice könnte beispielsweise weitere Umschläge öffnen, um aus dieser größeren Menge von Karten die besten Karten auszusuchen. Dies kann man zu diesem Zeitpunkt offensichtlich nicht verhindern. Wenn Alice aber fair gespielt hat, kann sie nach Ende des

Spiels, z. B. wenn sich Alice und Bob später einmal treffen, 42 geschlossene und von Bob unterschriebene Umschläge vorweisen. Da die Umschläge von Bob unterschrieben wurden, kann sie auch eine einmal entnommene Karte nicht wieder verpacken, ohne dass dies auffällt. Wenn Bob beim Verpacken der Karten einen Fehler gemacht hat, beispielsweise eine Karte behalten und dafür einen Umschlag leer gelassen hat, kann er ebenfalls keinen Vorteil daraus ziehen. Wenn er den leeren Umschlag während des Spiels zieht, hätte er sowieso die nicht verpackte Karte bekommen. Anderenfalls fällt spätestens bei der Kontrolle der Umschläge auf, dass er beim Verpacken der Karten einen Fehler gemacht hat.

### Bieten

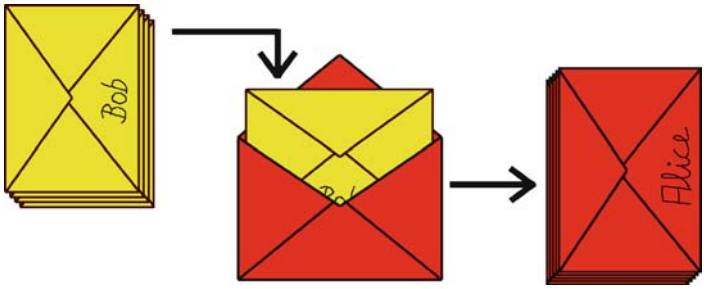
Nach dem Verteilen der Karten kommt beim Pokern das Bieten. Jeder Spieler kann einen Geldbetrag setzen oder erhöhen oder kann auch passen. Was beim normalen Pokern mündlich gemacht wird, kann man ohne weitere Ideen auch schriftlich durchführen, d.h., die Spieler teilen sich ihre Entscheidungen einfach in Briefen mit.

### Tauschen von Karten

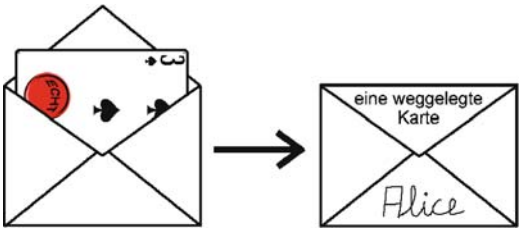
Nach dem Bieten darf jeder Spieler eine oder mehrere seiner Karten tauschen. Zuerst kann Alice $n$ Karten tauschen (wobei $n$ zwischen 1 und 5 liegt). Hierbei gibt es aber eine Komplikation: Alice muss zuerst $n$ Karten weglegen und darf dann erst $n$ neue Karten erhalten. Wenn sie sich selber die neuen Karten auf die oben beschriebene Weise gibt, kann man nicht mehr feststellen, ob sie nicht zuerst $n$ neue Karten genommen und dann erst die Karten ausgewählt hat, die sie angeblich schon zuvor weggelegt hat. Also muss Bob an der Verteilung von weiteren Karten an Alice beteiligt werden.

Andererseits darf Alice die 42 vorhandenen Umschläge nicht einfach an Bob zurücksenden. Damit verliert sie ihren Beweis, dass sie bis jetzt fair gespielt hat. Weiterhin könnte Bob geheime Markierungen an den gelben Umschlägen angebracht haben, mit deren Hilfe er die guten Karten erkennt, beispielsweise durch geringfügige Unterschiede in seinen Unterschriften. Dies geht auch noch unauffälliger als in dem folgenden Bild, in dem das kleine „b“ je nach Inhalt etwas anders geschrieben ist.

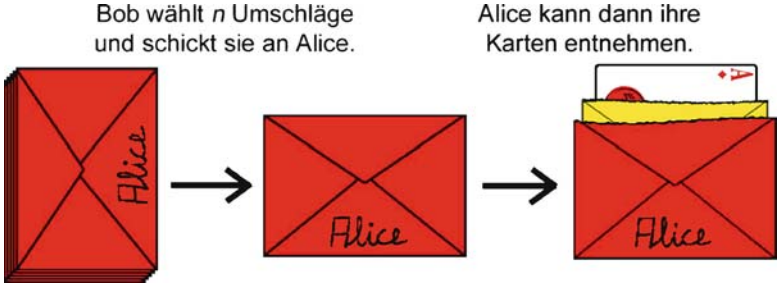
Der Trick mit Umschlägen funktioniert aber auch hier. Alice verpackt die verbliebenen 42 gelben Umschläge in etwas größere rote Umschläge, unterschreibt diese, mischt den Stapel und sendet die Umschläge an Bob.




Weiterhin verpackt sie die $n$ Karten, die sie tauschen möchte, in einem separaten Umschlag und schickt diesen ebenfalls an Bob. Den Umschlag mit den weggelegten Karten lässt er ungeöffnet, da er nicht erfahren darf, welche Karten Alice weggelegt hat. Später kann dann leicht überprüft werden, ob dieser Umschlag verschlossen geblieben ist und tatsächlich $n$ Karten enthält.




Da für Bob die roten Umschläge alle gleich aussehen, kann er nichts Besseres machen, als die Umschläge ebenfalls zu mischen und dann $n$ rote Umschläge auszuwählen. Diese schickt er an Alice, die damit die ausgetauschten Karten erhält.




Wenn Bob Karten tauschen möchte, geht dies nach demselben Schema, wobei Bob die roten Umschläge in eine weitere Schicht von Umschlägen verpackt.

### Aufdecken der Karten

Das Pokerspiel endet mit dem Aufdecken der Karten. Jeder der Spieler teilt dem anderen nur mit, welche Karten er hat. Die Karten behält jeder Spieler, um bei Bedarf bei einem späteren Treffen nachweisen zu können, dass er diese Karten wirklich hat. Damit steht der Gewinner fest.

### Kontrolle, ob fair gespielt wurde

Das bisher beschriebene Verfahren bietet beiden Spielern viele Möglichkeiten, durch abweichendes Verhalten einen Vorteil zu erlangen. Beispielsweise könnte ein Spieler Umschläge öffnen, die er nicht öffnen darf, um eine größere Auswahl an Karten oder Informationen über die Karten des Gegners zu erhalten. Dies lässt sich aber einfach entdecken: Die Karten und die ungeöffneten Umschläge werden bis zum nächsten Treffen von Alice und Bob aufbewahrt. Dann können beide die Karten vorweisen und die Umschläge gemeinsam öffnen, um zu überprüfen, ob der andere sich an das beschriebene Verfahren gehalten hat, also fair gespielt hat.

### Diskussion

Das Pokern mit normaler Post hat mehrere offensichtliche Nachteile:

- Das Verpacken in Umschläge ist nur von Hand möglich und recht aufwändig. Die gebrauchten Umschläge können nicht wiederverwendet werden.
- Die Überprüfung, ob der andere Spieler fair gespielt hat, kann erst bei dem nächsten Treffen der beiden durchgeführt werden. Für jedes weitere Spiel vor diesem Treffen wird also ein weiteres Kartenspiel mit einem jeweils anderen Siegel benötigt.
- Die normale Post ist zu langsam, sodass das Spiel nicht viel Spaß macht und im Vergleich zu E-Mail auch teuer ist.
- Wenn ein Brief abhanden kommt, kann man das Spiel nicht zu Ende spielen. Wenn ein Spieler merkt, dass er verliert, könnte er sogar einen Brief verschwinden lassen, und es ist später nicht mehr feststellbar, wer Schuld daran ist.

Die naheliegende Frage ist nun, ob man „elektronische Briefumschläge“ realisieren kann, die ähnliche oder vielleicht sogar bessere Eigenschaften als die Briefumschläge aus Papier haben. Insbesondere sollte man sie mit Hilfe eines Computers erzeugen und per E-Mail verschicken können. Man spart dann die Arbeit, die Karten einzeln von Hand zu verpacken, und verloren gegangene E-Mails kann man ein zweites Mal verschicken.

## Pokern mit elektronischer Post

### Elektronische Umschläge

Wie können die Umschläge mit elektronischer Post realisiert werden? Eine Idee besteht darin, dass Bob die Karten codiert und Alice beim Mischen nur die Codes sieht. Dabei soll Alice keine Idee haben, welcher tatsächlichen Karte ein Code entspricht. Bevor wir allgemein beschreiben, wie das Mischen und Verteilen der Karten realisiert werden kann, konzentrieren wir uns auf den Spezialfall, dass Karten nur an Bob auszugeben sind. Wir gehen im Folgenden an manchen Stellen davon aus, dass den Karten feste Nummern zugeordnet wurden und beide Spieler diese Zuordnung kennen, also 0 entspricht dem Kreuz-As, 1 der Kreuz-Zwei, 2 der Kreuz-3, ..., 12 dem Kreuz-König, 13 dem Pik-As usw. bis 51 dem Karo-König.

### Mischen und Ausgabe der Karten an Bob

Bob erzeugt zu Beginn zufällige Codes für die Karten, d.h. eine Tabelle der folgenden Form:

| Karte | | Code |
|---|---|---|
| 0 (Kreuz-As) | → | 1 |
| 1 (Kreuz-Zwei) | → | 42 |
| 2 (Kreuz-Drei) | → | 22 |
| 3 (Kreuz-Vier) | → | 25 |
| 4 (Kreuz-Fünf) | → | 51 |

| Karte | | Code |
|---|---|---|
| 5 (Kreuz-Sechs) | → | 0 |
| 6 (Kreuz-Sieben) | → | 43 |
| ⋮ | | |
| 51 (Karo-König) | → | 13 |

In der linken Spalte der Tabelle sind alle Karten aufgeführt. Die rechte Spalte wurde zufällig erzeugt, sodass jeder Code aus dem Bereich von 0 bis 51 genau einmal vorkommt. Alice soll diese Tabelle zunächst nicht erhalten.

Um fünf Karten für Bob auszuwählen, wählt Alice zufällig fünf Codes aus dem Bereich von 0 bis 51 aus und sendet sie an Bob. Dieser benutzt dann die Tabelle, um herauszufinden, welche Karten er bekommen hat. Wenn Alice beispielsweise die Codes 0, 1, 13, 42 und 51 ausgewählt hat, erhält Bob gemäß der Tabelle oben die Karten Kreuz-Sechs, Kreuz-As, Karo-König, Kreuz-Zwei und Kreuz-Fünf. Da Alice die Tabelle nicht kennt, kann sie keinen Einfluss auf die Karten von Bob nehmen. Weiterhin kann sie sich merken, welche Codes bereits gebraucht wurden, sodass jede Karte nur einmal ausgegeben wird.

Die Vorgehensweise ist also ähnlich zu den Briefumschlägen. Anstatt das Kreuz-As in einen gelben Briefumschlag zu verpacken, erhält es von Bob einen Code, im betrachteten Beispiel die 1. Bei der Verwendung von Briefumschlägen kann Alice nicht in den Umschlag hineinsehen. Hier kennt Alice die Bedeutung des Codes 1 nicht. Somit kann Alice in beiden Fällen keinen Einfluss auf die gewählten Karten nehmen.

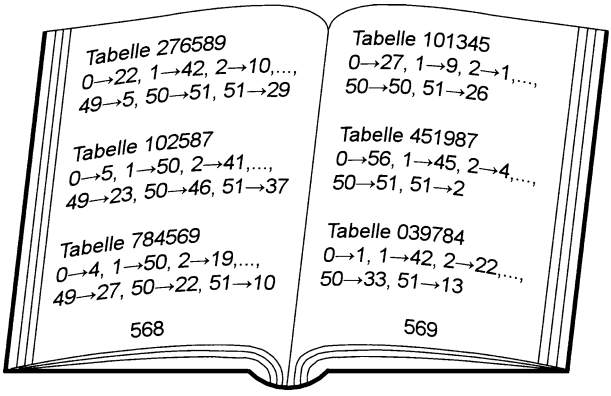
Allerdings könnte Bob am Ende des Spiels behaupten, dass er eine ganz andere Tabelle erzeugt hat und sich somit nachträglich bessere Karten geben.

Also muss sich Bob zu Beginn in einer für Alice nachprüfbaren Weise auf eine Tabelle festlegen, sodass er sie im Nachhinein nicht mehr verändern kann. Wir verwenden dazu die so genannten Einwegfunktionen.

### Einwegfunktionen

Einwegfunktionen wurden bereits im Kap. 14 ausführlich vorgestellt. Wir erinnern uns: Eine Einwegfunktion $f$ ist eine Funktion, die leicht berechnet werden kann, bei der aber die Umkehrfunktion $f^{-1}$ schwer zu berechnen ist. Ein Beispiel im Kap. 14 war ein Telefonbuch: Die Einwegfunktion $f$ entspricht dem Finden einer Telefonnummer zu einem gegebenen Namen, was leicht ist. Die Umkehrfunktion $f^{-1}$ entspricht dem Finden des Namens zu einer Telefonnummer, was dagegen schwer ist.

Wie können wir jetzt Einwegfunktionen nutzen, damit Bob die Tabelle der Codierungen der Karten nachträglich nicht mehr ändern kann? Wir stellen uns dazu in Analogie zu dem Telefonbuch vor, dass Alice und Bob ein Buch mit sehr vielen Codierungstabellen bekommen haben, das zu jeder aufgeführten Codierungstabelle eine eindeutige Zahl angibt.




Die Ausgabe der Karten an Bob erfolgt dann so: Zuerst wählt Bob zufällig eine Codierungstabelle aus dem Buch (z.B. die untere auf Seite 569) und sendet die zugehörige eindeutige Zahl an Alice. Im Beispiel ist dies 039784. Wenn Alice erfahren möchte, welche Codierungstabelle Bob benutzt, muss sie im Wesentlichen das gesamte Buch durchlesen. Wenn dies für sie zu aufwändig ist, hat sie keine Möglichkeit, die verwendete Codierungstabelle zu finden. Sie kann also nichts Besseres tun, als zufällig fünf Zahlen aus dem Bereich von 0 bis 51 auszuwählen und an Bob zu senden. Der erhält dann anhand der verwendeten Tabelle seine Karten. Der Karte mit der Nummer 0 (also dem Kreuz-As) wird also der Code 1 zugeordnet, der Karte mit der Nummer 1 (also der Kreuz-Zwei) der Code 42 usw. Nach Ende des Spiels kann Bob angeben, wo in dem Buch die verwendete Tabelle steht. Somit erhält Alice die Tabelle und kann nachsehen, ob die Nummer dieser Tabelle mit der zu

Beginn von Bob genannten Nummer übereinstimmt und ob Bob wirklich die Karten bekommen hat, von denen er dies behauptet.

### Tauschen von Karten

Das Weglegen von Karten erfordert nun keine neuen Ideen. Wenn Bob im obigen Beispiel die Kreuz-Zwei weglegen will, teilt er Alice einfach mit, dass er die Karte mit dem Code 42 weglegen möchte. Da Alice den Zusammenhang zwischen Kreuz-Zwei und dem Code 42 nicht kennt, erfährt sie auch nicht, welche Karte Bob weggelegt hat. Anschließend kann Alice Bob neue Karten geben. Da sie weiß, welche Codes sie bereits an Bob gesendet hat, kann sie auch verhindern, Karten mehrfach zu geben, ohne zu wissen, welche Karten sie bereits ausgegeben hat.

### Mathematischere Formulierung

Etwas mathematischer formuliert, beschreibt das Buch mit den Codierungstabellen eine Einwegfunktion. Diese Einwegfunktion $f$ bildet die Positionen der Codierungstabellen auf Zahlen ab. In dem beschriebenen Verfahren wählt Bob zufällig eine Codierungstabelle mit der Position $x$ im Buch und sendet $f(x)$ an Alice. Da $f$ eine Einwegfunktion ist, kann Alice nicht auf effiziente Weise $x$ aus $f(x)$ berechnen; dies würde dem Durchsuchen des Buches entsprechen. Nach Ende des Spiels kann Bob ihr $x$ zusenden. Alice kann dann leicht $f(x)$ berechnen und überprüfen, ob dies wirklich der Wert ist, den sie zu Beginn erhalten hat. Somit kann Bob nicht nachträglich behaupten, eine andere Codierungstabelle verwendet zu haben.

Für einen Computer ist es nun kein Problem, ein komplettes Buch zu speichern und zu durchsuchen. Statt eines solchen Buches sollte man daher Einwegfunktionen verwenden, die aus der Codierungstabelle direkt die Zahl ausrechnen, mit der sich die Spieler auf die verwendete Tabelle festlegen. Auf weitere Einzelheiten dazu wollen wir hier aber nicht eingehen.

Man kann auch jede Codierungstabelle selbst als Funktion auffassen. Wir verwenden dazu die festen Nummern der Karten, die wir bereits oben in der Tabelle angegeben haben. Die Codierungstabelle ist dann eine Funktion $b$, die den Nummern der Karten (aus dem Bereich von 0 bis 51) Codierungen zuordnet (ebenfalls aus dem Bereich von 0 bis 51). Anhand der Codierungstabelle können wir auch leicht die Umkehrfunktion von $b$ berechnen. Die Funktion $b^{-1}$ ordnet jedem Code die Nummer der zugehörigen Karte zu. Um $b^{-1}(z)$ zu berechnen, suchen wir den Eintrag $z$ in der rechten Spalte der Tabelle und lesen das Ergebnis in der linken Spalte ab. Da in der rechten Spalte der Tabelle jede Zahl genau einmal vorkommt, gilt für alle $x$, dass $b^{-1}(b(x)) = x$ ist.

### Verteilen von Karten an beide Spieler

Um Karten an beide Spieler ausgeben zu können, benutzen Alice und Bob eigene Codierungstabellen, die sie unabhängig voneinander erzeugen und dem

Gegenspieler jeweils nicht bekannt geben. Die Funktion, die von Alice' Codierungstabelle beschrieben wird, bezeichnen wir mit $a$, die von Bobs Codierungstabelle mit $b$. Als weitere Voraussetzung an $a$ und $b$ verlangen wir, dass für alle $x$ aus dem Bereich von 0 bis 51 gilt, dass $a(b(x)) = b(a(x))$ ist. Die Mathematiker sagen auch dazu, dass die Funktionen $a$ und $b$ kommutieren, d.h., dass wir unabhängig davon, ob wir zuerst $b$ und dann $a$ auf $x$ anwenden oder umgekehrt, denselben Funktionswert erhalten.

Ein Beispiel für kommutierende Funktionen sind die Folgenden:

$$a(x) = \begin{cases} x + 25, & \text{falls } x + 25 < 52, \\ x + 25 - 52, & \text{falls } x + 25 \geq 52, \end{cases}$$

und

$$b(x) = \begin{cases} x + 37, & \text{falls } x + 37 < 52, \\ x + 37 - 52, & \text{falls } x + 37 \geq 52. \end{cases}$$

Wir haben hier die Codierungstabellen nicht vollständig aufgeschrieben, sondern nur auf mathematische Weise beschrieben, wie man zu einem Eintrag in der linken Spalte den zugehörigen Eintrag in der rechten Spalte findet. Man rechnet leicht nach, dass $a(b(x))$ und $b(a(x))$ übereinstimmen: Für die Berechnung von $a(b(x))$ addiert man zu $x$ zunächst die 37 und anschließend die 25, wobei man jeweils 52 abzieht, falls das Ergebnis größer als 51 wird. Für die Berechnung von $b(a(x))$ führt man diese Additionen einfach in der umgekehrten Reihenfolge aus. Statt der 37 und der 25 kann man auch andere Zahlen nehmen.

Dieses anschauliche Beispiel von kommutierenden Funktionen ist für die praktische Anwendung allerdings ungeeignet. Wenn beispielsweise Bob für eine Nummer $x$ den Code $a(x)$ erfährt, kann er hieraus leicht den von Alice verwendeten Summanden 25 berechnen. Damit erfährt er die gesamte Funktion $a$ und kann alle Codes von Alice entschlüsseln. Weitere Details zu den verwendeten kommutierenden Funktionen finden sich in den im Abschn. „Zum Weiterlesen“ genannten Artikeln.

### Festlegung auf die verwendeten Codierungstabellen

Mit $a$ und $b$ bezeichnen wir die Codierungstabellen, die Alice und Bob gewählt haben. Wie oben verwenden wir eine Einwegfunktion $f$. Alice berechnet $f(a)$ und sendet diesen Wert an Bob. Ebenso berechnet Bob den Wert $f(b)$ und sendet ihn an Alice. Aus den Werten $f(a)$ bzw. $f(b)$ können Bob bzw. Alice nicht auf effiziente Weise Informationen über die Codierungstabelle des Partners erhalten, da $f$ eine Einwegfunktion ist. Andererseits haben sich die Spieler damit auf die Codierungstabellen $a$ bzw. $b$ festgelegt. Nach Ende des Spiels sendet Alice die Codierungstabelle $a$ an Bob, der dann $f(a)$ berechnen und somit überprüfen kann, ob $a$ wirklich die Codierungstabelle ist, auf die sich Alice durch Übersenden von $f(a)$ festgelegt hat. Ebenso kann Alice prüfen, ob die am Ende des Spiels von Bob angegebene Tabelle diejenige ist, auf die er sich zu Beginn festgelegt hat.

### Verpacken von Karten in Umschläge

Wenn Alice die Karte $x$ in einen Umschlag verpacken möchte, berechnet sie einfach $a(x)$. Wenn sie die Karte aus dem Umschlag $a(x)$ entnehmen möchte, genügt es, die Umkehrfunktion $a^{-1}$ auf $a(x)$ anzuwenden, denn $a^{-1}(a(x)) = x$. Ebenso kann Bob mit Hilfe der Funktion $b$ die Karten in Umschläge verpacken. Da wir sowohl für die Karten als auch die Codierungen (also die Umschläge) die Zahlen 0 bis 51 verwenden, kann Alice auch von Bob erzeugte Umschläge der Form $b(x)$ in ihre Umschläge verpacken, indem sie $a(b(x))$ berechnet.

Das Protokoll für die Briefpost sah vor, dass zuerst Bob die Karten in Umschläge verpackt und anschließend Alice die Umschläge in eine zweite Schicht von Umschlägen verpackt. Bob sollte also zuerst $b(0), \ldots, b(51)$ berechnen. Man überlegt leicht, dass dies genau die Zahlen 0 bis 51 sind. Ebenso sollte Alice anschließend $a(b(0)), \ldots, a(b(51))$ berechnen, was wiederum genau die Zahlen 0 bis 51 sind. Alice und Bob wissen also, dass zu Beginn die Codes 0 bis 51 vorhanden sind. Diese fassen sie als $a(b(0)), \ldots, a(b(51))$ auf. Da Alice $b$ nicht kennt, kann sie zu keinem Code bestimmen, welches die codierte Karte ist. Ebenso kann Bob dies nicht, da er $a$ nicht kennt.

### Auswahl von Karten für Alice

Bob wählt aus der Liste der noch vorhandenen Codes (zu Beginn 0 bis 51) für jede Karte, die Alice erhalten soll, einen Code zufällig aus und streicht den gewählten Code aus der Liste. Da er $a$ nicht kennt, hat er keine Möglichkeit, Einfluss auf die gewählten Karten zu nehmen. Wenn $a(b(x))$ ein von Bob gewählter Code ist, wendet er auf diesen $b^{-1}$ an und erhält $b^{-1}(a(b(x))) = b^{-1}(b(a(x))) = a(x)$, da $a$ und $b$ kommutieren und $b^{-1}(b(z)) = z$ gilt. Bob sendet dann den Wert $a(x)$ zusammen mit $a(b(x))$ an Alice. Alice kann dann auf $a(x)$ die Funktion $a^{-1}$ anwenden und erhält somit die Nummer $x$ der Karte. Weiterhin kann sie $a(b(x))$ aus der Liste der noch vorhandenen Codes streichen, sodass diese Karte nicht noch einmal vergeben werden kann.

### Auswahl von Karten für Bob

Alice wählt aus den noch vorhandenen Codes für jede Karte, die Bob erhalten soll, einen Code zufällig aus. Da sie $b$ nicht kennt, kann auch sie keinen Einfluss darauf nehmen, welche Karte gewählt wird. Wenn $a(b(x))$ ein solcher von Alice gewählter Code ist, wendet sie hierauf $a^{-1}$ an und erhält $a^{-1}(a(b(x))) = b(x)$, da $a^{-1}(a(z)) = z$ gilt. Den Wert $b(x)$ sendet sie zusammen mit $a(b(x))$ an Bob. Aus $b(x)$ kann Bob wieder $x$, also die Nummer der gewählten Karte erhalten. Weiterhin kann er $a(b(x))$ aus der Liste der noch vorhandenen Codes streichen.

### Weglegen von Karten

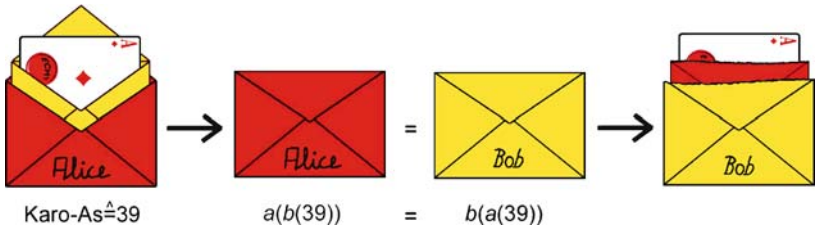
Wenn ein Spieler eine Karte $x$ weglegen will, sendet er dem anderen Spieler eine entsprechende Mitteilung zusammen mit dem Code $a(b(x))$. Wie bereits

festgestellt, kann der andere Spieler nicht die zugehörige Karte $x$ herausfinden, da er entweder $a$ oder $b$ nicht kennt.

### Besondere Eigenschaften der elektronischen Umschläge

Wir vergleichen nun die Umschläge aus Papier mit den elektronischen Umschlägen: Dazu betrachten wir noch einmal die Auswahl der Karten für Alice. Dem Verpacken der Karten in gelbe Umschläge durch Bob entspricht die Anwendung der Funktion $b$. Dem Verpacken dieser Umschläge in rote Umschläge durch Alice entspricht die Anwendung der Funktion $a$. Nach Auswahl der Karten für Alice entfernt Bob die inneren gelben Umschläge (Anwendung von $b^{-1}$), ohne die äußeren roten Umschläge zu öffnen oder etwas über den Inhalt der gelben Umschläge zu erfahren. Alice kann schließlich die roten Umschläge öffnen (Anwendung von $a^{-1}$). Die elektronischen Umschläge haben also Eigenschaften, die man mit Umschlägen aus Papier nicht realisieren kann:

- Alice kann die gelben Umschläge nicht öffnen, Bob kann die roten Umschläge nicht öffnen. Das heißt insbesondere, dass nicht mehr überprüft werden muss, ob die Spieler die nicht verwendeten Umschläge auch nicht geöffnet haben, da sie dies nicht können.
- Man kann Umschläge zusammen mit ihrem Inhalt kopieren, ohne dazu etwas über den Inhalt wissen zu müssen oder zu erfahren.
- Ein roter Umschlag, der einen gelben Umschlag mit einer Karte enthält, stimmt mit einem gelben Umschlag, der einen roten Umschlag mit derselben Karte enthält, überein. Somit ist es auch möglich, zuerst den gelben Umschlag zu entfernen, auch wenn die Karte zuerst in einen gelben und dieser dann in einen roten Umschlag gesteckt wurde.




### Kontrolle, ob fair gespielt wurde

Nach Ende des Spiels können die Spieler ihren Partnern die Codierungstabellen $a$ bzw. $b$ mitteilen. Dann können beide $f(a)$ bzw. $f(b)$ berechnen und somit überprüfen, ob dies wirklich die Codierungstabelle ist, auf die sich der andere zu Beginn festgelegt hat. Mit Hilfe der Codierungstabellen können dann beide Spieler überprüfen, ob ihre Partner Fehler bei den Berechnungen gemacht haben oder fair gespielt haben. Während des Spiels können die Spieler Umschläge nur gemeinsam öffnen, da man zum Öffnen beide Funktionen $a$ und $b$

kennen muss. Daher ist es nicht mehr nötig, dass sich die Spieler später noch einmal treffen, um zu überprüfen, ob die nicht verwendeten Umschläge immer noch verschlossen sind.

## Pokern mit mehr als zwei Spielern

Bis jetzt haben wir nur die Situation mit zwei Spielern betrachtet. Was passiert bei drei oder mehr Spielern? Man kann natürlich versuchen, das oben beschriebene Verfahren auf mehrere Spieler zu verallgemeinern. Es gibt aber ein grundsätzliches Problem. Unser Ausgangspunkt war, dass die Spieler sich nicht gegenseitig beobachten können. Wie also will der dritte Spieler verhindern, dass Alice und Bob zwischendurch telefonieren und Informationen über ihre Karten austauschen? Bei kommerziellen Online-Poker-Systemen besteht eine Möglichkeit darin, Spieler einander so zuzuordnen, dass sie sich höchstwahrscheinlich nicht gegenseitig kennen. Eine weitere Möglichkeit besteht darin, Auffälligkeiten im Verhalten der Gegenspieler zu finden, beispielsweise, wenn immer der Gegenspieler mit den schlechteren Karten passt. Aber selbst dann dürfte ein Betrug nur schwer nachzuweisen sein. Wenn man also in einer größeren Runde spielen will, sollte man sich trotz Computer und Internet weiterhin treffen, was ja vielleicht auch einfach mehr Spaß macht.

## Zum Weiterlesen

1. Adi Shamir, Ronald L. Rivest und Leonard M. Adleman: *Mental Poker*. In: *The Mathematical Gardner*, S. 37–43. Herausgegeben von David A. Klarner, Wadsworth International, 1981. Online erhältlich unter: `http://people.csail.mit.edu/~rivest/ShamirRivestAdleman-MentalPoker.eps`

   In diesem Artikel wurde erstmals ein Protokoll für Poker beschrieben. Das hier beschriebene Protokoll ist eine geringfügige Modifikation davon.

2. Bruce Schneier: *Angewandte Kryptographie*. Addison-Wesley, 1996.

   Dieses Buch beschreibt im Abschnitt 4.11 das Protokoll von Shamir, Rivest und Adleman für Poker und enthält darüber hinaus auch Protokolle für verschiedene andere Aufgabenstellungen.

3. Detlef Sieling, Ergänzungen zu „Poker per E-Mail“, 2006. Online erhältlich unter: `http://ls2-www.cs.uni-dortmund.de/~sieling/algodw/poker.eps`

   Hier werden weitere technische Einzelheiten des Protokolls von Shamir, Rivest und Adleman beschrieben.

Die in den Abbildungen verwendeten Spielkarten wurden von David Bellot entworfen. Sie sind unter `http://david.bellot.free.fr/svg-cards` erhältlich und stehen unter der LGPL (`http://www.gnu.org/copyleft/lesser.html`).

# 19

# Fingerprinting

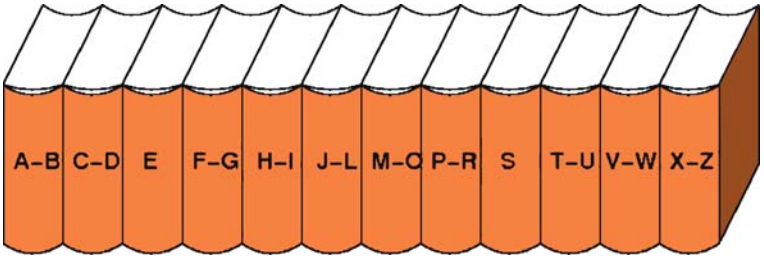
Martin Dietzfelbinger

Technische Universität Ilmenau

## Wie vergleicht man lange Texte übers Telefon?

Alice und Bob sind dicke Freunde. Alice wohnt in Adelaide in Australien und Bob in Barnsley in (Groß-)Britannien. Sie telefonieren gerne miteinander, was aber ganz schön ins Geld geht, wenn es länger dauert. Die beiden haben sehr ähnliche und sehr breit gestreute Interessen; da ist es kein Zufall, dass sich beide ein Lexikon anschaffen. Sie erzählen sich am Telefon von ihrem Lexikonkauf. So ein Zufall! Das gleiche Lexikon! Alice stellt Bob am Telefon eine einfache Frage: Steht in beiden Lexika derselbe Text? Sind sie wirklich Wort für Wort, Komma für Komma identisch?

Sogar wenn Alice und Bob feststellen, dass beide z. B. die 37. Auflage des Lexikons besitzen, muss das nicht unbedingt heißen, dass in dem in Australien gedruckten Exemplar wortwörtlich derselbe Text steht wie in dem in England gedruckten. Vielleicht wurden irgendwelche Druckfehler ausgebessert? Was tun? Alice könnte Bob ihr Lexikon vorlesen. Bob würde mitlesen und Wort für Wort vergleichen. Das würde theoretisch funktionieren, aber höllisch Telefongebühren kosten, denn ...




Alice und Bob haben ein sehr dickes Lexikon gewählt: 12 Bände, etwa 1500 Seiten pro Band, etwa 2800 Schriftzeichen pro Seite: macht etwa 18000 Seiten und etwa 50 Millionen Schriftzeichen. Wenn Alice für eine Seite nur 5 Minuten braucht, sind die beiden ohne Pause mehr als 60 Tage beschäftigt.

Wir können in einem Computer ein Schriftzeichen (auch die Kommas und Zwischenräume) mit einem Binärcode darstellen, z. B. mit 8 Bits (d.h. 1 Byte) pro Zeichen. Für das ganze Lexikon ergibt das 50 Millionen Bytes, oder etwa 50 Megabytes. Alice und Bob schaffen es irgendwie, jeder für sich, den Text ihres jeweiligen Lexikons in den Computer einzugeben. Wenn das Lexikon erst einmal elektronisch gespeichert ist, ist es heutzutage eigentlich kein Problem mehr, diese Menge an Daten per E-Mail von Australien nach England zu schicken. Wir wollen aber annehmen, dass die Verbindung extrem teuer oder sehr fehleranfällig ist, so dass eine Übermittlung dieser großen Menge von Daten nicht erwünscht oder möglich ist.

Können Alice und Bob feststellen, ob die beiden Texte gleich sind, ohne Buchstabe für Buchstabe zu vergleichen? Alice und Bob wollen möglichst wenig Text übermitteln.

Wer häufiger große Dateien per E-Mail verschickt oder auf seiner vollen Festplatte Platz schaffen musste, weiß, dass es „Datenkompression“ gibt. Das sind Methoden, mit denen man Daten wie Texte oder Bilder „zusammenquetscht“, um Übertragungszeiten zu verkürzen und um Speicherplatz zu sparen. Alice und Bob könnten solche Methoden benutzen. Aber sogar wenn sie eine Komprimierung auf ein Fünftel der ursprünglichen Länge bewerkstelligen könnten, wäre es den beiden immer noch zu viel Kommunikation. Also ist die Datenkompression kein Ausweg.

Hier ist noch eine ganz einfache Beobachtung: Alice sollte zunächst die Buchstaben in ihrem Lexikon zählen. Das Ergebnis wollen wir $n$ nennen. Alice sagt Bob, was $n$ ist. Das sind 8 Dezimalziffern, die sie leicht telefonisch durchgeben kann. Bob hat inzwischen die Buchstaben in seinem Lexikon gezählt, mit dem Resultat $n'$. Wenn $n$ und $n'$ verschieden sind, haben die Lexika verschiedene Texte, und wir sind fertig. (Wir nehmen an, dass Alice und Bob sich nicht verrechnen oder verzählen.) Die Länge des Textes ist also eine Kenngröße, die sehr wenig Platz einnimmt. Ab hier können wir uns vorstellen, dass die Texte von Alice und Bob genau gleich lang sind.

## Texte als Zahlenfolgen und Modulare Arithmetik

Um zu einem Trick zu kommen, der zu kurzen Nachrichten führt, wollen wir Texte in Zahlen übersetzen und dann mit diesen Zahlen rechnen. Dazu müssen wir uns ein bisschen Handwerkszeug zurechtlegen. Wir haben schon gesagt, dass man Buchstaben im Computer mit Mustern aus 8 Bits, also Bytes, darstellt. Eine Standard-Darstellung ist der ASCII-Code. In dieser Darstellung sehen `A`, `B`, `C`, ... so aus: 01000001, 01000010, 01000011 usw. Diese Bitmuster kann man auch als binäre Darstellung von Zahlen auffassen. Damit ergibt sich die folgende Codierung von Buchstaben als Zahlen:

| `A` | `B` | `C` | ··· | `Z` | `a` | `b` | `c` | ··· | `z` |
|---|---|---|---|---|---|---|---|---|---|
| 65 | 66 | 67 | ··· | 90 | 97 | 98 | 99 | ... | 122 |

Auch den Satzzeichen werden Zahlen zugeordnet, z.B. hat das Ausrufezeichen die Nummer 33 und ein Zwischenraum die Nummer 32. Auf diese Weise wird jedes Schriftzeichen durch eine Zahl zwischen 0 und 255 dargestellt. Der Text

```
Alice und Bob telefonieren.
```

wird, inklusive Zwischenräumen und Punkt, in die Folge

$$65\ 108\ 105\ 99\ 101\ 32\ 117\ 110\ 100\ 32\ 66\ 111\ 98\ 32$$
$$116\ 101\ 108\ 101\ 102\ 111\ 110\ 105\ 101\ 114\ 101\ 110\ 46$$

übersetzt, die wir mathematisch als

$$(65, 108, 105, 99, 101, 32, 117, 110, 100, 32, 66, 111, 98, 32,$$
$$116, 101, 108, 101, 102, 111, 110, 105, 101, 114, 101, 110, 46)$$

schreiben. Wir können uns nun also vorstellen, dass Alice ihr ganzes Lexikon in eine einzige lange Folge

$$T_A = (a_1, a_2, \ldots, a_{n-1}, a_n)$$

von Zahlen zwischen 0 und 255 übersetzt und dass Bob das Gleiche mit seinem Lexikon gemacht hat:

$$T_B = (b_1, b_2, \ldots, b_{n-1}, b_n).$$

Dabei ist $n$ ungefähr 50 Millionen. Derart lange Folgen können wir hier natürlich nicht aufschreiben. Wir nehmen als Beispiel zwei Folgen der Länge $n = 8$:

Texte („Adelaide“ und „Barnsley“) als Zahlenfolgen

$$T_{\mathrm{Ad}} = (a_1, a_2, \ldots, a_8) = (65, 100, 101, 108, 97, 105, 100, 101)$$
$$T_{\mathrm{Ba}} = (b_1, b_2, \ldots, b_8) = (66, 97, 114, 110, 115, 108, 101, 121).$$

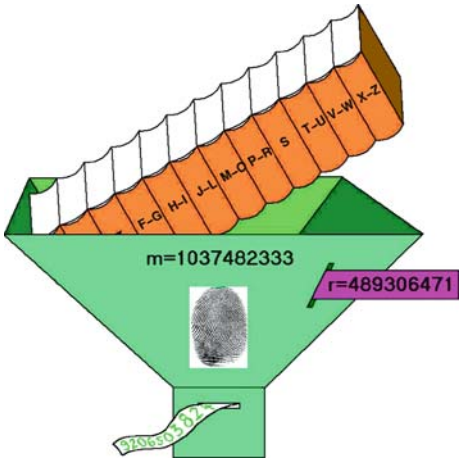
Mit den Zahlenfolgen wollen wir rechnen. Dazu brauchen wir eine Methode, die in Kap. 17 schon angeklungen ist und in Kap. 25 noch genauer erkärt wird: *Modulo-Rechnung* oder *Modulare Arithmetik*. Arithmetik modulo einer ganzen Zahl $m > 1$ heißt, dass man für eine ganze Zahl $a$ den Rest bei der Division von $a$ durch $m$ ausrechnet, bzw. zählt, wie viele Schritte man auf der Zahlengeraden von $a$ nach links gehen muss, bis man auf ein Vielfaches von $m$ trifft. Wenn z. B. $m = 7$ ist, hat man $16 \bmod 7 = 2$ und $-4 \bmod 7 = 3$. In der folgenden Tabelle sind noch mehr Werte aufgeschrieben. Man erkennt das Muster: Wenn man die Zahlengerade entlangspaziert, gehen die Restwerte in $\{0, 1, \ldots, m-1\}$ immer im Kreis herum.

| $a$ | … | $-4$ | $-3$ | $-2$ | $-1$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | … |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $a \bmod 7$ | … | 3 | 4 | 5 | 6 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 0 | 1 | 2 | 3 | … |

„Modulare Arithmetik“ bedeutet dann, dass man Zahlen „modulo $m$“ addiert und multipliziert. Das geht folgendermaßen: Man addiert und multipliziert ganz normal und berechnet vom Ergebnis den Rest bei der Division durch $m$. Zum Beispiel ist $3 \cdot (-6) \bmod 7 = (-18) \bmod 7 = 3$. Um sich die Arbeit zu erleichtern, darf man bei längeren Rechnungen auch unterwegs Zwischenergebnisse immer durch ihren Rest ersetzen. Um z. B. $(6 \cdot 5 + 5 \cdot 4) \bmod 7$ zu berechnen, rechnet man $6 \cdot 5 \bmod 7 = 30 \bmod 7 = 2$ und $5 \cdot 4 \bmod 7 = 20 \bmod 7 = 6$ und erhält das Ergebnis als $(2 + 6) \bmod 7 = 8 \bmod 7 = 1$.

## Fingerabdrücke

Nun wollen wir modulare Arithmetik auf unsere Beispieltexte $T_{\mathrm{Ad}}$ und $T_{\mathrm{Ba}}$ anwenden. Wir legen eine Zahl $m$ fest. Später werden wir sehen, dass $m$ eine Primzahl sein sollte, die größer als 255 und größer als $n$ ist, vielleicht etwa so groß wie $2n$ oder $10n$. Für das Beispiel wählen wir aber $m = 17$, um die Rechnung anschaulich zu halten.




Für $r = 0, 1, 2, \ldots, m-1$ und einen Text $T = (a_1, a_2, \ldots, a_{n-1}, a_n)$ sehen wir die folgende Zahl an:

$$\mathrm{FP}_m(T, r) = (a_1 \cdot r^n + a_2 \cdot r^{n-1} + \cdots + a_{n-1} \cdot r + a_n \cdot r) \bmod m.$$

Zum Beispiel bekommen wir für $T_{\mathrm{Ad}}$ und $r = 3$: $\mathrm{FP}_m(T_{\mathrm{Ad}}, 3) =$

$$(65 \cdot 3^8 + 100 \cdot 3^7 + 101 \cdot 3^6 + 108 \cdot 3^5 + 97 \cdot 3^4 + 105 \cdot 3^3 + 100 \cdot 3^2 + 101 \cdot 3) \bmod 17.$$

Es ist wichtig, sich gleich klarzumachen, dass die Textlänge $n$ sehr groß sein kann, aber die Anzahl der Ziffern von $m$ und damit auch die Anzahl der Ziffern der Zahl $\mathrm{FP}_m(T, r)$ recht klein ist. Wir nennen (die Dezimaldarstellung von) $\mathrm{FP}_m(T, r)$ einen „Fingerabdruck“ (engl.: „**F**inger**p**rint“) für den Text $T = (a_1, a_2, \ldots, a_n)$ (mit $r$ berechnet).

Die Idee hinter diesem Namen ist, dass in der Zahl $\mathrm{FP}_m(T, r)$ auf kleinem Raum eine gewisse Information über $T$ gespeichert ist, die wir vielleicht nutzen können, um $T$ von anderen Texten zu unterscheiden, so wie ein kleiner Fingerabdruck genügt, um einen Menschen von anderen zu unterscheiden. Ein ganz primitiver „Fingerabdruck“ ist natürlich auch die Länge $n$ des Textes.

Die Berechnung von $\mathrm{FP}_m(T, r)$ sieht, besonders für die langen Texte, die uns eigentlich interessieren, ziemlich gefährlich und teuer aus, weil durch die hohen Potenzen riesengroße Zahlen entstehen. Hier hilft ein recht einfacher Trick, nämlich geschicktes Ausklammern:

$$\mathrm{FP}_m(T, r) = ((((\cdots(((a_1 \cdot r) + a_2) \cdot r) + \cdots) \cdot r + a_{n-1}) \cdot r + a_n) \cdot r) \bmod m.$$

Wenn man diesen Ausdruck wie üblich von innen nach außen fortschreitend ausrechnet und „unterwegs“ bei jedem Zwischenergebnis den Rest „modulo $m$“ bildet, kann man die Zwischenergebnisse klein halten. Beispielsweise ist $\mathrm{FP}_m(T_{\mathrm{Ad}}, 3) =$

$$(((((((((65{\cdot}3)+100){\cdot}3+101){\cdot}3+108){\cdot}3+97){\cdot}3+105){\cdot}3+100){\cdot}3+101){\cdot}3) \bmod 17,$$

und mit $r = 3$ ergeben sich die folgenden Rechenschritte:

| | Werte | Zwischenergebnis | |
|---|---|---|---|
| $a_1$ | 65 | $(65 \cdot 3) \bmod 17$ | $= (14 \cdot 3) \bmod 17 = 8$ |
| $a_2$ | 100 | $((8 + 100) \cdot 3) \bmod 17$ | $= (6 \cdot 3) \bmod 17 = 1$ |
| $a_3$ | 101 | $((1 + 101) \cdot 3) \bmod 17$ | $= (0 \cdot 3) \bmod 17 = 0$ |
| $a_4$ | 108 | $((0 + 108) \cdot 3) \bmod 17$ | $= (6 \cdot 3) \bmod 17 = 1$ |
| $a_5$ | 97 | $((1 + 97) \cdot 3) \bmod 17$ | $= (13 \cdot 3) \bmod 17 = 5$ |
| $a_6$ | 105 | $((5 + 105) \cdot 3) \bmod 17$ | $= (8 \cdot 3) \bmod 17 = 7$ |
| $a_7$ | 100 | $((7 + 100) \cdot 3) \bmod 17$ | $= (5 \cdot 3) \bmod 17 = 15$ |
| $a_8$ | 101 | $((15 + 101) \cdot 3) \bmod 17$ | $= (14 \cdot 3) \bmod 17 = 8$ |

Also ist $\mathrm{FP}_{17}(T_{\mathrm{Ad}}, 3) = 8$.

Hier ist der Algorithmus zur Berechnung eines Fingerabdrucks $\mathrm{FP}_m(T, r)$:

Algorithmus FP berechnet einen Fingerabdruck $\mathrm{FP}_m(T, r)$

```
1  procedure FP(m, T, r)
2  begin
3      fp := (a_1 · r) mod m;
4      for i from 2 to n do
5          fp := ((fp + a_i) · r) mod m;
6      endfor
7  return fp
8  end
```

Natürlicherweise ist der Fingerabdruck selber immer ein Rest modulo $m$, also eine Zahl zwischen 0 und $m-1$, und auch die Zwischenresultate bei der Berechnung sind nicht größer als $m^2$.

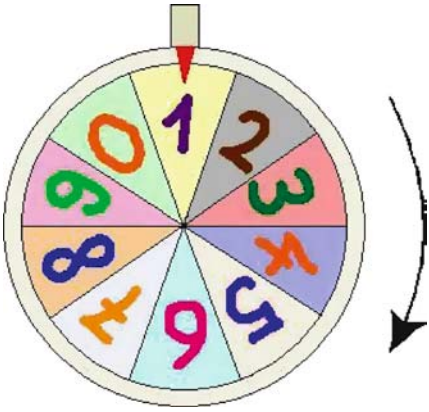
Mit diesem Verfahren rechnen wir einmal alle $m = 17$ Fingerabdrücke für $T_{\text{Ad}}$ und für $T_{\text{Ba}}$ aus.

| $r$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\text{FP}_m(T_{\text{Ad}}, r)$ | 0 | 12 | 7 | 8 | 11 | 14 | 15 | 5 | 11 | 1 | 2 | 12 | 13 | 13 | 6 | 6 | 0 |
| $\text{FP}_m(T_{\text{Ba}}, r)$ | 0 | 16 | 9 | 2 | 2 | 6 | 14 | 3 | 11 | 12 | 2 | 10 | 11 | 15 | 2 | 10 | 11 |

(Alice könnte die erste Zeile ausrechnen, Bob die zweite.) Wir vergleichen die Werte für $T_{\text{Ad}}$ und $T_{\text{Ba}}$. Bei $r = 0$ steht da in beiden Zeilen 0 – das ist keine Überraschung, da im Algorithmus FP die letzte Aktion eine Multiplikation mit $r$ ist. Ein Fingerabdruck mit $r = 0$ enthält also keine Information, und man braucht $r = 0$ gar nicht zu betrachten. Sonst kann man an den Zahlen kein rechtes System erkennen. Wir vergleichen übereinander stehende Zahlen. Bei $r = 3$ (unser Beispiel) ist $\text{FP}_{17}(T_{\text{Ad}}, r) = 8$ und $\text{FP}_{17}(T_{\text{Ba}}, r) = 2$. Der Fingerabdruck mit $r = 3$ würde also helfen zu sagen, dass die Texte verschieden sind. Bei $r = 8$ und bei $r = 10$ kommt aber dasselbe Ergebnis heraus (11 bzw. 2), diese Werte von $r$ helfen also nicht.

## Fingerabdrücke mit Zufallszahlen

Jetzt kommt die entscheidende Idee. Nehmen wir einmal an, Alice und Bob machen sich überhaupt nicht die Mühe, alle Werte in der Tabelle auszurechnen. (Bei größeren $n$ und $m$ geht das sowieso aus Zeitgründen nicht.) Alice wählt aber eine Zahl $r$ zwischen 1 und $m-1$ *zufällig*. Sie könnte z. B. wiederholt ein Glücksrad drehen, dessen Rand in zehn gleichgroße Teile unterteilt ist, um die Dezimalziffern von $r$ zu bestimmen.




(In allen Programmiersprachen gibt es eine Operation zur Erzeugung von Zufallszahlen. Kapitel 25 beschäftigt sich mit der Frage, was dahinter steckt.)

Alice ruft Bob an und sagt ihm, welches $r$ sie gewählt hat. Dazu muss sie nur wenige Dezimalziffern nennen. Anschließend berechnet Alice mit Hilfe ihres Computers die Zahl $\mathrm{FP}_m(T_A, r)$ und gleichzeitig berechnet Bob $\mathrm{FP}_m(T_B, r)$. Das dauert möglicherweise eine Weile, kostet aber keine Kommunikation und keine Telefongebühren. Dann ruft Alice Bob erneut an und nennt ihm „ihren" Fingerabdruck $\mathrm{FP}_m(T_A, r)$. Nun gibt es verschiedene Möglichkeiten.

1. *Fall*: Die Texte $T_A$ und $T_B$ sind gleich. Dann haben Alice und Bob immer die gleichen Resultate erhalten, ganz egal, welches $r$ Alice gewählt hat.
2. *Fall*: Die Texte $T_A$ und $T_B$ sind verschieden (im Beispiel „Adelaide" und „Barnsley" bei $m = 17$).
   - Wenn Alice eine Zahl $r$ mit $\mathrm{FP}_m(T_A, r) = \mathrm{FP}_m(T_A, r)$ gewählt hat (im Beispiel: $r = 8$ oder $r = 10$), dann bekommt Bob denselben Fingerabdruck wie Alice heraus, und es sieht für die beiden so aus, als könnten die Texte gleich sein.
   - Wenn Alice eine Zahl $r$ mit $\mathrm{FP}_m(T_A, r) \neq \mathrm{FP}_m(T_A, r)$ gewählt hat (im Beispiel: eine der anderen 14 Zahlen), dann bekommt Bob einen anderen Fingerabdruck heraus und er kann Alice mitteilen, dass die Texte mit Sicherheit verschieden sind.

In unserem kleinen Beispiel ist die Chance, dass der Unterschied gefunden wird, 14:16, also 87,5 Prozent. Wie stehen die Chancen, einen Unterschied zu finden, im allgemeinen Fall? Um hierzu etwas sagen zu können, müssen wir etwas tiefer in die mathematische Trickkiste greifen. Man kann beweisen, dass Folgendes passiert:

**Fingerprinting-Satz**

Wenn $T_A$ und $T_B$ verschiedene Texte (Zahlenfolgen) der Länge $n$ sind, und wenn $m$ eine Primzahl ist, die größer ist als die größte Zahl in $T_A$ und $T_B$, dann können von den $m$ Zahlenpaaren

$$\mathrm{FP}_m(T_A, r), \mathrm{FP}_m(T_B, r), \quad r = 0, 1, \ldots, m-1,$$

höchstens $n$ viele aus gleichen Zahlen bestehen.

Diese mathematische Tatsache ist schon seit mehr als 200 Jahren bekannt. Sie ist eine der einfacheren der vielen wunderbaren Eigenschaften, die Primzahlen haben.

Wie man den Fingerprinting-Satz beweist, ist für Alice und Bob bzw. für unsere Überlegungen eigentlich egal, da der Beweis für den Algorithmus keine Rolle spielt. Im letzten Abschnitt dieses Kapitels gibt es eine Skizze einer Begründung dafür, dass der Fingerprinting-Satz wahr ist.

Für unser Beispiel mit $n = 8$ heißt das Folgendes. Ganz egal wie $T_A$ und $T_B$ aussehen: Wenn sie verschieden sind, gibt es in unserer Tabelle nie mehr als 7 Werte $r \neq 0$, die Alice und Bob zum selben Ergebnis führen. Die Chance, dass der Unterschied gefunden wird, ist also immer mindestens 9:16, also mehr als

50 Prozent. Halt, das stimmt nicht ganz! Weil wir für das Beispiel $m$ sehr klein gewählt haben und nicht größer als 255, wie im Fingerprinting-Satz verlangt, gilt die Grenze 7 nur für Paare $T_A$ und $T_B$, für die $a_i \bmod 17 \neq b_i \bmod 17$ für mindestens ein $i$ gilt. Dieses Problem verschwindet, wenn $m$ größer als 255 gewählt wird.

Nun kehren wir zur Anfangssituation mit Texten der Länge $n \approx 50$ Millionen zurück. Damit jetzt etwas Vernünftiges herauskommt, müssen Alice und Bob $m$ um einiges größer als $n$ wählen, sagen wir: $m$ ist eine Primzahl etwas größer als 1 Milliarde, z.B. $m = 1037482333$. Für so große $n$ und $m$ können und wollen wir die Tabelle der $\mathrm{FP}_m(T, r)$-Werte bestimmt nicht mehr aufschreiben. Aber nach dem Fingerprinting-Satz wissen wir auch ohne die Tabelle anzusehen, dass es unter den $m$ Spalten höchstens $n$ viele gibt, in denen die Werte $\mathrm{FP}_m(T_A, r), \mathrm{FP}_m(T_B, r)$ übereinstimmen. Eine davon ist die Spalte für $r = 0$.

Wenn nun Alice $r$ zwischen 1 und $m-1$ zufällig wählt und Alice und Bob rechnen und sich Zahlen und Fingerabdrücke mitteilen wie eben beschrieben, dann ist die Wahrscheinlichkeit, dass Alice einen der „schlechten“ Werte für $r$ erwischt und die beiden nicht merken, dass die Texte verschieden sind, höchstens

$$\frac{n-1}{m-1} \approx \frac{50000000}{1000000000} = 0{,}05,$$

also 5%. Die Chance, dass sie den Unterschied entlarven, ist also mindestens 95 Prozent.

Alice und Bob müssen zwar viel rechnen bzw. ihre Computer rechnen lassen, aber sie müssen nicht viel Information austauschen: Alice muss Bob die Zahl $n$ (8 Ziffern) und die Primzahl $m$ nennen (10 Ziffern), und sie muss ihm die beiden Zahlen $r$ und $\mathrm{FP}_m(T_A, r)$ vorlesen (20 Ziffern).

Alice muss Bob weniger als **40 Dezimalziffern** mitteilen,
und erreicht damit eine Irrtumswahrscheinlichkeit von weniger als 5%!

Wenn man mit einer Chance auf Entlarvung von unterschiedlichen Texten von 95 Prozent nicht zufrieden ist, gibt es eine auch nicht allzu teure Verbesserungsmöglichkeit: Alice wählt **zwei** Zahlen $r_1$ und $r_2$ zufällig und nennt Bob diese beiden und die Werte $\mathrm{FP}_m(T_A, r_1)$ und $\mathrm{FP}_m(T_A, r_2)$. Bob erklärt die beiden Texte für gleich (wobei er sich irren kann), wenn für $T_B$ bei $r_1$ und $r_2$ dieselben Zahlen herauskommen. Die Chance, dass Bob irrtümlich zwei verschiedene Texte für gleich hält, ist höchstens

$$\frac{(n-1)^2}{(m-1)^2} < \left(\frac{n}{m}\right)^2 \approx 0.05^2 = 0{,}0025,$$

die Chance, den Unterschied zu entdecken, ist also 99,75 Prozent. Wenn Alice gar drei Zahlenpaare (insgesamt weniger als 80 Ziffern) schickt, fällt die Wahrscheinlichkeit, sich zu irren, auf höchstens $(n^3/m^3) \approx 0{,}000125$ oder 0,0125 Prozent; die Entlarvungschance steigt auf 99,9875 Prozent.

## Das Protokoll

Wir fassen die Methode von Alice und Bob, ihre Texte auf Identität zu testen, noch einmal zusammen. Da es um eine Kombination von Berechnungen und Kommunikation geht, spricht man nicht von einem Algorithmus, sondern von einem „Protokoll“ (im Sinne einer Vorschrift darüber, wie sich die Beteiligten verhalten sollen).

**Protokoll Textvergleich mit Fingerprinting**

Alice hat die Zahlenfolge $T_A = (a_1, \ldots, a_n)$ mit Ziffern zwischen 0 und $d-1$.
Bob hat die Zahlenfolge $T_B = (b_1, \ldots, b_{n'})$ mit Ziffern zwischen 0 und $d-1$.

1. Alice sagt Bob, was $n$ ist. Wenn $n \neq n'$ ist, sagt Bob „ungleich“ und STOP.
2. Alice und Bob einigen sich auf eine Wiederholungsanzahl $k$.
3. Alice sucht eine Primzahl $m$, die größer als $d$ und $10n$ ist. Sie wählt $k$ Zahlen $r_1, \ldots, r_k$ zwischen 1 und $m-1$ zufällig, und nennt Bob die Zahlen $m$ und $r_1, \ldots, r_k$.
4. Alice berechnet $\mathrm{FP}_m(T_A, r_1), \ldots, \mathrm{FP}_m(T_A, r_k)$. (Sie ändert dazu den Algorithmus FP so ab, dass sie mit *einem* Durchlauf durch den Text $T_A$ alle $k$ Resultate berechnet.)
5. Bob berechnet $\mathrm{FP}_m(T_B, r_1), \ldots, \mathrm{FP}_m(T_B, r_k)$.
6. Alice übermittelt Bob ihre $k$ Resultate.
7. Bob vergleicht mit seinen $k$ Werten. Wenn es Unterschiede gibt, sagt er „ungleich“ und STOP. Wenn alle Werte gleich sind, sagt er „kein Unterschied zu sehen“ und STOP.

Über das Ergebnis des Protokolls kann man Folgendes sagen.

- Wenn Alice und Bob denselben Text haben, dann stimmen die von den beiden berechneten Fingerabdrücke paarweise überein. Also ist das Ergebnis des Protokolls immer „kein Unterschied zu sehen“.
- Wenn Alice und Bob verschiedene Texte haben, dann gibt es nach dem Fingerprinting-Satz unter den $m-1$ Zahlen, aus denen Alice wählt, höchstens $n-1$ viele Werte für $r$, bei denen die Fingerabdrücke $\mathrm{FP}_m(T_A, r)$ und $\mathrm{FP}_m(T_B, r)$ übereinstimmen. Also gilt für ein zufälliges $r$, dass mit Wahrscheinlichkeit höchstens $(n-1)/(m-1)$ die Ergebnisse $\mathrm{FP}_m(T_A, r)$ und $\mathrm{FP}_m(T_B, r)$ gleich sind. Die Wahrscheinlichkeit, dass bei allen $k$ Versuchen ein $r$ gewählt wird, das dieselben Fingerabdrücke liefert, dass also Alice und Bob insgesamt das (unerwünschte) Resultat „kein Unterschied zu sehen“ bekommen, ist höchstens

$$\frac{(n-1)^k}{(m-1)^k} = \left(\frac{n-1}{m-1}\right)^k < \left(\frac{n}{m}\right)^k \text{, also } < \frac{1}{10^k}.$$

Mit einer passenden Wahl von $k$ können die beiden also die Irrtumswahrscheinlichkeit so winzig einstellen, wie sie möchten.

Wenn $m \approx 10n$ ist, und $n$ genau $l$ Dezimalziffern hat, und die Irrtumswahrscheinlichkeit höchstens $10^{-k}$ sein soll, genügt es, wenn Alice $(l+1) \cdot (2+2k)$ Ziffern übermittelt. Es ist erstaunlich, wie gutmütig diese Formel auf eine Vergrößerung der Textlänge reagiert: Wenn wir einen 10-mal so langen Text zu vergleichen haben, d. h., wenn $l$ um 1 wächst, vergrößert sich die Anzahl der Ziffern, die übermittelt werden müssen, nur um $2k$.

## Fazit

- Wenn man beim Textvergleich absolute Sicherheit haben will, kann man Datenkompressionsverfahren benutzen, spart aber bei gewöhnlichen Texten nicht mehr als einen Faktor 5 bei der Länge des zu übermittelnden Textes ein.
- Wenn es akzeptabel ist, dass man mit einer (sehr) kleinen Wahrscheinlichkeit verschiedene Texte irrtümlich für gleich hält, dann kann man ein Fingerabdruck-Verfahren benutzen. Dieses verkürzt die zu übermittelnden Botschaften dramatisch.
- Bei Texten der Länge $n$ verwendet man eine Primzahl $m > n$. Die Irrtumswahrscheinlichkeit ist höchstens $\left(\frac{n}{m}\right)^k$, wenn man $k$ Fingerabdrücke verschickt. Man muss dann $(2+2k)$ Zahlen übermitteln, die höchstens so groß wie $m$ sind.
- Die Verwendung von *Zufall* in Algorithmen und Kommunikationsprotokollen kann zu wesentlichen Einsparungen führen, wenn man bereit ist, in Kauf zu nehmen, dass mit einer kleinen Wahrscheinlichkeit ein falsches Ergebnis auftritt. Oft kann man recht einfach (z. B. durch Wiederholung) die Fehlerwahrscheinlichkeit so klein machen, dass sie nicht mehr stört.
- Algorithmen oder Protokolle, die manche Entscheidungen oder Auswahlen zufällig treffen, heißen „randomisiert“ (von englisch „*at random*“ = zufällig). In Kap. 25 wird diskutiert, wie „der Zufall in den Rechner kommt“, d. h., wie man den Computer dazu bringt, „zufällige“ Zahlen zu erzeugen.
- Manchmal helfen sehr abstrakt aussehende mathematische Tatsachen, deren Anwendbarkeit alles andere als offensichtlich ist, um ganz konkret Rechen- und Kommunikationsaufwand einzusparen.

## Bemerkungen zum Fingerprinting-Satz

Wir können hier keinen vollständigen Beweis des Fingerprinting-Satzes geben, sondern nur andeuten, weshalb er gilt. Hierzu betrachten wir *Polynome*, genauer gesagt „rationale Polynome“. Diese kann man sich als Ausdrücke

$$f(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0$$

vorstellen, wobei die „Koeffizienten“ $c_n, c_{n-1}, \ldots, c_1, c_0$ irgendwelche rationalen Zahlen, also Brüche, sind. Beispielsweise sind folgende Ausdrücke rationale Polynome:

$$2x^2 + \tfrac{3}{2}, \quad \tfrac{3}{4}x - \tfrac{1}{10}, \quad x^5 + 4x^4 - 3x^2 - \tfrac{15}{29}x + \tfrac{1}{3}, \quad \tfrac{7}{8}, \quad 0.$$

Beim vorletzten Beispiel ($\frac{7}{8}$) ist $n = 0$ und $c_0 = \frac{7}{8}$; beim letzten Beispiel (0) gibt es überhaupt keine Terme, die nicht 0 sind. Wenn man Polynome schreibt, lässt man Terme $c_i x^i$, deren Koeffizient $c_i$ gleich 0 ist, meistens weg. Polynome kann man addieren und subtrahieren, indem man die üblichen Rechenregeln anwendet:

$$\begin{aligned}\left(2x^2 + \tfrac{3}{2}\right) + \left(-3x^2 + \tfrac{3}{4}x - 1\right) &= (2-3)\,x^2 + \tfrac{3}{4}x + \left(\tfrac{3}{2} - 1\right) = -x^2 + \tfrac{3}{4}x + \tfrac{1}{2},\\ \left(2x^2 + \tfrac{3}{2}\right) - \left(-3x^2 + \tfrac{3}{4}x - 1\right) &= (2+3)\,x^2 - \tfrac{3}{4}x + \left(\tfrac{3}{2} + 1\right) = 5x^2 - \tfrac{3}{4}x + \tfrac{5}{2}.\end{aligned}$$

Wenn man ein Polynom von sich selber subtrahiert, ergibt sich das Nullpolynom: $f(x) - f(x) = 0$. Natürlich kann man Polynome auch multiplizieren. Man „multipliziert aus“, nach den üblichen Regeln, und fasst dann die Koeffizienten zusammen, die bei derselben Potenz von $x$ stehen. Zum Beispiel:

$$(2x^2 + \tfrac{3}{2}) \cdot (\tfrac{3}{4}x^3 - x) = \tfrac{3}{2}x^5 + \tfrac{9}{8}x^3 - 2x^3 - \tfrac{3}{2}x = \tfrac{3}{2}x^5 - \tfrac{7}{8}x^3 - \tfrac{3}{2}x.$$

Wichtig ist noch der Vorgang des „Einsetzens“: Wenn $f(x)$ ein Polynom ist und $r$ eine rationale Zahl, schreiben wir $f(r)$ für das Resultat, das sich ergibt, wenn man in dem Ausdruck $f(x)$ für $x$ überall $r$ einsetzt und dann auswertet. Wenn also $f(x) = x^3 - \frac{1}{2}x^2 + 2x - 1$, dann ist $f(0) = -1$ und $f(\frac{1}{2}) = 0$ und $f(1) = \frac{3}{2}$. Eine rationale Zahl $r$ heißt eine *Nullstelle* von $f(x)$, wenn $f(r) = 0$ gilt. Zum Beispiel ist $r = \frac{1}{2}$ eine Nullstelle von $f(x) = x^3 - \frac{1}{2}x^2 + 2x - 1$.

Das Nullpolynom 0 hat natürlich unendlich viele Nullstellen. Das Polynom $f(x) = 10$ hat überhaupt keine, das Polynom $2x + 5$ hat genau eine Nullstelle, nämlich $r = -\frac{5}{2}$. Das Polynom $x^2 - 1$ hat zwei Nullstellen, nämlich 1 und $-1$; das Polynom $x^2 + 1$ hat keine einzige Nullstelle. Man kann Folgendes beweisen:

**Satz über die Anzahl der Nullstellen von Polynomen**

Wenn $f(x) = c_n x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$ mit $n \geq 0$ und $c_n \neq 0$ ein Polynom ist, dann hat $f$ höchstens $n$ verschiedene Nullstellen.

(Der Grund hierfür ist grob folgender: Wenn $r_1, \ldots, r_k$ verschiedene Nullstellen von $f(x)$ sind, dann kann man $f(x)$ als Produkt $(x - r_1) \cdots (x - r_k) \cdot g(x)$ schreiben, für ein Polynom $g(x) \neq 0$. Weil die höchste Potenz von $x$ in $f(x)$ aber $x^n$ ist, kann $k$ nicht größer als $n$ sein.)

Aus dem Satz über die Nullstellen können wir eine Folgerung ziehen: Wenn

$$\begin{aligned} g(x) &= c_n x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0 \text{ und}\\ h(x) &= d_n x^n + d_{n-1}x^{n-1} + \cdots + d_1 x + d_0\end{aligned}$$

verschiedene Polynome sind, dann gibt es höchstens $n$ verschiedene Zahlen $r$

mit $g(r) = h(r)$. Wieso? Wir bilden das Polynom

$$f(x) = g(x) - h(x) = (c_n - d_n)x^n + (c_{n-1} - d_{n-1})x^{n-1} + \cdots + (c_1 - d_1)x + (c_0 - d_0).$$

Es könnte natürlich $c_n = d_n$ und $c_{n-1} = d_{n-1}$ usw. sein, so dass viele Koeffizienten von $f(x)$ gleich 0 sind. Aber weil $g(x)$ und $h(x)$ verschieden sind, ist $f(x)$ nicht das Nullpolynom, und man kann $f(x) = e_k x^k + \cdots + e_1 x + e_0$ schreiben, mit $0 \leq k \leq n$ und $e_k \neq 0$. Also hat $f(x)$ nicht mehr als $k$, also auch nicht mehr als $n$ Nullstellen. Nun gilt für jedes $r$: Wenn $g(r) = h(r)$, dann ist $f(r) = g(r) - h(r) = 0$, also ist $r$ eine Nullstelle von $f(x)$. Daraus folgt, dass es nicht mehr als $n$ Zahlen $r$ mit $g(r) = h(r)$ geben kann.

Das ist ja fast die Formulierung des Fingerprinting-Satzes! Der einzige Unterschied ist, dass im Fingerprinting-Satz von Rechnungen modulo $m$ die Rede ist anstelle von Rechnungen mit rationalen Zahlen. Die Sache verhält sich aber so: Damit die Überlegung mit der Anzahl der Nullstellen stimmt, braucht man nicht unbedingt die rationalen Zahlen, sondern nur einen Bereich, in dem man ohne Einschränkungen addieren, subtrahieren, multiplizieren und dividieren kann. Man kann überlegen, dass Arithmetik modulo $m$ diese Eigenschaften hat, falls $m$ eine Primzahl ist, so dass der Satz über die Anzahl der Nullstellen eines Polynoms auch dort gilt.

## Zum Weiterlesen

1. A. Steger: *Diskrete Strukturen, Band 1: Kombinatorik, Graphentheorie, Algebra*. Springer, Berlin Heidelberg New York, 2001.

   In Kap. 3.3 dieses Buchs werden Polynome beschrieben und der Satz über die Anzahl der Nullstellen von Polynomen wird bewiesen. In Kap. 5.3 wird erklärt, wieso man in der Menge $\{0, \ldots, m-1\}$ mit Addition und Multiplikation modulo $m$ auch eine Division hat und daher der Satz von der Nullstellenzahl auch für Polynome „modulo $m$“ gilt.

2. J. Hromkovič: *Randomisierte Algorithmen – Methoden zum Entwurf von zufallsgesteuerten Systemen für Einsteiger*. Teubner, Stuttgart Leipzig Wiesbaden, 2004.

   Dieses Buch beschreibt eine Vielzahl von randomisierten Algorithmen und Verfahren und ebenso die Grundlagen für die Untersuchung solcher Algorithmen.

3. Für diejenigen, die es ganz genau wissen wollen: Ein vollständiger Beweis des Fingerprinting-Satzes findet sich unter:
   `http://eiche.theoinf.tu-ilmenau.de/fingerprint/`

4. Eine vollständige Beschreibung des ASCII-Codes findet sich z. B. bei:
   `http://de.wikipedia.org/wiki/ASCII`

Grafik unter Mitwirkung von J. D.

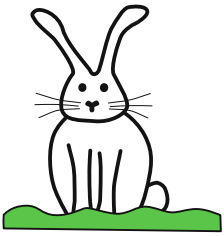
# 20

# Hashing

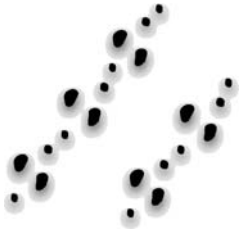
Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg

Um Hashing zu erklären, fangen wir mit einem Häschen an.



Einen Hasen zu finden, ist nicht leicht. Diese scheuen Tiere können sich sehr gut verstecken. Wenn man aufmerksam über ein verschneites Feld läuft, dann wird man vielleicht die folgenden Spuren finden:



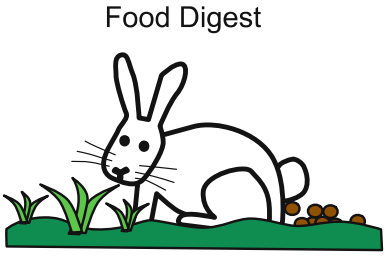
Hier sind zwei Hasen nebeneinander durch den Schnee gehoppelt. Aus einer Spur kann man allerhand über das Tier erfahren: Wie groß und wie schwer es ist, ob es in einer Gruppe unterwegs ist und vieles mehr. Manchmal findet man in solch einer Spur auch das:

Im Jägerlatein wird das Losung genannt. Es handelt sich um den Kot des Hasen. Aus der Losung lässt sich ablesen, was das Tier gefressen hat oder

ob es krank ist. Neuerdings kann man (nach einer Laboranalyse) damit auch ein Tier eindeutig identifizieren. Das funktioniert bei allen Tierarten durch die Analyse der DNS im Kot. (Damit möchte man beispielsweise in Madrid demnächst der hündischen Umweltverschmutzer habhaft werden.)

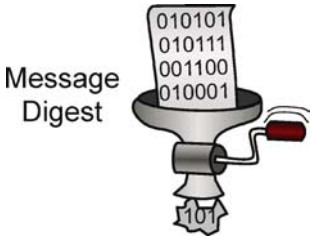
## Message Digest

Was hat das mit Hashing zu tun? Nun, eine Losung entsteht, indem der Hase frisst und die Losung ablegt. Also das:




Hierzu wird das Futter zerhackt, vermischt, zerrührt, verdaut, entwässert und ausgestoßen. Das Resultat ist ein kleiner Haufen. Das Endprodukt lässt sich der Eingabe, dem Futter, zuordnen. Hierbei geht natürlich eine ganze Menge an Information verloren; die Zuordnung ist aber immer noch möglich.

Dasselbe lässt sich auch mit digitalen Dokumenten, wie Text, Musik-Dateien und Film-Dateien durchführen. Für einen Informatiker sind das einfach Folgen von Nullen und Einsen, genannt Bits. Somit sind das alles Bitfolgen. Hierzu wird das Originaldokument durch eine Folge von Operationen vermengt, vermischt und verdichtet, bis eine Bitfolge fester Länge ausgeschieden wird. Das sieht in etwa so aus:




Ein bekannter Algorithmus, der diesen Vorgang durchführt, ist MD-5 (Message Digest 5), was soviel wie Nachrichtenverdauer Version 5 heißt. Er wurde 1991 von Ronald Rivest entwickelt und die genaue Beschreibung kann man bei Wikipedia nachlesen.[1] Andere bekannte Algorithmen sind SHA-1,

[1] siehe `http://de.wikipedia.org/wiki/MD5`

SHA-224, SHA-256, SHA-384 und SHA-512, übersetzt sind das „sichere Hack-Algorithmen“ (Secure Hash-Algorithm).[2] Sie sollen das Gleiche leisten wie ein Nachrichtenverdauer.

## Sicheres Hashing

Was leisten aber diese Hash-Algorithmen? Zuerst bilden sie Dateien unterschiedlicher Länge auf Bitfolgen fester Länge ab. Als zweites lassen sich aus dem Ergebnis die ursprünglichen Dateien identifizieren, aber nicht unbedingt rekonstruieren. Die ersten beiden Eigenschaften formulieren wir mathematisch:

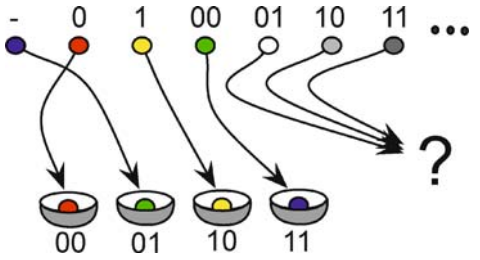
Mit $\{0,1\}^* = \{-, 0, 1, 00, 01, 10, 11, 0000, 0001, \ldots\}$ beschreiben wir die Menge aller Bitfolgen einschließlich der leeren Bitfolge „$-$“ für leere Dateien. Mit $\{0,1\}^k$ beschreiben wir die Menge alle Bitfolgen mit genau $k$ Bits, also: $\{000, 001, 010, 011, 100, 101, 110, 111\}$ im Fall von $k = 3$. Eine Hash-Funktion $f$ ist demnach eine Abbildung:

$$f : \{0,1\}^* \to \{0,1\}^k \ .$$

Was bedeutet es nun, dass das Ergebnis einer Hash-Funktion das Original eindeutig identifiziert?

Es heißt, dass genau dann und nur dann $f(x) = f(y)$ richtig ist, wenn $x = y$ ist. Das Gegenteil dieser Aussage ist: Es gibt zwei verschiedene Werte $x$ und $y$, so dass $f(x) = f(y)$ ist. Dann würden zwei Dateien $x$ und $y$ den gleichen Hash-Wert besitzen. Dieses Zusammentreffen wird Kollision genannt.

Eingangs forderten wir von einem Hash-Algorithmus, dass er beliebige Dateien auf Bitfolgen fester Länge so abbildet, dass die Ursprungsdatei identifizierbar ist. Wir behaupteten also, dass es eine Hash-Funktion ohne Kollisionen gibt. Mathematisch gesehen, ist diese Behauptung absoluter Unfug! Warum?

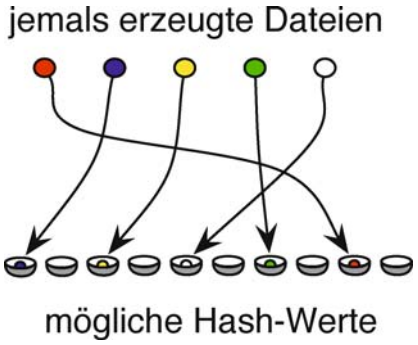



Stellen wir uns vor, dass jeder Hash-Wert eine Schale ist, dann gibt es so viele Schalen wie es Bitfolgen der Länge $k$ gibt. Im Fall von $k = 3$ zählen wir 8 Möglichkeiten, nämlich zwei Möglichkeiten für jedes Bit: $2 \cdot 2 \cdot 2 = 2^3 = 8$. Für allgemeine Längen $k$ erhalten wir genauso $2^k$ Möglichkeiten. Wir behaupten also, dass in jede dieser Schalen $f(x)$ nur eine Originalbitfolge hineingeworfen

[2] siehe `http://de.wikipedia.org/wiki/Secure_Hash_Algorithm`

wird. Davon gibt es aber unendlich viele, also mehr als 8 oder $2^k$. Damit müssen in mindestens einer der Schalen unendlich viele Originalwerte liegen und in dieser Schale treten unendlich viele Kollisionen auf.

Es gibt aber einen Ausweg! Wir wählen $k$ so groß, dass wir so viele Schalen haben, dass für jede Datei, die auf Rechnern jemals gespeichert wird, mindestens ein Hash-Wert vorhanden ist. Diese Diskussion kennen wir noch aus dem Kap. 14 mit der Einweg-Funktion. Also wählen wir z. B. $k = 512$. Dann gibt es $2^{512} > 10^{154}$ Schalen, das ist eine Zahl mit mindestens 154 Ziffern. Wenn es jetzt gelingt, diese Schalen so zu füllen, dass niemand (Mensch oder Rechner) eine Kollision konstruieren kann, sind wir aus dem Schneider.




Bis heute ist nicht klar, ob das funktionieren kann. Zwar gibt es mögliche Anwärter wie SHA-512 (mit 512 Bits für den Hash-Wert). Aber das heißt nicht viel. Zum Beispiel galt lange MD-5 als praktisch kollisionsfrei und heute kennt man Methoden zur Konstruktion von beliebig vielen Kollisionen. Man kann also eine ganze Reihe von Dateien konstruieren, deren Hash-Werte gleich sind.

Praktisch kollisionsfreie Hash-Funktionen sind in der Informatik sehr nützlich. So kann man damit die Korrektheit von übermittelten Dateien durch einen angehängten Hash-Wert belegen und dadurch Übertragungsfehler oder Fälschungsversuche entlarven.

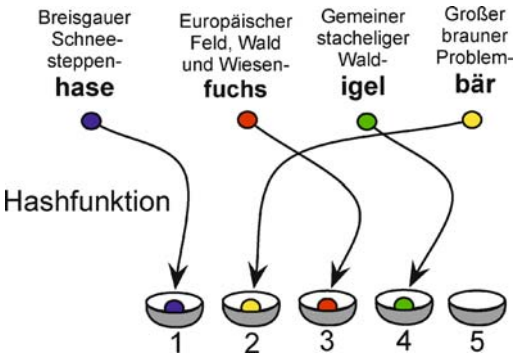
## Hashing für Wörterbücher

Hash-Funktionen bieten neben der Identifizierung von Dateien auch eine effiziente Möglichkeit zur Speicherung von Daten. Stellen wir uns vor, dass wir einen Speicherbereich $S$ zur Speicherung von insgesamt $m$ Daten zur Verfügung haben. Dieser Speicher ist als Tabelle oder Array organisiert. Man kann also auf den Speicherwert $S[1], \ldots, S[m]$ jeweils direkt zugreifen. In diesen Speicher wollen wir jetzt Daten ablegen, welche mit einem Wort als Schlüssel identifiziert werden.

Also z. B.:

| Tier | Anzahl |
|---|---|
| Breisgauer Schneesteppenhase | 12 |
| Europäischer Feld-, Wald und Wiesenfuchs | 2 |
| Gemeiner stacheliger Wildigel | 4 |
| Großer brauner Problembär | 1 |

Die Daten sind hier sehr kompakt, während der Suchindex (Suchwort) sehr lang ist. Angenommen wir hätten eine Hash-Funktion $f$, die Zeichenfolgen auf das Intervall $\{1, 2, 3, \ldots, m\}$ abbildet, d.h.




Dann könnten wir an die Speicherstelle $S[f(\text{„Breisgauer}\ldots\text{hase“})]$, also $S[1]$ den Wert 12 ablegen, an der Stelle $S[f(\text{„Europ}\ldots\text{fuchs“})]$, also $S[3]$, den Wert 2 in der Hash-Tabelle ablegen und so weiter. Diese Operation beschreiben wir mit Put.

Der Algorithmus Put speichert den Zahlenwert $z$ für das Suchwort $x$ mit Hilfe der Hash-Funktion $f$ und dem Speicherfeld $S[1], \ldots, S[m]$.

```
1  procedure Put (String x, Zahl z)
2  begin
3      S[f(x)] := z
4  end
```

Mit Get bekommen wir den Wert wieder, wobei der Wert 0 anzeigt, dass keine Dateien gespeichert sind.

Der Algorithmus Get holt den Wert für das Suchwort $x$ mit Hilfe der Hash-Funktion $f$ und dem Speicherfeld $S[1], \ldots, S[m]$.

```
1  procedure Get (String x)
2  begin
3      return S[f(x)]
4  end
```

Leider funktionieren die beiden Funktionen nur, wenn die Hash-Funktion absolut kollisionsfrei ist, also wenn z.B. Breisgauer ... hase und Gemeiner

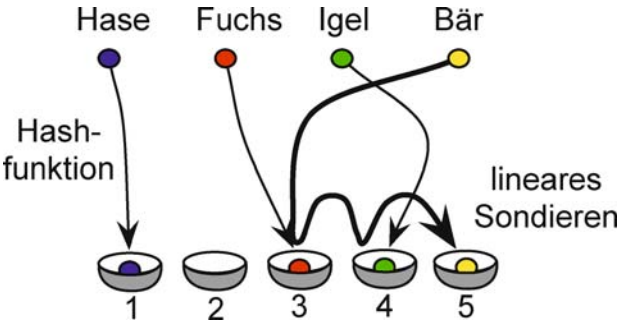
. . . igel nicht zusammen in einem Speicherplatz abgebildet werden. Das kann man für kleine Werte $m$ nur garantieren, wenn man die Schlüssel vorher kennt (also Breisgauer Schneesteppenhase, Europäischer Feld-, Wald und Wiesenfuchs, Gemeiner stacheliger Wildigel und Großer brauner Problembär) und dann eine so genannte perfekte Hash-Funktion wählt. Dieses Verfahren beschreiben wir jetzt etwas genauer.

## Kollisionsbehandlung

Treten Kollisionen auf, so muss man einen leeren Platz finden. Dafür gibt es mehrere Methoden. Die einfachste ist das so genannte lineare Sondieren. Man speichert hierfür in jedem Tabellenplatz das Datum und den Schlüssel.

### Speichern von Datum $z$ unter Schlüssel $x$

Zuerst berechnet man den Hash-Wert von $x$. Ist der Platz schon mit einem fremden Schlüssel besetzt, so geht man nach rechts weiter, bis man einen leeren Platz findet oder einen Platz mit dem gesuchten Schlüssel. Ist man ganz rechts, so springt man an die erste Stelle und fährt dort fort. Hat man den Tabellenplatz mit dem Schlüssel oder einen leeren Tabellenplatz schlussendlich gefunden, so schreibt man den Schlüssel $x$ und das Datum $z$ dorthin. Ist man einmal erfolglos außenrum gelaufen, so ist der Speicher voll. Das gibt dann eine Fehlermeldung.

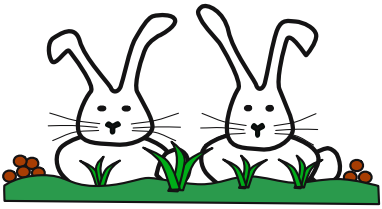



### Suchen von Datum unter Schlüssel x:

Wieder wird zuerst der Hash-Wert $f(x)$ von $x$ berechnet. Ist dort ein fremder Schlüssel, geht man nach rechts bis man den richtigen Schlüssel oder einen leeren Speicherplatz findet. Stößt man bei dieser Suche auf den rechten Rand, dann sucht man von vorne weiter. Die Suche ist erfolglos, wenn man auf einen leeren Speicherplatz stößt oder einmal rundherum gelaufen ist. Ansonsten hat man den Schlüssel $x$ gefunden und kann das Datum ausgeben.

Mit dieser Kollisionsbehandlung kann man nun immer $m$ Daten speichern, ganz unabhängig von der Güte der Hash-Funktion. Wer diese einmal selbst ausprobieren möchte, kann hierzu als Schlüssel die ganzen Zahlen und als Hash-Funktion die Modulo-$m$-Funktion benutzen. Das ist der Rest nach der Division durch $m$. Am Anfang wird das Speichern und Suchen erstaunlich schnell gehen. Aber wenn sich die Tabelle füllt, dann wird der Algorithmus immer langsamer werden.

Das liegt an der schlechten Kollisionsbehandlung. Denn die lineare Kollisionsbehandlung sucht immer wieder an denselben Stellen. In der Informatik kennt man bessere Lösungen, wie z.B. die quadratische Methode oder das doppelte Hashing. Dieses Kapitel schließen wir nun aber mit einem doppelten Häschen ...



## Zum Weiterlesen

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest: *Introduction to Algorithms*. The MIT Press, 2001 (1184 Seiten, ISBN 0-262-53196-8).

   Hier findet man viele schöne Algorithmen und insbesondere auch die Hash-Tabelle. In der deutschen Fassung heißt das Buch: Algorithmen – Eine Einführung.

2. Aus Wikipedia:
   - MD5:
     `http://de.wikipedia.org/wiki/MD5`
   - SHA-1:
     `http://de.wikipedia.org/wiki/MD5`
   - Hash-Tabelle:
     `http://de.wikipedia.org/wiki/Hash-Tabelle`

# 21

# Fehlererkennende Codes: Was ist eigentlich EAN?

Alexander Souza und Angelika Steger

Albert-Ludwigs-Universität Freiburg
ETH Zürich, Schweiz

Begegnen uns Algorithmen eigentlich auch im Alltag? Ja, aber meist merken wir (fast) gar nichts davon. Vermutlich hat jeder schon einmal auf Produktverpackungen, Fahrkarten und Postpaketen seltsame Ziffernfolgen, Strichcodes und andere kryptische Zeichen wahrgenommen. Dabei handelt es sich oft um fehlererkennende Codes.

Aber was nützen diese Codes? Ein Beispiel: Angenommen wir möchten das Buch „Introduction to Algorithms“ von Cormen, Leiserson, Rivest und Stein, eine umfangreiche Einführung in die Algorithmik, bestellen. Jedes Buch hat eine international einheitliche Identifikationsnummer, die EAN, die auch ISBN-13 genannt wird. (Das ISBN-13-System hat am 1.1.2007 das ISBN-10-System abgelöst). Nehmen wir an, dass wir uns die EAN dieses Titels notiert haben:

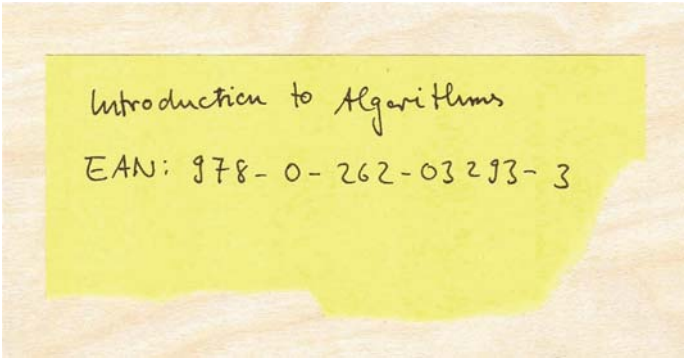



**Abb. 21.1.** Unser Notizzettel

Wir geben die Ziffern 978-0-362-03293-3 in eine Suchmaske ein und wundern uns, dass wir die Fehlermeldung „Ungültige EAN“ bekommen. Ach ja, wir haben uns bei der fünften Ziffer vertippt – das erklärt die Sache. Mit der

richtigen Nummer 978-0-262-03293-3 können wir das Buch bestellen. Durch das EAN-System konnte offenbar erkannt werden, dass wir uns vertan haben. Wie funktioniert dieses System eigentlich?

## Fehlererkennende Codes am Beispiel von EAN

EAN steht für International Article Number (bzw. stand früher für European Article Number). Diese Nummern werden verwendet, um Produkte aller Art (also nicht nur Bücher) international einheitlich und eindeutig zu kennzeichnen. Häufig werden sie in Klartext und in einem Strichcode angegeben, der maschinell lesbar ist.

Eine EAN besteht aus 13 Ziffern, die folgende Bedeutung haben: Internationale Lokationsnummer (7 Stellen), Artikelnummer (5 Stellen), Prüfziffer (1 Stelle). Die Lokationsnummern werden von einer internationalen Organisation, der GS1, vergeben und codieren das Ursprungsland und den Hersteller eines Produkts. Die Artikelnummer wählt der Hersteller eigenständig. Die Prüfziffer dient dazu, dass einzelne Tippfehler und einige Zahlendreher zu einer ungültigen EAN führen und somit – wie in obigem Beispiel – erkannt werden können.

Bevor wir uns mit der Berechnung dieser Prüfziffer näher beschäftigen, benötigen wir noch das Konzept der modularen Arithmetik aus der Mathematik. Modulare Arithmetik spielt auch in den Kap. 25 (Zufallszahlen) und 19 (Fingerprinting) eine Rolle.


**Abb. 21.2.** Eine EAN hat jeder schon einmal gesehen

### Modulare Arithmetik

Betrachten wir zwei beliebige ganze Zahlen $a$ und $m \geq 2$ und dividieren $a$ „mit Rest“ durch $m$. Dabei erhalten wir einen (ganzzahligen) Quotienten $q$ und einen Rest $r$, der eine natürliche Zahl zwischen 0 und $m-1$ ist, wobei $a = q \cdot m + r$ gilt. Wir sagen dann, dass die Zahl $a$ kongruent $r$ modulo $m$ ist.

Zur Veranschaulichung können wir uns dies bei positivem $a$ wie das zyklische Wandern auf einem Rad vorstellen, bei dem die Zahlen $0, 1, 2, \ldots, m-1$ aufgetragen sind. Um $a$ modulo $m$, also den Rest von $a$ bei der Division durch $m$, zu bestimmen, starten wir bei 0 und gehen dann $a$-mal (zyklisch) in Einerschritten im Uhrzeigersinn. Die Anzahl der Umrundungen des Rades ist der oben erwähnte Quotient, die Zahl, bei der wir landen, ist der Rest.

Beispielsweise ist 9 kongruent 1 modulo 8, weil 9 bei Division durch 8 den Rest 1 ergibt. Die Zahl 16 ist kongruent 0 modulo 8, weil 16 ohne Rest durch 8 teilbar ist.

Es haben sich zwei verschiedene Schreibweisen durchgesetzt, bei denen jeweils unterschiedliche Aspekte betont werden. Man schreibt $a \equiv b \pmod{m}$ wenn man zum Ausdruck bringen möchte, dass zwei ganze Zahlen $a$ und $b$ den gleichen Rest bezüglich Division durch $m$ haben (bzw. wenn $a-b$ durch $m$ teilbar ist). In einer anderen Notation ist mit $a \bmod m$ das eindeutig bestimmte $r \in \{0, 1, \ldots, m-1\}$ gemeint, für das $a \equiv r \pmod{m}$ gilt.

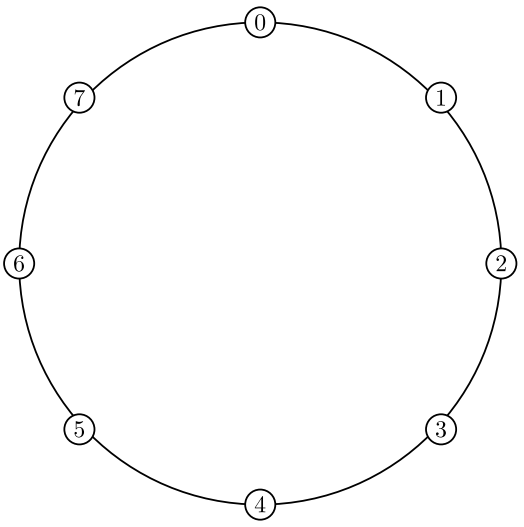



**Abb. 21.3.** Modulare Arithmetik als Rad

Unter modularer Arithmetik versteht man das Rechnen, also insbesondere Addition und Multiplikation, wobei das Ergebnis stets modulo einer Zahl $m$ genommen wird.

Wenn wir also $a+b$ modulo $m$ bestimmen wollen, berechnen wir zunächst die Zahl $(a+b)$ und von dieser den Rest modulo $m$. Entsprechend geht man bei der Multiplikation vor: erst multiplizieren, dann den Rest bei der Division durch $m$ bilden. Die Umkehrung führt übrigens zum gleichen Ergebnis: Wir können zunächst jeweils $a$ bzw. $b$ modulo $m$ bilden, dann rechnen und das Ergebnis erneut modulo $m$ nehmen. Dies hat den Vorteil, dass die Zwischenergebnisse unserer Rechnungen nicht so groß sind wie bei der ersten Methode.

### Prüfziffern bei EAN

Im Folgenden wird eine EAN durch eine Folge

$$x = x_1x_2x_3x_4x_5x_6x_7x_8x_9x_{10}x_{11}x_{12}x_{13},$$

bezeichnet, wobei die $x_i$ jeweils Ziffern zwischen 0 und 9 sind.

Die letzte Ziffer $x_{13}$ ist die Prüfziffer. Diese wird bei EAN berechnet, indem zunächst folgende Summe $s(x_1,\ldots,x_{12})$ der ersten 12 Ziffern gebildet wird:

$$s(x_1,\ldots,x_{12}) = (x_1 + x_3 + \cdots + x_{11} + 3\cdot(x_2 + x_4 + \cdots + x_{12})) \bmod 10.$$

Die Prüfziffer $x_{13}$ ist dann diejenige Ziffer, für die $s(x_1,\ldots,x_{12}) + x_{13}$ durch 10 teilbar ist. Formal wird $x_{13}$ also durch

$$x_{13} = (10 - (s(x_1,\ldots,x_{12}) \bmod 10)) \bmod 10$$

berechnet.

Bei der EAN 978-0-262-03293-? im Beispiel ergibt sich die Prüfziffer 3:

$$s = 9+8+2+2+3+9+3\cdot(7+0+6+0+2+3) = 87 \equiv 7 \pmod{10}$$
$$x_{13} = 10 - 7 \equiv 3 \pmod{10}$$

Die Prüfziffer der falschen Eingabe 978-0-362-03293-? ist 2, wie man leicht nachrechnet. Wenn man also die Folge 978-0-362-03293-3 sieht, kann man erkennen, dass etwas nicht stimmt.

### Erkennung von Tippfehlern

Jetzt überlegen wir uns, dass es immer erkannt werden kann, wenn wir uns bei der Eingabe einer EAN an genau einer Stelle vertan haben.

Wir schreiben für die (richtige) EAN $x = x_1x_2x_3\ldots x_{13}$. Wie oben erklärt, ist die Prüfziffer eine ganze Zahl zwischen 0 und 9. Die Ziffernfolge, die wir eingegeben haben (also die „falsche EAN“), bezeichnen wir mit $y = y_1y_2y_3\ldots y_{13}$.

Von einem Tippfehler sprechen wir, wenn wir uns an genau einer Ziffer geirrt haben, d.h., $y$ unterscheidet sich von $x$ an genau einer Stelle.

Nehmen wir zunächst an, dass wir uns bei der Prüfziffer selbst vertippt haben. In unserer Schreibweise heißt das $x_j = y_j$ für alle $j$ von 1 bis 12 und $x_{13} \neq y_{13}$. Dies kann mit Hilfe der ersten zwölf (richtigen) Ziffern erkannt werden: Wegen $s(x_1, \ldots, x_{12}) = s(y_1, \ldots, y_{12})$ muss natürlich auch $x_{13} = y_{13}$ gelten.

Nun nehmen wir an, dass wir uns bei genau der $i$-ten der zwölf ersten Ziffern vertan haben. Es gilt, dass $y_i = x_i + t$, wobei $t$ eine ganze Zahl im Bereich $-9$ bis 9 ist, aber ungleich Null (sonst wäre die $i$-te Stelle ja richtig). Alle anderen Positionen sind gleich, d.h., $y_j = x_j$ für $j \neq i$. Insbesondere ist $x_{13} = y_{13}$. Wir zeigen, dass sich die Prüfziffer der „falschen EAN“ $y$ aber von der Prüfziffer der richtigen EAN $x$ unterscheiden muss.

Wir berechnen die korrekte Prüfziffer der Zahlenfolge $y_1 \ldots y_{12}$. Zunächst nehmen wir an, dass die Fehlerposition $i$ ungerade ist und $x_i$ somit in der Summe $s(y_1, \ldots, y_{12})$ mit 1 gewichtet wird:

$$\begin{aligned} s(y_1, \ldots, y_{12}) &= (y_1 + y_3 + \cdots + y_i + \cdots + y_{11} \\ &\quad + 3 \cdot (y_2 + \cdots + y_{12})) \bmod 10 \\ &= (x_1 + x_3 + \cdots + x_i + t + \cdots + x_{11} \\ &\quad + 3 \cdot (x_2 + \cdots + x_{12})) \bmod 10 \\ &= (s(x_1, \ldots, x_{12}) + t) \bmod 10 \end{aligned}$$

Da $t$ im Bereich $-9$ bis 9 liegt, aber ungleich der Null ist, gilt für die Prüfziffer

$$y_{13} \equiv 10 - s(y_1, \ldots, y_{12}) \equiv 10 - s(x_1, \ldots, x_{12}) - t \equiv x_{13} - t \not\equiv x_{13} \pmod{10}.$$

Der andere Fall, wenn $i$ gerade ist und $x_i$ in $s(y_1, \ldots, y_{12})$ mit 3 gewichtet wird, verläuft analog, da es kein ganzzahliges $t$ zwischen $-9$ und 9 gibt, so dass $3 \cdot t$ durch 10 teilbar ist. (Hätte man als Gewichtung nicht 3, sondern z.B. 2 oder 5 gewählt, würde dies nicht gelten.) Damit haben wir gezeigt, dass sich die Prüfziffern unterscheiden, wenn wir uns an genau einer Stelle geirrt haben.

Anders sieht es aus, wenn wir uns an zwei oder mehr Stellen vertippt haben. Dann kann es passieren, dass die falsche EAN die gleiche Prüfziffer wie die richtige hat. Beispielsweise hat 978-0-262-34293-? ebenfalls die Prüfziffer 3 und unterscheidet sich von 978-0-262-03293-3 an genau der achten und neunten Ziffer.

Wenn wir uns an mehreren Stellen vertan haben, kann es sogar sein, dass wir ein unerwünschtes Buch bekommen, das zufälligerweise die EAN hat, die wir eingegeben haben. Hätten wir versehentlich 978-0-262-03295-7 eingegeben, wäre das Buch „Melancholia and Moralism“ geliefert worden.

### Erkennung von Zahlendrehern

Ein weiterer „beliebter“ Fehler bei der Eingabe sind Zahlendreher, d.h. die Vertauschung von benachbarten Ziffern. Beispielsweise entsteht die falsche EAN 978-0-226-03293-3 durch Vertauschen der sechsten und siebten Ziffer.

Mit der Prüfziffer bei EAN können manche, aber leider nicht alle Zahlendreher erkannt werden. In diesem Fall aber schon: Die korrekte Prüfziffer von 978-0-226-03293-? ist 1.

Welche Zahlendreher können erkannt werden, welche nicht? Betrachten wir allgemein zwei Folgen $x$ und $y$, bei denen zwei benachbarte Ziffern $x_i = y_{i+1}$ und $x_{i+1} = y_i$ vertauscht sind, die aber ansonsten identisch sind. Für die folgende Rechnung wird angenommen, dass $i$ ungerade ist. Der Fall, dass $i$ gerade ist, wird ähnlich behandelt. Wir berechnen wieder die korrekte Prüfziffer der Zahlenfolge $y_1 \dots y_{12}$:

$$\begin{aligned} s(y_1, \dots, y_{12}) &= (y_1 + y_3 + \cdots + y_i + \cdots + y_{11} \\ &\quad + 3 \cdot (y_2 + \dots y_{i+1} + \cdots + y_{12})) \bmod 10 \\ &= (x_1 + x_3 + \cdots + x_{i+1} + \cdots + x_{11} \\ &\quad + 3 \cdot (x_2 + \cdots + x_i + \cdots + x_{12})) \bmod 10 \\ &= (x_1 + x_3 + \cdots + x_{i+1} + \cdots + x_{11} \\ &\quad + 3 \cdot (x_2 + \cdots + x_i + \cdots + x_{12}) \\ &\quad + 2x_{i+1} - 2x_i - 2x_{i+1} + 2x_i) \bmod 10 \\ &= (s(x_1, \dots, x_{12}) + 2 \cdot (x_i - x_{i+1})) \bmod 10 \end{aligned}$$

Demnach gilt für die Prüfziffer

$$\begin{aligned} y_{13} &\equiv 10 - s(y_1, \dots, y_{12}) \equiv 10 - s(x_1, \dots, x_{12}) + 2(x_i - x_{i+1}) \\ &\equiv x_{13} + 2(x_i - x_{i+1}) \pmod{10}. \end{aligned}$$

Wenn sich die beiden am Zahlendreher beteiligten Ziffern um genau 5 unterscheiden, d.h., wenn $x_i - x_{i+1} \equiv 5 \pmod{10}$ gilt, dann hat die „falsche EAN“ $y$ wegen $2(x_i - x_{i+1}) \equiv 0 \pmod{10}$ die *gleiche* Prüfziffer wie die richtige EAN $x$. Also können Zahlendreher von dieser Bauart mit diesem System *nicht* erkannt werden. Wenn sich die beteiligten Ziffern nicht um genau 5 unterscheiden, wird der Dreher erkannt, weil $x_i - x_{i+1} \not\equiv 5 \pmod{10}$ und daher $2(x_i - x_{i+1}) \not\equiv 0 \pmod{10}$ gilt.

### Alternative Prüfzifferdefinition

Wir haben gesehen, dass die Prüfziffer bei EAN so gebildet wird, dass die meisten, aber eben nicht alle Zahlendreher erkannt werden können. Hätte man dies durch geschickte Definition der Prüfziffer nicht irgendwie erreichen können? Doch. Wir beschreiben jetzt ein Prüfziffernverfahren, mit dem neben einzelnen Tippfehlern auch alle Zahlendreher erkannt werden.

Wir definieren die „gewichtete Quersumme“

$$s(x_1, \ldots, x_{12}) = \left( \sum_{i=1}^{12} i \cdot x_i \right) \bmod 13$$

und setzen als Prüfziffer $x_{13} = s(x_1, \ldots, x_{12})$. Offensichtlich kann $x_{13}$ jetzt Werte zwischen 0 und 12 annehmen. Da wir aber trotzdem bei einzelnen Schriftzeichen bleiben wollen, schreiben wir für 10 ein $A$, für 11 ein $B$ und für 12 ein $C$.

Wir rechnen aus zwei Gründen modulo 13. Alle in obiger Summe auftretenden Zahlen, also die Ziffern $x_i$ und die Gewichte $i$, sind kleiner als 13. Zudem ist 13 eine Primzahl. Beides ist für die Erkennung von Tippfehlern wichtig, wie wir gleich sehen werden. Die Zahl 13 hat sich angeboten, da wir die Prüfziffer einer zwölfstelligen Zahlenfolge berechnen möchten. Wäre unsere Zahlenfolge länger bzw. kürzer, würden wir modulo einer entsprechend größeren bzw. kleineren Primzahl rechnen. Natürlich müssen wir dann auch den Zeichenvorrat für die Prüfziffern entsprechend anpassen.

### Erkennung von Tippfehlern

Jetzt überlegen wir, dass auch mit dieser Prüfziffer einzelne Tippfehler erkannt werden. Wie zuvor betrachten wir zwei Zahlenfolgen $x$ und $y$, die sich an genau einer Position $i$ unterscheiden: $y_i = x_i + t$ für ein ganzzahliges $t$ ungleich Null.

Wenn wir uns bei der Prüfziffer selbst vertan haben, heißt das: $x_j = y_j$ für alle $j$ von 1 bis 12 und $x_{13} \neq y_{13}$. Mit unserer Definition kann das aber natürlich erkannt werden: Wegen $s(x_1, \ldots, x_{12}) = s(y_1, \ldots, y_{12})$ gilt auch $x_{13} = y_{13}$.

Wenn wir uns bei genau der $i$-ten der zwölf ersten Ziffern vertan haben, ist zu zeigen, dass sich die Prüfziffer $y_{13}$ der „falschen EAN“ von der Prüfziffer $x_{13}$ der richtigen EAN unterscheiden muss. Dazu berechnen wir die korrekte Prüfziffer der Zahlenfolge $y_1 \ldots y_{12}$:

$$\begin{aligned} s(y_1, \ldots, y_{12}) &= (y_1 + 2 \cdot y_2 + \cdots + 12 \cdot y_{12}) \bmod 13 \\ &= (x_1 + 2 \cdot x_2 + \cdots + i \cdot (x_i + t) + \cdots + 12 \cdot x_{12}) \bmod 13 \\ &= (s(x_1, \ldots, x_{12}) + i \cdot t) \bmod 13 \\ &= (x_{13} + i \cdot t) \bmod 13 \end{aligned}$$

Die Zahl $(x_{13} + i \cdot t) \bmod 13$ kann nur dann gleich $x_{13}$ sein, wenn $i \cdot t \equiv 0 \pmod{13}$. Da weder $t$ noch $i$ gleich 0 sind, kann $i \cdot t$ ebenfalls nicht 0 sein. Bleibt zu zeigen, dass $i \cdot t$ kein Vielfaches von 13 sein kann. Das ist leicht: 13 ist eine Primzahl, d.h., alle Vielfachen davon müssen 13 als Primfaktor enthalten. Da sowohl $t$ als auch $i$ kleiner als 13 sind, ist das unmöglich. Wir haben also gezeigt, dass $i \cdot t \not\equiv 0 \pmod{13}$ ist und sich die Prüfziffern von $x$ und $y$ unterscheiden.

**Erkennung von Zahlendrehern**

Als nächstes zeigen wir, dass alle Zahlendreher erkannt werden können. Wir betrachten wieder zwei Folgen $x$ und $y$, bei denen zwei benachbarte Ziffern $x_i = y_{i+1}$ und $x_{i+1} = y_i$ vertauscht sind, die aber ansonsten identisch sind. Damit wir überhaupt von einem Zahlendreher sprechen können, nehmen wir an, dass $x_i$ und $x_{i+1}$ unterschiedlich sind. Wir berechnen erneut die Prüfziffer von $y_1 \dots y_{12}$:

$$\begin{aligned} s(y_1, \dots, y_{12}) &= (y_1 + \cdots + i \cdot y_i + (i+1) \cdot y_{i+1} + \cdots + 12 \cdot y_{12}) \bmod 13 \\ &= (x_1 + \cdots + i \cdot x_{i+1} + (i+1) \cdot x_i + \cdots + 12 \cdot x_{12}) \bmod 13 \\ &= (x_1 + \cdots + (i+1) \cdot x_{i+1} + i \cdot x_i + \cdots + 12 \cdot x_{12} \\ &\quad + x_i - x_{i+1}) \bmod 13 \\ &= (s(x_1, \dots, x_{12}) + x_i - x_{i+1}) \bmod 13 \\ &= (x_{13} + x_i - x_{i+1}) \bmod 13 \end{aligned}$$

Die Zahl $(x_{13} + x_i - x_{i+1}) \bmod 13$ kann nur dann gleich $x_{13}$ sein, wenn $x_i - x_{i+1} \equiv 0 \pmod{13}$ ist. Das ist aber unmöglich, da sowohl $x_i$ als auch $x_{i+1}$ ganze Zahlen zwischen 0 und 12 sind und zusätzlich $x_i \neq x_{i+1}$ gilt. Damit haben wir gezeigt, dass mit dieser Prüfziffer alle Zahlendreher erkannt werden können.

**Diskussion**

Warum hat man bei EAN nicht die eben beschriebene Definition gewählt? Diese Prüfsumme ist doch nicht schwerer zu bestimmen als die bei EAN. Der Hauptnachteil liegt wohl darin, dass diese Variante die drei zusätzlichen Ziffern $A$, $B$ und $C$ benötigt, was man durchaus als unpraktikabel ansehen kann. Zudem werden die EAN üblicherweise nicht von Menschen eingegeben, sondern maschinell eingelesen, z.B. von Laserscannern. Dabei besteht die Gefahr von Zahlendrehern eher nicht. Da EAN ja auch viele Dreher erkennen kann, wurde das System als ausreichend sicher und gleichzeitig praktikabel angesehen.

Die Prüfzifferdefinition, die wir hier vorgestellt haben, wurde übrigens in leicht abgewandelter Form beim mittlerweile veralteten ISBN-10-Standard zur Buchkennzeichnung verwendet. ISBN-10-Nummern haben 11 Stellen, 10 davon sind Ziffern zwischen 0 und 9, die elfte ist die Prüfziffer. Diese wird wie oben als gewichtete Summe der ersten 10 Ziffern bestimmt, nur dass modulo 11 gerechnet wird und eine möglicherweise entstehende Prüfziffer 10 mit $X$ bezeichnet wird.

Nachdem ISBN-10 im Gegensatz zu EAN alle einzelnen Zahlendreher erkennen kann, ist der neue EAN-Standard aus technischer Sicht eigentlich ein Rückschritt. Dies wird jedoch für den Vorteil der internationalen Standardisierung in Kauf genommen.

Dass solche Standards durchaus wichtig sind, veranschaulicht das folgende drastische Beispiel: Am 23.9.1999 stürzte die NASA Raumsonde „Mars Climate Orbiter“ ab, weil ihre tatsächliche Umlaufbahn niedriger war als berechnet. Wie konnte das passieren? Ein Teil der Software rechnete in Inch, ein anderer in Meter!

## Ausblick: Fehlererkennende/-korrigierende Codes

EAN ist ein Beispiel eines fehlererkennenden Codes. Allgemein dienen diese dazu, wie der Name schon sagt, um zu erkennen, dass bei der Übertragung von Daten Fehler entstanden sind. Dazu wird der eigentlichen Information (Nutzdaten), die übertragen werden soll, zusätzliche, eigentlich überflüssige Information (Redundanz) hinzugefügt. Die Redundanz ist natürlich nicht beliebig, sondern enthält Informationen über die Nutzdaten in geeigneter Form. Diese dient dem Empfänger zur Kontrolle, ob Fehler aufgetreten sind, und gestattet eventuell sogar die Bestimmung von Fehlerpositionen.

Im Beispiel der EAN sind die Nutzdaten die ersten zwölf Ziffern und die Redundanz die Prüfziffer, mit der einzelne Tippfehler und manche Zahlendreher erkannt werden können.

Fehler*korrigierende* Codes arbeiten nach dem gleichen Prinzip wie fehler*erkennende*, aber sie gestatten es zusätzlich, einige Fehler zu beheben. Die Ergänzung der zu übertragenden Daten allein um eine Prüfsumme genügt nicht, um Fehlerkorrektur zu ermöglichen. Dazu ist eine geschicktere Codierung nötig. Wir können uns die Funktionsweise an folgendem Beispiel verdeutlichen.

Angenommen wir wollen einen binären Text, d.h. eine Folge von Nullen und Einsen, übertragen, also z.B. die Zeichenkette 0110. Anstatt diese Nutzdaten zu übertragen, codieren wir wie folgt:

| **Nutzdatum** | **Codierung** |
|---|---|
| 0 | 000 |
| 1 | 111 |

Aus den Nutzdaten 0110 wird also 000111111000. Der Empfänger bildet dann gemäss folgender Tabelle aus dem (eventuell gestörten) empfangenen Code wieder Nutzdaten. Dieser Schritt heißt Decodierung.

| **Empfangener Code** | **Nutzdatum** |
|---|---|
| 000, 001, 010, 100 | 0 |
| 111, 110, 101, 011 | 1 |

Mit dieser Decodiertabelle wird aus einem Dreierblock ein richtiges Nutzdatum gebildet, falls in dem Block nur höchstens ein Bit falsch übertragen

wurde. Angenommen, bei der Übertragung einer codierten Null 000 wird ein einzelnes Bit gestört und beispielsweise 001 anstelle von 000 empfangen. Gemäß obiger Decodiertabelle übersetzen wir 001 trotzdem zu 0.

Mit diesem Verfahren erreichen wir also einen gewissen Schutz vor Übertragungsfehlern, aber erkaufen dies durch erhöhten Übertragungsaufwand; schließlich übertragen wir die dreifache Anzahl Bits. Eine etwas wirtschaftlichere Codierung erzielen wir, wenn wir die Bits nicht einzeln codieren, sondern mehrere geschickt zu Gruppen zusammenfassen:

| **Nutzdaten** | **Codierung** |
|---|---|
| 00 | 00000 |
| 01 | 00111 |
| 10 | 11100 |
| 11 | 11011 |

Folgende Decodiertabelle erfasst alle Übertragungsfehler einzelner Bits in Fünferblöcken.

| **Empfangener Code** | **Nutzdaten** |
|---|---|
| 00000, 10000, 01000, 00100, 00010, 00001 | 00 |
| 00111, 10111, 01111, 00011, 00101, 00110 | 01 |
| 11100, 01100, 10100, 11000, 11110, 11101 | 10 |
| 11011, 01011, 10011, 11111, 11001, 11010 | 11 |

Auch diese Codierung hat die Eigenschaft, dass einzelne Bitfehler korrigiert werden können, aber das Verhältnis der Nutzdatenlänge zur Codelänge ist $\frac{2}{5}$ anstelle $\frac{1}{3}$ wie bei der vorangegangenen Lösung. Dieses Verfahren ist damit (etwas) effizienter.

Den acht in der Decodiertabelle fehlenden Fünferblöcken können leider keine sinnvollen Nutzdaten zugeordnet werden. Beispielsweise können aus dem gestörten Code 10101 die Codes 00111 und 11100 jeweils durch Änderung von zwei Bits gebildet werden. Diese Codes gehören jedoch zu jeweils unterschiedlichen Nutzdaten, und es ist somit unklar, was ursprünglich codiert wurde. Wenn also ein Code empfangen wird, für den die Tabelle keine Nutzdaten vorsieht, sollte ein entsprechendes Fehlerzeichen ausgegeben werden.

Fehlerkorrigierende Codes begegnen uns übrigens auch im täglichen Leben. Beispielsweise wird ein solches Verfahren benutzt, um Fehler, die beim Lesen einer CD auftreten, zu korrigieren und so den Hörgenuss zu steigern. Dabei werden bei der Codierung, im Prinzip wie oben, aus dem laufenden Datenstrom jeweils 192 Bits zusammengefasst und mit 32 Fehlerkorrekturbits ausgestattet. Dieser Block wird noch geschickt weiter codiert. Ein Kratzer auf der CD kann ohne weiteres die Bits eines halben Blocks beschädigen, aber durch die Codierung kann die verkratzte CD trotzdem fehlerfrei wiedergegeben werden.

## Zum Weiterlesen

1. T.H. Cormen, C. E. Leiserson, R. L. Rivest und C. Stein: *Introduction to Algorithms*. MIT Press, 2001.

   Eine umfassende, grundlegende und gut geschriebene Einführung in die Algorithmik, deren EAN uns bei diesem Beitrag als Beispiel diente.

2. A. Steger: *Diskrete Strukturen – Band 1*. Springer Verlag, 2002.

   Eine Einführung in die diskrete Mathematik, in der unter anderem die modulare Arithmetik ausführlicher behandelt wird als es in diesem Kapitel möglich ist.

3. W.C. Huffman und V. Pless: *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

   Ein weiterführendes Buch zum Thema Fehlererkennende/-korrigierende Codes. Die benötigten Grundlagen aus der Algebra (z. B. endliche Körper, euklidischer Algorithmus) bzw. aus der elementaren Zahlentheorie werden eingeführt.

4. Aus Wikipedia:
   `http://de.wikipedia.org/wiki/European_Article_Number`
   `http://de.wikipedia.org/wiki/Fehlerkorrekturverfahren`

   Im zweiten Wikipedia-Beitrag ist auch die Datencodierung bei CDs beschrieben.

# Teil III

# Planen, strategisches Handeln und Computersimulationen

# Übersicht

Helmut Alt und Rüdiger Reischuk

FU Berlin
Universität zu Lübeck

*Strategisches Denken und Planen* werden landläufig als typisch menschliche Fähigkeiten angesehen. Spätestens jedoch seit es Computerprogramme gibt, die Großmeister im Schach schlagen können, sieht man auch, dass manche dieser Fähigkeiten erfolgreich von einer Maschine bewerkstelligt werden können. Vielen anspruchsvollen Spielprogrammen liegt der so genannte *Alpha-Beta-Algorithmus* zugrunde, der sehr geschickt und effizient eine große Anzahl von möglichen Fortsetzungen eines Spiels unter Berücksichtigung aller möglichen Reaktionen des Gegners bewertet und es uns so erlaubt, einen „guten" Spielzug auszuwählen. Dieser Algorithmus und die Idee dahinter werden in einem zentralen Beitrag dieses Kapitels vorgestellt.

Auf der anderen Seite gibt es aber auch Spiele, die man mit einer ganz einfachen Strategie gewinnen kann – diese muss man nur kennen. In einem weiteren Beitrag wird dies eindrucksvoll am Streichholzspiel *Nim* demonstriert.

Bei vielen Spielen ist es wichtig, dass der Gegner unsere Züge nicht vorhersehen kann. Eine einfache Strategie – in der informatischen Fachsprache *deterministisch* genannt – läßt sich jedoch vorherberechnen. Dies kann man verhindern, wenn man Zufallsentscheidungen einbaut – ohne diese wären viele Knobelspiele wie etwa *Stein-Schere-Papier* langweilig. Auch viele Algorithmen kann man auf diese Weise verbessern oder beschleunigen – man spricht dann von *probabilistischen* oder *randomisierten* Algorithmen. Nun müssen wir uns aber die Frage stellen, wie kann ein Computer, von dem man eigentlich hundertprozentige Exaktheit erwartet, würfeln oder eine Münze werfen? Der Beitrag über Zufallszahlen liefert hierzu eine Antwort.

Nicht nur beim Spielen, sondern auch bei ernsthaften Problemen des täglichen Lebens ist ein strategisches und algorithmisches Vorgehen sinnvoll. Soll z. B. eine Nachricht an eine größere Gruppe von Menschen durch Telefonanrufe oder an eine größere Menge von Computern über ein elektronisches Netz verbreitet werden, empfiehlt sich eine gut überlegte Planung, um dies schnell und sicher zu bewerkstelligen. Dies wird im ersten Beitrag über schnelles *Broadcasting* gezeigt. In einem weiteren Beitrag werden wir sehen, wie man

geschickt mit wenig Aufwand den Gewinner einer Wahl bestimmt, wenn man die Problemstellung vorher eingehend analysiert.

Manche Aufgabenstellungen erfordern sogar eine sorgfältige längerfristige *Planung,* wie sich an dem Problem zeigt, einen Spielplan für eine Bundesligasaison festzulegen, der eine Reihe von Randbedingungen berücksichtigt. Unser letzter Beitrag wird sich mit dem geschickten Aufteilen eines Kuchens an mehrere Personen befassen, so dass sich niemand benachteiligt fühlt. Probleme dieser Art treten aber nicht nur bei der Verteilung des Weihnachtsstollens in einer kinderreichen Familie auf, sondern auch in größeren Maßstäben in einer Volkswirtschaft bei der Verteilung öffentlicher Güter und knapper Ressourcen oder bei Auktionen zur Preisfindung von Wirtschaftsgütern. Gute algorithmische Lösungsverfahren werden daher auch für Wirtschaft und Politik immer interessanter und stellen das Thema einer in den letzten Jahren stark an Bedeutung gewonnenen Teildisziplin dar, der *algorithmischen Spieltheorie.*

Zwei Beiträge dieses Kapitels befassen sich mit *Simulationen*, d. h. dem Nachbilden von in der Natur auftretenden Vorgängen *in* einem Computer. Zum einen handelt es sich um ein physikalisches Problem. Wir werden sehen, wie die Wärmeverteilung in einem Metallstab oder einer Platte, die an bestimmten Stellen erhitzt werden, mit der so genannten *Gauß-Seidel-Iteration* berechnet werden kann. Beim zweiten Thema, das der Biologie entstammt, wird gezeigt, wie man aus der Erbinformation (DNA) zweier Lebewesen bestimmen kann, wie nahe diese miteinander verwandt sind und durch welche Mutationen – das sind minimale Änderungen des Erbguts – sie auseinander oder aus einem gemeinsamen Vorfahren hervorgegangen sind.

Auf den bedeutenden Mathematiker Leonhard Euler geht das Problem zurück, ob man alle sieben Brücken der Stadt Königsberg bei einem Spaziergang genau einmal überqueren und dann zum Ausgangspunkt zurückkommen kann. Diese spielerisch anmutende Frage (die Antwort ist übrigens „nein“!) hat wichtige Anwendungen, etwa bei der Routenplanung und wird im Beitrag *Eulerkreise* behandelt. Bei der Fahrzeug-Navigation sind wir es inzwischen gewohnt, durch eine freundliche Stimme auf Richtungsänderungen oder noch zurückzulegende Kilometer hingewiesen zu werden. Eine natürliche Sprachausgabe war für Computer lange Zeit ein ungelöstes Problem. Wir werden in diesem Kapitel sehen, dass allein die Aussprache langer Zahlen schon einigen algorithmischen Aufwand erfordert.

Schließlich betrachten wir noch ein Problem aus der Computergrafik: Zeichne einen *möglichst runden* Kreis auf einem Bildschirm, der physikalisch, wie jeder weiß, aus einzelnen gitterförmig angeordneten Bildpunkten oder Quadraten (so genannten *Pixels*) besteht. Eine schräge oder gekrümmte Linie wie mit Papier und Bleistift können wir also genau genommen gar nicht erzeugen. Auch hier zeigt sich, dass eine genaue Analyse der Problemstellung zu überraschend einfachen und schnellen Lösungsalgorithmen führt.

# 22

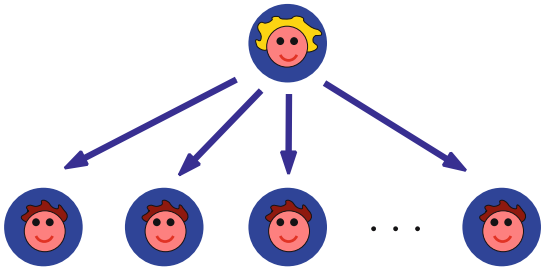
# Broadcasting: Wie verbreite ich schnell Informationen?

Christian Scheideler

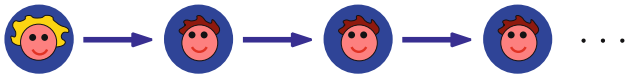
Technische Universität München

Im Mittelalter gab es noch keine Massenmedien wie Fernsehen oder Radio. Weil die meisten Leute weder schreiben noch lesen konnten, wurden Informationen überwiegend von Mund zu Mund weitergegeben, und da die Reisegeschwindigkeit der Menschen zu dieser Zeit recht begrenzt war, konnten sich Informationen höchstens mit der Geschwindigkeit von Pferden ausbreiten (obwohl spätestens seit den Kreuzzügen auch Brieftauben in unseren Breiten eingesetzt worden sind). Heutzutage ist es Dank des Telefons und neuer Medien wie E-Mails auch als Privatperson sehr einfach, Informationen schnell zu verbreiten. Betrachten wir dazu ein konkretes Beispiel.

Steffi hat soeben die Aufgabe bekommen, eine Jahrgangsstufenfete zu organisieren, und das, wo doch gerade die Ferien angefangen haben! Jetzt muss sie versuchen, sämtliche Mitschüler über Handy oder E-Mails zu erreichen. Die E-Mail-Adressen der anderen Schüler weiß Steffi zwar nicht, aber es gibt zum Glück eine Liste sämtlicher 121 Schüler in ihrem Jahrgang mit deren Telefonnummern, die jeder ihrer Mitschüler besitzt (oder hoffentlich noch besitzt!). Nun könnte Steffi natürlich 120 Telefonate führen, wozu sie allerdings weder Zeit noch Lust hat. Daher überlegt sie sich, ob es nicht auch andere Möglichkeiten gibt, möglichst schnell und einfach alle 120 Mitschüler zu erreichen.



**Strategie 1:** Alle direkt anrufen



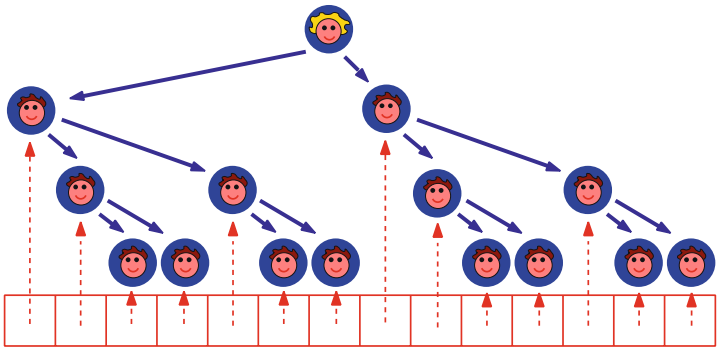
**Strategie 2:** Stille Post

Die erste Strategie, die ihr dabei einfällt, ist, einfach das Stille-Post-Spiel durchzuziehen. Das heißt, sie ruft einfach den ersten auf der Liste an und bittet ihn, den nächsten auf der Liste anzurufen, der dann wieder den nächsten auf der Liste anrufen soll, usw., bis das Ende der Liste erreicht ist.

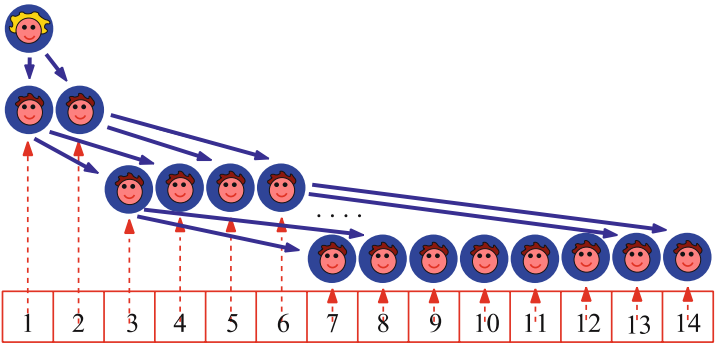
Der Vorteil dieses Vorgehens ist, dass jeder Schüler nur einen Anruf tätigen muss. Da diese Anrufe allerdings hintereinander durchgeführt werden müssen, kann eine erhebliche Zeitspanne verstreichen, bis alle Mitschüler informiert sind. In der Tat, wenn auch nur 10% den nächsten auf der Liste nicht am gleichen Tag anrufen oder erreichen können, dauert es mindestens 12 Tage, bis alle Schüler informiert sind. Schlimmer noch: Wenn einer gar nicht reagiert, bricht das gesamte System zusammen! Steffi überlegt sich also eine andere Strategie.

Da sie sich für Informatik interessiert und sich daher regelmäßig den Algorithmus der Woche anschaut, erinnert sie sich an die Sortiermethoden, die in Kap. 3 vorgestellt worden sind. Dort hat ein Meister zwei Gehilfen benutzt, um das Sortierproblem in zwei Teilprobleme aufzuteilen, und jeder dieser Gehilfen hat wiederum zwei Gehilfen benutzt, um diese Teilprobleme weiter zu unterteilen usw. Solch eine Strategie mit Gehilfen sollte sich doch auch auf Anrufe anwenden lassen! Dazu teilt Steffi die Telefonliste in zwei gleichgroße Teillisten auf. Wenn sie sich nun vornimmt, den ersten auf jeder Teilliste anzurufen, und diesen bittet, die Teilliste weiter in zwei gleichgroße Teillisten aufzuteilen und die ersten dieser Teillisten anzurufen usw., dann können ihre Mitschüler viel schneller erreicht werden.

In der Tat, Steffi rechnet sich aus, dass schon nach 7 Anrufrunden alle 120 Mitschüler informiert sind. Das ist schon wesentlich besser als 120 Anrufrun-



**Strategie 3:** Aufteilung in Teillisten




**Strategie 4:** Jeder auf Listenplatz $i$ ruft die Plätze $2i + 1$ und $2i + 2$ an

den! Allerdings klingt das mit der Teillisteneinteilung etwas zu technisch. Ob das alle Mitschüler wirklich so mitmachen??? Sie überlegt sich also eine andere Strategie:

Angenommen, sie ruft die ersten beiden Mitschüler auf der Liste an, den Andi und den Berthold, und bittet Andi, die Mitschüler auf den Plätzen 3 und 4 anzurufen, und Berthold, die Mitschüler auf den Plätzen 5 und 6 anzurufen. Außerdem sollen sie die Regel weitergeben, dass jeder auf Listenplatz $i$ die Mitschüler auf den Plätzen $2i + 1$ und $2i + 2$ anruft. Dann pflanzt sich die Information mit derselben Geschwindigkeit wie in Strategie 3 fort, aber die Anrufregel klingt nicht mehr so technisch.

Trotzdem ist Steffi noch nicht recht zufrieden mit ihrer Anrufstrategie. Was, wenn sich einer der Mitschüler verzählen sollte und ein falsches Paar Schüler auf der Liste anruft? Außerdem kann es natürlich ein paar Schlafmützen geben, die einfach vergessen, die nächsten auf der Liste anzurufen. Dann würden einige Mitschüler nicht informiert werden, die dann natürlich sauer auf Steffi wären!

Steffi überlegt sich also die Strategie, dass jeder auf Listenplatz $i$ die vier Mitschüler auf den Plätzen $2i+1$ bis $2i+4$ anruft. Dann wird im Idealfall jeder Schüler (außer den ersten vier, die allerdings von Steffi angerufen werden) von genau zwei Mitschülern angerufen. Solange also für jeden auf der Liste höchstens einer von den beiden Mitschülern, die ihn anrufen sollen, nicht erreichbar ist (oder einen Fehler macht oder den Anruf schlichtweg verpennt), werden alle Schüler, die erreichbar sind, informiert. Das kann man sich anschaulich so überlegen: Wenn man für jeden Schüler einen Vorgänger wählt, der ordentlich arbeitet, dann bekommt man für jeden eine ununterbrochene funktionierende Kette bis zum Anfang (siehe auch Abb. 22.1).

Steffi überlegt sich, dass diese Strategie vielleicht noch ein wenig robuster ausgelegt werden sollte, damit sie wirklich sicher sein kann, dass auch alle erreicht werden. Sie überlegt sich dazu Folgendes: Wenn jeder Schüler auf Listenplatz $i$ die Mitschüler auf den Plätzen $2i + 1$ bis $2i + 2r$ für eine feste Zahl $r$ anruft, dann wird im Idealfall jeder Schüler (außer den ersten $2r$, die

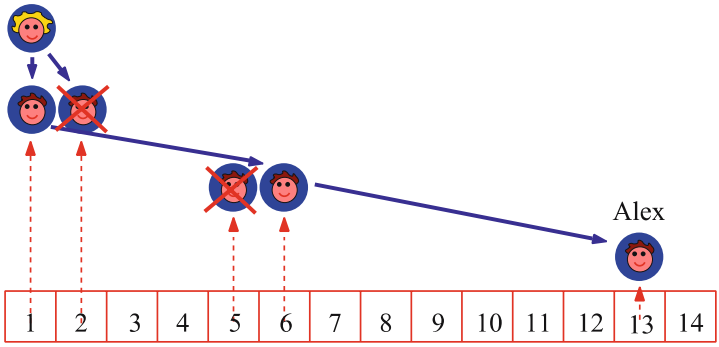



**Abb. 22.1.** Kette zuverlässiger Schüler für Alex, wenn die Plätze $2i+1$ bis $2i+4$ angerufen werden

allerdings von Steffi angerufen werden) von $r$ anderen Schülern angerufen (siehe auch Abb. 22.2). Dann können beliebige $r-1$ der $r$ Schüler Probleme machen, und alle erreichbaren Mitschüler werden immer noch erreicht!

Hier ist ein guter Zeitpunkt gekommen, mal ein paar Experimente durchzuführen. Für eine gegebene Anzahl $x$ (z.B. 10) problematischer Schüler, die zufällig über die Listenplätze verteilt sind, wollen wir herausfinden, wie groß das $r$ gewählt werden muss, damit die Wahrscheinlichkeit, dass nicht alle erreichbaren Schüler erreicht werden, höchstens $p$ (z.B. 10%) ist. Dazu kann man den im Folgenden abgebildeten Algorithmus als Grundlage verwenden. Dieser Algorithmus spielt nicht wahrheitsgemäß die Informationsverbreitung durch (die ja parallel abläuft), sondern errechnet lediglich, ob unter Anwendung des vorgegebenen Kommunikationsmusters am Ende alle erreichbaren Schüler informiert wurden. In der Rechnung reicht es aus, die for-Schleife in Zeile 6 nur bis $N/2$ laufen zu lassen, da Schüler mit größeren Nummern keinen weiteren Schüler mehr anrufen. Der Algorithmus basiert auf einem Feld $A$, das wie folgt definiert ist.

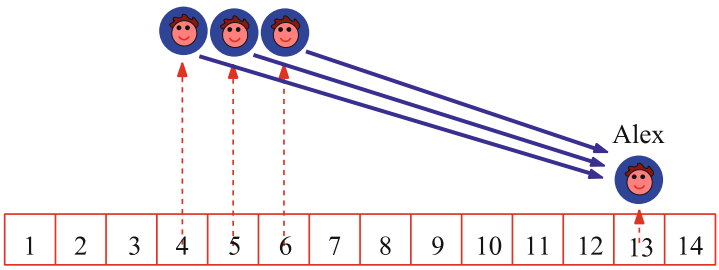



**Abb. 22.2.** Die $r$ Schüler, die Alex im Idealfall anrufen, für $r=3$

- $A$: Array $[1..N]$ of Integer: $A[i]$ zählt für einen zuverlässigen Schüler auf Platz $i$ die Anzahl der Anrufe, die dieser Schüler bekommt. $N$ ist die Anzahl der Schüler.
- Für alle zuverlässigen Schüler wird $A[i]$ anfangs auf 0 gesetzt.
- Für alle unzuverlässigen Schüler wird $A[i]$ anfangs auf $-r-1$ gesetzt (damit sie auch bei $r$ Anrufen nicht auf einen Wert $\geq 0$ kommen).

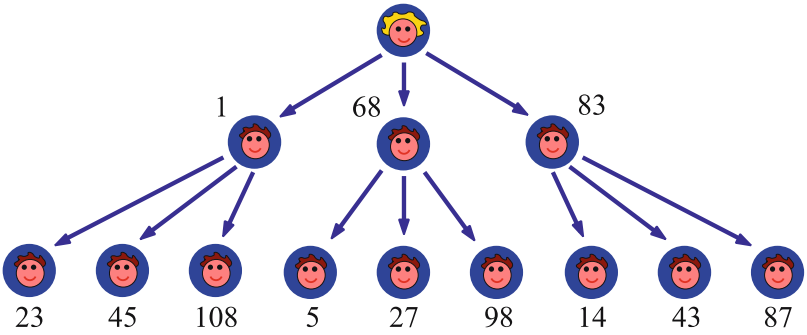
Algorithmus für $r$-fache Informationsverbreitung.

```
 1  procedure BROADCAST (r)
 2  begin
 3      for j := 1 to 2 * r do      // Steffi ruft Schüler 1 bis 2r an
 4          A[j] := A[j] + 1
 5      endfor
 6      for i := 1 to N/2 do      // Schüler i ruft 2i + 1 bis 2i + 2r an
 7          if A[i] > 0 then    // sofern Anruf erhalten
 8              for j := 2 * i + 1 to 2 * i + 2 * r do
 9                  if j ≤ N then A[j] := A[j] + 1
10                  endif
11              endfor
12          endif
13      endfor
        // Hat's geklappt?
14      for i := 1 to N do
15          if A[i] = 0 then Ausgabe „leider nicht alle erreicht“, stop
16          endif
17      endfor
18      Ausgabe „alle erreicht“
19  end
```

Bei dieser ganzen Überlegerei hat Steffi so langsam richtig Spaß daran bekommen, sich Strategien auszudenken. Sie stellt sich jetzt erschwerend die Situation vor, dass zwar jeder Schüler die Telefonnummern aller Mitschüler der Jahrgangstufe kennt, es aber keine einheitlich geordnete Liste gibt, so dass die Strategien mit den Listenplätzen oben nicht anwendbar sind. Gibt es dann immer noch ein schnelles und robustes Verfahren, um alle Schüler zu erreichen, selbst wenn, sagen wir, ein beliebiges Viertel der Schüler unzuverlässig ist?

Nach einigem Nachdenken kommt Steffi auf die Idee, dass sowohl Steffi als auch jeder zum ersten Mal angerufene Schüler einfach $r$ zufällig gewählte Mitschüler anruft (siehe Strategie 5).

Wenn Steffi mit dieser Strategie anfängt, dann informiert sie garantiert $r$ noch nicht angerufene Mitschüler, falls diese erreichbar sind. Im Idealfall sind alle zuverlässig und rufen dann jeweils $r$ weitere zufällig ausgewählte Schüler an, so dass im Idealfall $r^2$ noch nicht angerufene Mitschüler erreicht werden. In Wirklichkeit kann es dabei natürlich zu Überschneidungen kommen, d.h., derselbe Schüler wird mehrmals angerufen. Da er aber nur beim ersten Anruf aktiv wird (sonst würde die Anruferei nie terminieren!), schadet das der Aus-




**Strategie 5:** Jeder Schüler, inklusive Steffi, ruft $r$ zufällig gewählte Mitschüler an, für $r = 3$

breitung. Außerdem kann es passieren, dass unzuverlässige Schüler angerufen werden, was auch der Ausbreitung der Nachricht schadet, d. h., die Zahl der angerufenen Schüler erhöht sich nicht wesentlich. Trotzdem kann man aber durch Experimente sehen, dass sich Steffis Informationen schon bei relativ kleinem $r$ mit hoher Zuverlässigkeit schnell ausbreiten. Dafür können wir den unten angegebenen Algorithmus verwenden. Er basiert auf den Feldern $A$ und $C$, die wie folgt definiert sind:

- $A$: Array $[1..N]$ of Integer: Array, in dem anfangs $A[i] = -1$ ist, wenn der Schüler i unzuverlässig ist, und $A[i] = 0$ ist, wenn der Schüler $i$ zuverlässig ist. $N$ ist die Anzahl der Schüler.
- Wenn ein zuverlässiger Schüler auf Platz $i$ zum ersten Mal angerufen wird, wird $A[i]$ auf 1 gesetzt, und sobald er alle seine Anrufe erledigt hat, wird $A[i]$ auf 2 gesetzt.
- $C$: Array $[0..N][1..r]$ of Integer: Array, in dem $C[i][j]$ den Listenplatz des $j$-ten Schülers angibt, den der Schüler auf Platz $i$ anruft (Steffi zählt hier als Schüler 0). $C$ wird anfangs zufällig gewählt.

Natürlich kann man sich noch jede Menge anderer Informationsverbreitungsstrategien ausdenken, und jeder sei an dieser Stelle ermuntert, diese mal zu testen. Welche Strategie hättet ihr denn an Stelle von Steffi gewählt?

Algorithmus für zufällige $r$-fache Informationsverbreitung.

```
 1  procedure RandomBroadcast (r)
 2  begin
 3     for j := 1 to r do      // Anrufe von Steffi
 4        if A[C[0][j]] = 0 then A[C[0][j]] := 1
 5        endif
 6     endfor
 7     weiter := 1    // Indikator für neu angerufene Schüler
 8     while weiter = 1 do
 9        weiter := 0
10        for i := 1 to N do      // suche nach neu angerufenen Schülern
11           if A[i] = 1 then
12              weiter := 1; A[i] := 2
13              for j := 1 to r do
14                 if A[C[i][j]] = 0 then A[C[i][j]] := 1
15                 endif
16              endfor
17           endif
18        endfor
19     endwhile
       // Hat's geklappt?
20     for i := 1 to N do
21        if A[i] = 0 then Ausgabe „leider nicht alle erreicht“, stop
22        endif
23     endfor
24     Ausgabe „alle erreicht“
25  end
```

## Zum Weiterlesen


1. `http://de.wikipedia.org/wiki/Broadcast`

   Der Wikipedia-Artikel gibt eine Einführung in den Begriff der Informationsverbreitung (im Fachjargon auch „Broadcasting“ genannt) und in gebräuchliche Broadcasting-Protokolle.

2. C. Diot, W. Dabbous und J. Crowcroft: *Multipoint Communication: A Survey of Protocols, Functions and Mechanisms*. IEEE Journal on Selected Areas in Communications 15(3), Seite 277–290, 1997.

   Dieser Artikel empfiehlt sich für eine Einführung in die Fachliteratur bezüglich Broadcasting.

3. R. Karp, S. Shenker, C. Schindelhauer und B. Vöcking: *Randomized Rumor Spreading*. In: *IEEE Symposium on Foundations of Computer Science (FOCS)*, Seite 565–574, 2000.

Dieser Artikel enthält fortgeschrittene Broadcasting-Methoden, die noch effektiver, aber auch deutlich komplexer als die hier vorgestellten sind. Es empfiehlt sich für alle, die wirklich tief in die Materie einsteigen wollen und sich von mathematischen Formeln nicht abschrecken lassen.

# 23

# Zahlen auf Deutsch aussprechen

Lothar Schmitz

Universität der Bundeswehr München

In diesem Beitrag geht um das Aussprechen von Zahlen, wie das z. B. ein Auto-Navigationssystem für jede benötigte Entfernungsangabe fertigbringt. Die freundliche Stimme unseres Auto-Navis sagt dabei nicht wie ein alter Blech-Roboter:


Nein, sie spricht die Zahl genauso aus wie wir:


Ein „weltraumtaugliches“ Navigationssystem müsste noch viel größere Zahlen aussprechen können, z. B. die Zahl 12 345 678 987 654 321 als

zwölf Billiarden
dreihundertfünfundvierzig Billionen
sechshundertachtundsiebzig Milliarden
neunhundertsiebenundachtzig Millionen
sechshundertvierundfünfzigtausend
dreihunderteinundzwanzig

So etwas kann eigentlich jeder von uns. Aber wie sieht ein entsprechendes Computer-Programm aus? Da steckt der Teufel wie so oft im Detail! Wir präzisieren zuerst die Aufgabenstellung.

**Aufgabe:** Gegeben sei eine Zahl $x$ mit $1 \leq x \leq 10^{24}$. Erzeuge den deutschen Zahltext zu $x$. Wir gehen davon aus, dass sich Zahlen dieser Größe in der verwendeten Programmiersprache darstellen lassen und dass Grundrechenarten wie Addition, Subtraktion, Multiplikation und Division (ganzzahlig mit Rest) zur Verfügung stehen.

## Schrittweise Entwicklung eines Algorithmus

Das einfachste Verfahren wäre, zu jeder Zahl die zugehörige deutsche Aussprache zu speichern und bei Bedarf abzurufen. Wegen des enormen Speicherbedarfs ist dieses Vorgehen nicht praktikabel. Auch wir als Menschen wären damit überfordert, uns derart viel zu merken. Stattdessen erzeugen wir die deutschen Zahltexte jedesmal neu, wenn wir sie aussprechen. Offenbar braucht man sich dafür weniger zu merken.

Versuchen wir also, so genau wie möglich das Verfahren zu beschreiben, das wir selber verwenden! Wie funktioniert das?

Zuerst teilt man die gegebene Zahl von rechts (also beim kleinsten Gewicht) beginnend in Gruppen zu je drei Ziffern auf. Für das Beispiel oben ergibt sich:

Einer: 321
Tausender: 654
Millionen: 987
Milliarden: 678
Billionen: 345
Billiarden: 12

Wir beobachten: Die Zifferngruppen beschreiben jeweils Zahlen zwischen 0 und 999. Die höchstgewichtete Zifferngruppe kann auch weniger als drei Ziffern enthalten (mindestens aber eine).

Danach spricht man die Zifferngruppen in umgekehrter Reihenfolge aus. Für jede Zifferngruppe muss man zuerst den Zahltext einer Zahl zwischen 0 und 999 erzeugen und danach einen „Anhang“, der das Gewicht der Zifferngruppe bestimmt: „Billiarden“, „Billionen“, „Milliarden“ usw., wobei der Anhang „Einer“ einfach weggelassen wird.

Beim Erzeugen der Zahltexte zwischen 0 und 999 müssen wir darauf achten, dass die Ziffern im Deutschen in der Reihenfolge Hunderter-Einer-Zehner ausgesprochen werden und dass Hunderter, Einer und Zehner jeweils vorkommen oder fehlen können. Von der Regel abweichende Sonderfälle sind „elf“ und „zwölf“, wo Zehner und Einer zu einem Wort zusammengefasst sind. Und auch die Beugungen im Deutschen erforden Aufmerksamkeit. Wir sagen ja:

„ei**n**tausend“ für 1 000,
„ein**e** Million“ für 1 000 000,
„neunhunderttein**s** Millionen“ für 901 000 000.

Aber nun eins nach dem anderen!

### Aufteilen in Dreiergruppen ...

Um eine Zahl von rechts her in Dreierzifferngruppen zu zerlegen, teilt man sie immer wieder durch 1000: Der Teilungsrest ergibt jeweils die nächste Drei-

ergruppe. Folgendes Programmstück berechnet „nebenher“ in der Variablen $i$ die Anzahl der Dreiergruppen der gegebenen Zahl.

SCHRITT 1: Aufteilung der Zahl in Dreiergruppen. Die Dreiergruppen stehen danach im Array *gruppe*, und zwar die Einer in *gruppe*[0], die Tausender in *gruppe*[1], die Millionen in *gruppe*[2], die Milliarden in *gruppe*[3] ... Die Variable $i$ enthält stets die Anzahl der bisher gefundenen Dreiergruppen.

```
1  i := 0      { noch keine Gruppe gefunden }
2  while zahl > 0 do
3      gruppe[i] := zahl mod 1000      { Rest zur ... }
4      zahl := zahl/1000      { Division durch 1000 }
5      i := i + 1      { eine weitere Gruppe gefunden }
6  endwhile
```

### ... und „Aussprechen“ der Zahl

Wenn man die komplizierteren Teile des Verfahrens in Hilfsfunktionen auslagert, dann ist die Erzeugung des Zahltextes nicht schwer. Im folgenden Programmstück wird zum Erzeugen von Zahltexten zwischen 0 und 999 die Funktion SPRICHDREIER aufgerufen, zur Erzeugung des passenden Anhangs die Funktion SPRICHANHANG.

SCHRITT 2: Erzeugung des deutschen Zahltextes. Da im Array *gruppe* der kleinste Index 0 ist und da $i$ nach SCHRITT 1 die Anzahl der gefundenen Dreiergruppen enthält, ergibt sich der Index der höchstwertigen Gruppe zu $i - 1$. In der Variablen *txt* wird der Zahltext aufgesammelt. Der Operator „&“ fügt Texte aneinander.

```
1  text := „      { Text ist anfangs leer }
2  i := i − 1      { Index der höchstwertigen Gruppe }
3  while i ≥ 0 do
4      text := text & sprichDreier(gruppe[i])      { Gruppentext ... }
5      text := text & sprichAnhang(i)      { ... und Anhang anfügen }
6      i := i − 1      { zur nächsten Gruppe }
7  endwhile
```

Führt man nacheinander SCHRITT 1 und SCHRITT 2 aus, dann enthält *text* wunschgemäß den Zahltext, wie man ihn im Deutschen spricht!

### Die Funktion sprichDreier

Der Textanteil einer Dreiergruppe ist leer, wenn die Dreiergruppe den Wert 0 hat. Das zeigt ein Beispiel: 1 000 111 wird gesprochen als „eine Million einhundertelf“. Ansonsten zerlegt man die Dreiergruppe in ihre Bestandteile: die Anzahl der Hunderter als Ziffer $h$, die Anzahl der Zehner als Ziffer $z$ und die Anzahl der Einer als Ziffer $e$.

Spätestens an dieser Stelle muss man den Text zu jeder Ziffer kennen, um ihn ausgeben zu können. Da einige der Zahlen kleiner als 20 unregelmäßig gebildet werden – z.B. „siebzehn“ statt „siebenzehn“ –, merken wir uns für jede dieser Zahlen den Text in einem Array *kleiner*20. Entsprechend verfahren wir mit den Zehnern. Das ergibt folgende Vereinbarungen:

Vereinbarung von Arrays mit den Texten kleiner Zahlen (zwischen 1 und 19) und der Zehner. Für $i$ zwischen 1 und 19 steht in $kleiner20[i]$ der Text zur Zahl $i$. Für $j$ zwischen 2 und 9 ist $zehner[j]$ der Text zu $j \cdot 10$.

```
1  kleiner20 : array [1..19] of String :=
2      [ „ein“, „zwei“, „drei“, „vier“, ... , „achtzehn“, „neunzehn“ ]
3  zehner : array [2..9] of String :=
4      [ „zwanzig“, „dreißig“, „vierzig“, ... , „achtzig“, „neunzig“ ]
```

Wir kommen nun zum Kern des Verfahrens, der Funktion sprichDreier, die den Text zu einer höchstens dreistelligen Zahl erzeugt. Der Text wird in der String-Variablen *erg* aufgesammelt: *erg* ist anfangs leer; Text anfügen kann man mit „&“. Zuerst wird die Hunderterstelle angefügt, dann der Rest. Ist der Rest $< 20$, dann entnimmt man den Text direkt dem Array

Den Text einer Zahl zwischen 1 und 999 erzeugen.

```
1  function sprichDreier(zahl)
2      h := zahl/100      { die Hunderter-Ziffer }
3      r := zahl mod 100      { der Rest kleiner als 100 }
4      z := r/10      { die Zehner-Ziffer }
5      e := zahl mod 10      { die Einer-Ziffer }
6      erg := „“      { Text anfangs leer (bleibt leer, falls zahl = 0) }
7  begin
           { zuerst vorhandene Hunderter anfügen: }
8      if h > 0 then erg := erg & kleiner20[h] & „hundert“
           { dann den Rest kleiner 100 umwandeln: }
9      if r > 0 then      { falls es diesen Rest überhaupt gibt }
10         if r < 20 then
                 { kleine Zahlen direkt der Tabelle entnehmen: }
11             erg := erg & kleiner20[r]
12         else      { wenn 10er oder 1er fehlen, dann kein „und“! }
13             if e > 0 then erg := erg & kleiner20[e]      { Einer }
14             if e > 0 and z > 0 then erg := erg & „und“      { „und“ }
15             if z > 0 then erg := erg & zehner[z]      { Zehner }
16         endif
17     endif
18     return erg
19 end
```

*kleiner*20. Andernfalls fügt man an: Einerstelle – „und“ – Zehnerstelle (in dieser Reihenfolge!). Beachte, dass praktisch alles fehlen kann! So wird z. B. der Verbindungstext „und“ zwischen Einern und Zehnern nur dann gebraucht, wenn sowohl Einer als auch Zehner vorhanden sind.

### Sprachliche Feinheiten

Wandeln wir mit SPRICHDREIER dreistellige Zahlen um, dann ergibt sich für viele Zahlen das gewünschte Resultat:

Für 111 ergibt sich „einhundertelf“.
Für 627 ergibt sich „sechshundertsiebenundzwanzig“.
Für 308 ergibt sich „dreihundertacht“.

In einigen Fällen ist das Resultat jedoch noch unbefriedigend:

Für 1 ergibt sich „ein“.
Für 901 ergibt sich „neunhundertein“.

Tatsächlich gibt es immer dann Probleme, wenn die Dreiergruppe mit „01“ endet. Die naheliegende Reparatur, in dem Array kleiner20 den String „ein“ durch „eins“ zu ersetzen, funktioniert leider nicht: Für 41 ergäbe sich dann z. B. „einsundvierzig“. Wie schon erwähnt, sind bezüglich der Endziffer 1 bei großen Zahlen noch mehr Fälle zu berücksichtigen. Wir sagen ja:

„ei**n**tausend“ für 1 000,
„ein**e** Million“ für 1 000 000,
„neunhunderttein**s** Millionen“ für 901 000 000.

Die Lösung besteht aus zwei Maßnahmen:

1. In Dreiergruppen unterscheidet man zwischen Singular (für die 1), Plural und dem Sonderfall, dass die Dreiergruppe auf „01“ endet (z.B. in 901 oder in 001).
2. Die Anhänge an die Dreiergruppe unterscheidet man je nachdem, ob Singular oder Plural vorliegt und nach der Art der Dreiergruppe (Einer, Tausender, Millionen):
   - „eins“ und „neunhunderteins“
   - „ei**n**tausend“ und „neunhunderttein**s**tausend“
   - „ein**e** Million“ und „neunhunderttein**s** Millionen“

### Die Funktion SPRICHANHANG

Für die Anhänge verwenden wir drei Arrays von Strings mit den unterschiedlichen Anhängen, für den deutschen Singular und Plural die gleichnamigen Arrays, für den Sonderfall der Endung auf auf „01“ den Array *nulleins*:

Vereinbarung von Arrays mit den Texten von Anhängen.

```
1  singular : array [1..8] of String :=
2      [„s“, „tausend“, „e Million“, „e Milliarde“, . . . „,e Trilliarde“]
3  plural : array [1..8] of String :=
4      [„“, „tausend “, „ Millionen“, „ Milliarden“, . . . „, Trilliarden“]
5  nulleins : array [1..8] of String :=
6      [„s“, „stausend “, „s Millionen“, „s Milliarden“, . . . „,s Trilliarden“]
```

Damit kann man die Funktion sprichAnhang wie folgt formulieren:

Für die $i$-te Gruppe einen geeigneten Anhang mit dem Gewicht der Gruppe erzeugen. Dabei nichtleere Dreiergruppen durch Zwischenräume trennen.

```
1  function sprichAnhang(i)
2      anh := „“      { leerer Anhang }
3  begin
4      if gruppe[i] > 0 then      {kein Textbeitrag, falls gruppe[i] = 0}
5          if gruppe[i] = 1 then
6              anh := singular[i]      { einfacher Singular }
7          elsif gruppe[i] mod 100 = 1 then
8              anh := nulleins[i]      { gruppe[i] von der Form x01 }
9          else
10             anh := plural[i]      { einfacher Plural }
11         endif
12         anh := anh &  „ “      { einen Zwischenraum einfügen }
13     endif
14     return anh
15 end
```

## Was haben wir gelernt?

Offensichtlich wissen wir jetzt, wie man die Sprachausgabe für ein Auto-Navigationssystem programmieren kann. Da wir uns bei der Entwicklung des Programms an unserem eigenen Vorgehen orientiert haben, ist uns auch bewusst geworden, wie wir selber Zahlennamen systematisch erzeugen. Im Detail ist das ganz schön kompliziert, nicht wahr? Das liegt zum großen Teil daran, dass unsere deutsche Sprache eine so „verzwickte“ Grammatik hat.

Erstaunlich ist aber, wie wenig man auswendig wissen muss, um sehr viele Zahlennamen erzeugen zu können. Alle Benennungen, die der Algorithmus „kennt“, stehen in den Arrays *kleiner*20, *zehner*, *singular*, *plural* und *nulleins*. Insgesamt sind das gerade mal 51 kurze Strings, die man braucht, um die deutschen Namen von $10^{24}$ Zahlen erzeugen zu können!

Das geht so weiter. Um den „aussprechbaren“ Zahlenbereich zu vertausendfachen, benötigt man nur drei weitere Strings: je einen für die Arrays *singular*, *plural* und *nulleins*. Das mit dem Vertausendfachen kann man so

lange weitertreiben, wie man (z.B. bei Wikipedia) weitere Gewichtsbezeichnungen findet.

Für andere Sprachen, die wie das Englische oder das Französische ebenfalls ein dezimales Zahlensystem verwenden, in dem die Zahlennamen in Tausenderschritten gebündelt sind, lassen sich die Zahlennamen „im Prinzip“ auf die gleiche Weise erzeugen. Das heißt, wir können mit vermutlich relativ geringem Aufwand unser Programm an diese anderen Sprachen anpassen. Ein paar Tipps dazu gebe ich im folgenden Abschnitt.

*Anpassbarkeit an veränderte Aufgabenstellungen* ist eine wichtige Eigenschaft moderner Software – in unserem Beispiel die Möglichkeit der Vergrößerung des Zahlenbereichs bzw. der Umstellung auf eine andere Sprache. Wie bei unserem Algorithmus erreicht man dies häufig dadurch, dass man veränderbare Dinge in Datenstrukturen (hier die Arrays) auslagert.

## Zum Weiterlesen und Selbermachen

1. `http://de.wikipedia.org/wiki/Zahlennamen`

   In diesem Wikipedia-Artikel kann man Wissenswertes über Zahlensysteme nachlesen, z.B. welche Namen es für sehr große Zahlen gibt. Man erfährt u. a., dass auf die Billiarden die Trillionen und Trilliarden folgen und dass eine Quattuordezilliarde der Zahl $10^{87}$ enstpricht: Das ist eine 1 mit 87 Nullen!

2. Ein „echtes Programm“ schreiben:

   Wer versucht, den Algorithmus in seiner „Lieblingsprogrammiersprache“ auszuformulieren, um ein ablauffähiges Programm zu erhalten, wird vermutlich rasch an die Grenzen des Datentyps stoßen, mit dem ganze Zahlen dargestellt werden (`int`, `integer`, `long` oder so ähnlich). Tatsächlich ist es geschickter, eine Dezimalkonstante wie 12345678987654321 gleich als String „12345678987654321“ zu speichern. Wenn im Algorithmus *zahl* **mod** 1000 berechnet wird, dann entspricht das genau den letzten drei Stellen der Stringdarstellung. Der Ausdruck *zahl*/1000 besagt, dass man diese letzten drei Stellen von der Zahl abstreicht.

3. Noch größere Zahlen verarbeiten:

   Wie schon angedeutet, lässt sich unser Algorithmus leicht auf größere Zahlbereiche ausweiten. Man muss dazu nur die drei Arrays *singular*, *plural* und *nulleins* um entsprechende Einträge verlängern. Um beispielsweise alle Zahlen bis einschließlich den Quattuordezilliarden verarbeiten zu können, sind je 20 weitere Einträge pro Array erforderlich. Es ist auch nicht schwierig, den Zahlenbereich auf negative Zahlen (und die Null) auszudehnen.

4. Andere Sprachen sprechen:

   Wer mag, kann probieren, das Programm z.B. an englische, französische oder spanische Zahldarstellungen anzupassen. Hier ist ein wenig mehr Aufwand erforderlich: Nicht nur die Arrays *singular*, *plural* und *nulleins* sind in die andere Sprache zu übertragen, sondern auch die Basiszahlnamen in den Arrays *kleiner*20 und *zehner*. In der Funktion SPRICHDREIER hat man beim Englischen eine andere Reihenfolge der Ziffern, nämlich Hunderter-Zehner-Einer und

das verbindende „und“ wird nicht gebraucht. Die Funktion SPRICHANHANG wird beim Englischen wegen fehlender Beugungsformen einfacher.

5. Richard Bird: *Introduction to Functional Programming using Haskell*. Prentice-Hall, 1998.

   Dieses Lehrbuch über „Funktionale Programmierung“ (leider nur in Englisch erhältlich) widmet unserem Problem einen ganzen Abschn. (5.1), in dem ausführlich ein Haskell-Programm entwickelt wird, das englische Zahlen bis zu einer Million aussprechen kann.

# 24

# Mehrheitsbestimmung – Wer wird Klassensprecher?

Thomas Erlebach

University of Leicester, England

Heiko sitzt vor einem Stapel von Zetteln. In seiner Klasse wurde gerade die Wahl eines Schülers oder einer Schülerin zum Klassensprecher durchgeführt. Alle Schüler haben den Namen ihres Wunschkandidaten auf einen Zettel geschrieben, und Heiko hat sich freiwillig gemeldet, das Wahlergebnis zu bestimmen. Zuvor haben sich alle darauf geeinigt, dass ein Kandidat nur dann Klassensprecher werden soll, wenn mehr als die Hälfte der Schüler für ihn gestimmt hat. Falls niemand die absolute Mehrheit der Stimmen bekommen hat, soll die Wahl wiederholt werden. Heikos Aufgabe ist es jetzt also herauszufinden, ob irgendjemand mehr als die Hälfte aller Stimmen bekommen hat.

Wie soll Heiko diese Aufgabe erledigen? Er macht sich keine großen Gedanken und fängt einfach mit der naheliegendsten Methode an. Auf ein Blatt Papier will er alle Namen schreiben, für die gestimmt wurde, und zu jedem Namen eine Strichliste machen, die angibt, wie oft für diese Person gestimmt wurde. Er nimmt einen Wahlzettel nach dem anderen und schaut nach, welcher Name darauf steht. Wenn er den Namen noch nicht auf seinem Blatt stehen hat, so schreibt er ihn auf das Blatt und macht einen Strich daneben. Wenn der Name dagegen bereits auf dem Blatt steht, so macht er bei dem Namen nur einen zusätzlichen Strich. Als Heiko mit allen Wahlzetteln fertig ist, ergibt sich folgendes Bild:


Nun sucht Heiko denjenigen Namen, der am meisten Striche hat, und zählt diese Striche. Wenn die Anzahl der Striche mehr als die Hälfte der Anzahl

Schüler in seiner Klasse ist, so hat dieser Kandidat die Wahl gewonnen. Andernfalls hat niemand eine absolute Mehrheit und es muss eine neue Wahl stattfinden. Heiko sieht, dass Monika die meisten Stimmen bekommen hat, nämlich 14. In Heikos Klasse sind insgesamt 27 Schüler. Da 14 mehr als 13.5 (die Hälfte von 27) ist, hat Monika die absolute Mehrheit bekommen und ist somit zur Klassensprecherin gewählt. Monika war bereits letztes Jahr Klassensprecherin und wird die Aufgabe sicher auch weiterhin gut erledigen. Alle gratulieren Monika zum Gewinn der Wahl.

Später lässt sich Heiko das Auszählen der Wahlzettel noch einmal durch den Kopf gehen und überlegt, wie gut das von ihm verwendete Verfahren denn war. Er musste für jeden Wahlzettel die Liste aller Namen auf seinem Blatt durchgehen und dann entweder einen Strich machen oder den Namen neu in seine Liste aufnehmen. Bei 27 Wahlzetteln war das kein Problem, aber bei einer größeren Wahl würde das wohl ganz schön viel Arbeit machen. Man stelle sich nur eine Wahl mit Hunderten oder Tausenden von Wahlzetteln vor. Da könnte die Liste der Namen sehr, sehr lang werden. Wenn man einen Wahlzettel bearbeitet, würde es dementsprechend lange dauern herauszufinden, ob der Name bereits in der Liste ist oder nicht. Außerdem hat Heiko mit seiner Methode viel mehr Informationen produziert als verlangt: Er hat nicht nur die Wahlsiegerin bestimmt, sondern auch noch für alle Kandidaten gezählt, wie viele Stimmen sie erhalten haben; letztere Information wäre eigentlich gar nicht nötig gewesen, um seine Aufgabe zu lösen. Vielleicht hätte man die Berechnung unnötiger Informationen vermeiden und die Aufgabe dann mit weniger Aufwand lösen können. An dieser Stelle wäre noch anzumerken, dass es allgemein auch aus Datenschutzgründen oft sehr wichtig ist, das unnötige Erheben und Auswerten von Daten zu vermeiden und nur die Informationen zu generieren, die zur Erfüllung der gestellten Aufgabe unbedingt notwendig sind; auf diese Weise wird die Gefahr des Missbrauchs persönlicher Daten deutlich eingeschränkt. Wir können hier aber nicht weiter auf diese Datenschutzproblematik eingehen.

Heiko hat sich in letzter Zeit etwas mit Algorithmen beschäftigt und weiß inzwischen, dass es in vielen Fällen geschickte Verfahren gibt, die deutlich schneller als die naheliegendste Methode sind. Er beschließt daher, der Sache auf den Grund zu gehen und nachzuforschen, ob es eine schnellere Methode zur Bestimmung der Mehrheit gibt. Zusammen mit Laura, die sich ebenfalls für Algorithmen interessiert, schlägt Heiko in mehreren Büchern über Algorithmen nach, ob etwas zu diesem Problem zu finden ist.

## Majority-Algorithmus

Laura und Heiko finden heraus, dass das Problem, unter $N$ Elementen das Mehrheitselement (d.h. ein Element, das mehr als $N/2$ mal vorkommt) zu bestimmen, unter dem Namen Majority (Majority ist das englische Wort für

Mehrheit) behandelt wird. Sie stoßen auf eine Beschreibung des folgenden Algorithmus.

**Majority-Algorithmus**

1 Verwende einen Stapel von Elementen, der anfangs leer ist.
2 Phase 1: Bearbeite nacheinander jedes der $N$ gegebenen Elemente und führe dabei für jedes Element X das folgende aus:
3 Falls der Stapel leer ist, so lege X oben auf den Stapel.
4 Andernfalls vergleiche X mit dem obersten Element des Stapels. Falls X und dieses Element gleich sind, so lege X oben auf den Stapel; falls die beiden Elemente nicht gleich sind, so entferne das oberste Element des Stapels.
5 Falls der Stapel leer ist, so melde, dass es kein Mehrheitselement gibt.
6 Phase 2: Andernfalls nimm das oberste Element Y des Stapels und zähle, wie oft $Y$ unter allen $N$ gegebenen Elementen vorkommt. Wenn Y mehr als $N/2$ mal vorkommt, so gib Y als Antwort aus. Wenn Y nicht mehr als $N/2$ mal vorkommt, so melde, dass es kein Mehrheitselement gibt.

Laura und Heiko sind recht verblüfft, dass das funktionieren soll. Der Algorithmus würde das Auszählen einer großen Wahl in der Tat einfacher machen: Bei jedem Wahlzettel müsste man den Namen nur mit einem einzigen Namen vergleichen, nämlich mit dem Namen auf dem Wahlzettel, der oben auf dem Stapel liegt. Am Ende müsste man zwar noch ein zweites Mal alle Wahlzettel durchgehen und zählen, wie oft der in Phase 1 bestimmte Name insgesamt vorkommt, aber das wäre auch keine große Sache. Der Algorithmus klingt also interessant, aber Heiko und Laura bezweifeln noch, dass das wirklich funktioniert. Es sieht so aus, als ob am Ende von Phase 1 irgendein Element oben auf dem Stapel liegen könnte, das unter den letzten Elementen häufig vorkommt, obwohl vielleicht ein ganz anderes Element das Mehrheitselement ist. Sie sind also skeptisch und probieren daher erst einmal aus, wie der Algorithmus mit ein paar Beispieleingaben umgeht. Nehmen wir z. B. als Eingabe die folgenden $N = 7$ Elemente (der Einfachheit halber kennzeichnen wir Elemente durch Großbuchstaben): B, B, A, A, C, A, A. In Phase 1 läuft der Algorithmus wie folgt ab (jede Zeile der folgenden Tabelle entspricht einem Schritt des Algorithmus):

| **Stapel (v.u.n.o.)** | **Betrachtetes Element** | **Aktion** |
|---|---|---|
| leer | B | B auf den Stapel legen |
| B | B | B auf den Stapel legen |
| B, B | A | B vom Stapel entfernen |
| B | A | B vom Stapel entfernen |
| leer | C | C auf den Stapel legen |
| C | A | C vom Stapel entfernen |
| leer | A | A auf den Stapel legen |
| A | | |

Tatsächlich, am Ende liegt ein A auf dem Stapel. Der Algorithmus wird nun in Phase 2 zählen, wie oft A unter den gegebenen Elementen vorkommt. Da A unter den $N = 7$ Elementen B, B, A, A, C, A, A viermal vorkommt, wird A als Mehrheitselement erkannt und ausgegeben. In diesem Beispiel hat der Algorithmus also richtig funktioniert. Man sieht auch, dass zu jeder Zeit alle Elemente auf dem Stapel gleich sind; das ist ja eigentlich klar, da der Algorithmus nur dann ein Element auf den Stapel legt, wenn der Stapel leer ist oder wenn das Element gleich dem obersten Element auf dem Stapel ist.

Was macht der Algorithmus, wenn die Eingabe kein Mehrheitselement enthält, also z.B. bei der Eingabe A, B, C, C? Hier kommt C zwar unter $N = 4$ Elementen zweimal vor, aber ein Mehrheitselement müsste ja echt mehr als $N/2$ mal vorkommen. Der Algorithmus würde in Phase 1 wie folgt ablaufen:

| **Stapel (v.u.n.o.)** | **Betrachtetes Element** | **Aktion** |
|---|---|---|
| leer | A | A auf den Stapel legen |
| A | B | A vom Stapel entfernen |
| leer | C | C auf den Stapel legen |
| C | C | C auf den Stapel legen |
| C, C | | |

Hier ist am Ende von Phase 1 also ein C oben auf dem Stapel. Man sieht somit, dass Phase 2 des Algorithmus wirklich nötig ist. In Phase 2 zählt der Algorithmus nämlich, wie oft das C in der Eingabe vorkommt. Da es nur zweimal vorkommt, erkennt der Algorithmus richtig, dass die Eingabe kein Mehrheitselement enthält.

Wie wäre es mit einer Eingabe der Form A, A, A, B, B, bei der am Ende ein anderes Element als das Mehrheitselement oft vorkommt? Auch hier lässt sich der Algorithmus nicht täuschen:

| **Stapel (v.u.n.o.)** | **Betrachtetes Element** | **Aktion** |
|---|---|---|
| leer | A | A auf den Stapel legen |
| A | A | A auf den Stapel legen |
| A, A | A | A auf den Stapel legen |
| A, A, A | B | A vom Stapel entfernen |
| A, A | B | A vom Stapel entfernen |
| A | | |

Am Ende liegt ein A oben auf dem Stapel, also findet der Algorithmus auch hier wieder das korrekte Mehrheitselement. Irgendwie scheint der Algorithmus also vielleicht doch richtig zu sein, obwohl Laura und Heiko immer noch nicht richtig verstehen, wieso er eigentlich funktioniert. Die beiden schauen noch einmal in das Algorithmenbuch. Dort finden sie einen Beweis für die Korrektheit des Algorithmus. Zuerst scheint der Beweis recht kompliziert und schwer nachzuvollziehen, doch Laura und Heiko arbeiten ihn ein paar Mal durch und helfen einander, indem sie sich gegenseitig die Teile des Beweises erklären,

die sie jeweils schon verstanden haben. Am Ende verstehen sie, warum der Algorithmus tatsächlich immer funktioniert.

## Korrektheit des Majority-Algorithmus

Der Korrektheitsbeweis für den Majority-Algorithmus lässt sich wie folgt zusammenfassen: Wenn der Algorithmus ein Element X ausgibt, so muss dieses Element tatsächlich das Mehrheitselement sein, da der Algorithmus in Phase 2 ja bestätigt hat, dass X mehr als $N/2$-mal vorkommt. Somit könnte der Algorithmus also nur dann ein falsches Ergebnis liefern, wenn die Eingabe ein Mehrheitselement X enthält, am Ende von Phase 1 aber entweder der Stapel leer ist oder ein anderes Element als X oben auf dem Stapel liegt (und der Algorithmus dann ausgeben würde, dass es kein Mehrheitselement gibt). Die folgenden Überlegungen zeigen, dass dieser Fall nie eintreten kann.

Betrachten wir eine beliebige Eingabe mit $N$ Elementen, unter denen mehr als $N/2$ Elemente gleich X sind. Nehmen wir an, dass am Ende von Phase 1 des Algorithmus kein X oben auf dem Stapel liegt. Da der Stapel immer nur gleiche Elemente enthält, liegt also am Ende von Phase 1 überhaupt kein X im Stapel. Dann muss jedes X eines der folgenden beiden Schicksale erlitten haben:

1. Als X bearbeitet wurde, enthielt der Stapel ein oder mehrere Elemente ungleich X. Daher wurde X nicht auf den Stapel gelegt, sondern ein Element Y vom Stapel entfernt.
2. Als X bearbeitet wurde, war der Stapel leer oder enthielt Elemente gleich X. Daher wurde X auf den Stapel gelegt. Da am Ende von Phase 1 kein X mehr im Stapel liegt, muss nach diesem X später einmal ein Element Z gekommen sein, das dazu geführt hat, dass dieses X wieder vom Stapel entfernt wurde.

Gemäß diesen Überlegungen kann man jedes X-Element also einem Element ungleich X zuordnen: In Fall 1 wird X dem betreffenden Element Y zugeordnet, im Fall 2 dem betreffenden Element Z. Weiter sieht man, dass keine zwei Elemente X auf diese Weise demselben Element ungleich X zugeordnet werden. Es folgt also, dass es mindestens so viele Elemente ungleich X gibt, wie es Elemente gleich X gibt. Da es insgesamt nur $N$ Elemente gibt, ist dies ein Widerspruch zu der Voraussetzung, dass es mehr als $N/2$ viele Elemente gleich X gibt. Die Annahme, dass am Ende von Phase 1 kein X oben auf dem Stapel liegt, führt also auf einen Widerspruch und muss somit falsch sein. Es folgt daher, dass am Ende von Phase 1 wirklich ein X oben auf dem Stapel liegt und der Algorithmus somit korrekt X als Mehrheitselement erkennt.

## Wie viele Vergleiche sind nötig?

Es ist auch interessant zu betrachten, wie viele Vergleiche zwischen Elementen ein Algorithmus zur Mehrheitsberechnung machen muss, bis das Ergebnis bestimmt ist. Als Vergleich bezeichnen wir hier die Operation, die prüft, ob zwei Elemente gleich sind oder nicht. Der oben beschriebene Majority-Algorithmus macht in Phase 1 höchstens $N-1$ Vergleiche: Das erste Element wird ja einfach ohne einen Vergleich auf den Stapel gelegt, und jedes andere Element wird höchstens mit dem Element oben auf dem Stapel verglichen. Phase 2 kann ebenfalls leicht mit $N-1$ Vergleichen realisiert werden. Insgesamt macht der Algorithmus also auf einer Eingabe mit $N$ Elementen höchstens $2N-2$ Vergleiche.

Da stellt sich natürlich die Frage: Geht es besser, d.h. mit weniger als $2N-2$ Vergleichen? Die Antwort ist ja, denn die folgende Abwandlung des Majority-Algorithmus kommt immer mit höchstens $\lceil 3N/2 \rceil - 2$ Vergleichen aus. (Die Notation $\lceil . \rceil$ bezeichnet hier den zur nächsten ganzen Zahl aufgerundeten Wert; für $N = 5$ gilt z. B. $\lceil 3N/2 \rceil = \lceil 15/2 \rceil = \lceil 7,5 \rceil = 8$, und für $N = 6$ hätten wir $\lceil 3N/2 \rceil = \lceil 18/2 \rceil = \lceil 9 \rceil = 9$.)

Verfeinerter Majority-Algorithmus

```
1  Verwende zwei Stapel von Elementen, die anfangs beide leer sind.
2  Phase 1: Bearbeite nacheinander jedes der N gegebenen Elemente und
   führe dabei für jedes Element X das Folgende aus:
3      Falls Stapel 2 nicht leer ist und X gleich dem Element oben auf
       Stapel 2 ist, so lege X auf Stapel 1.
4      Andernfalls lege X auf Stapel 2 und, falls Stapel 1 nicht leer ist,
       entferne das oberste Element von Stapel 1 und lege es auf Stapel 2.
5  Nimm an, dass am Ende von Phase 1 ein Y das oberste Element auf
   Stapel 2 ist.
6  Phase 2: Solange Stapel 2 nicht leer ist, wiederhole folgende Operatio-
   nen:
7      Vergleiche das oberste Element auf Stapel 2 mit Y.
8      Falls die beiden Elemente gleich sind, entferne die beiden obersten
       Elemente von Stapel 2. (Falls Stapel 2 nur ein Element enthält, ent-
       ferne es von Stapel 2 und lege es auf Stapel 1.)
9      Andernfalls (d.h. falls die beiden Elemente nicht gleich sind) entfer-
       ne das oberste Element von Stapel 1 und das oberste Element von
       Stapel 2. (Falls Stapel 1 leer war und daher kein Element von Sta-
       pel 1 entfernt werden kann, so terminiere und gib aus, dass es kein
       Mehrheitselement gibt.)
10 Falls Stapel 1 nicht leer ist, gib Y als Mehrheitselement aus. Andernfalls
   gib aus, dass es kein Mehrheitselement gibt.
```

Wir beobachten, dass schon die Beschreibung der einzelnen Schritte des verfeinerten Majority-Algorithmus recht kompliziert ist. Hier ist es nicht nur schwierig, sich von der Korrektheit des Algorithmus zu überzeugen, man muss auch bei der Ausführung des Algorithmus auf einer Beispieleingabe aufpas-

sen, dass man die einzelnen Schritte alle richtig ausführt. Um den verfeinerten Majority-Algorithmus besser zu verstehen, betrachten wir daher erst einmal, wie bei ihm auf der Eingabe

B, B, A, A, C, D, A, A, A

die Phase 1 abläuft:

| **Stapel 1 (v.u.n.o.)** | **Stapel 2 (v.u.n.o.)** | **Betrachtetes Element** | **Aktion** |
|---|---|---|---|
| leer | leer | B | B auf Stapel 2 legen |
| leer | B | B | B auf Stapel 1 legen |
| B | B | A | A auf Stapel 2 legen, ein B von Stapel 1 auf Stapel 2 legen |
| leer | B,A,B | A | A auf Stapel 2 legen |
| leer | B,A,B,A | C | C auf Stapel 2 legen |
| leer | B,A,B,A,C | D | D auf Stapel 2 legen |
| leer | B,A,B,A,C,D | A | A auf Stapel 2 legen |
| leer | B,A,B,A,C,D,A | A | A auf Stapel 1 legen |
| A | B,A,B,A,C,D,A | A | A auf Stapel 1 legen |
| A,A | B,A,B,A,C,D,A | | |

Wir sehen, dass das Mehrheitselement A am Ende von Phase 1 tatsächlich oben auf Stapel 2 liegt. Phase 2 läuft nun wie folgt ab:

| **Stapel 1 (v.u.n.o.)** | **Stapel 2 (v.u.n.o.)** | **Aktion** |
|---|---|---|
| A,A | B,A,B,A,C,D,A | Element oben auf Stapel 2 ist gleich A (das muss automatisch so sein, da wir A ja als das Element gewählt haben, das am Ende von Phase 1 oben auf Stapel 2 lag), also entferne zwei Elemente (D,A) von Stapel 2 |
| A,A | B,A,B,A,C | Element C oben auf Stapel 2 ist ungleich A, also entferne ein Element von Stapel 2 und ein Element von Stapel 1 |
| A | B,A,B,A | Element oben auf Stapel 2 ist gleich A, also entferne zwei Elemente (B,A) von Stapel 2 |
| A | B,A | Element oben auf Stapel 2 ist gleich A, also entferne zwei Elemente (B,A) von Stapel 2 |
| A | leer | |

Nun ist der Stapel 2 leer, und Phase 2 endet somit. Da Stapel 1 nicht leer ist und ein A enthält, gibt der Algorithmus korrekt das Mehrheitselement

A aus. Wir sehen auch, dass der Algorithmus in Phase 2 nur 4 Vergleiche gemacht hat. Beim ersten der vier Vergleiche war das Ergebnis des Vergleichs außerdem sowieso klar (was von der Bemerkung in Klammern erklärt wird), es mussten also in Phase 2 eigentlich nur drei Vergleiche gemacht werden.

Wir wollen kurz skizzieren, wieso der verfeinerte Majority-Algorithmus richtig funktioniert. Die Rolle des Stapels aus dem ersten Majority-Algorithmus wird hier im Wesentlichen von Stapel 1 zusammen mit dem obersten Element von Stapel 2 übernommen. Ferner gilt, dass Stapel 1 immer identische Elemente enthält (die auch gleich dem obersten Element auf Stapel 2 sind) und dass auf Stapel 2 niemals zwei gleiche Elemente direkt aufeinander zu liegen kommen. Daraus folgt, dass am Ende von Phase 1 das Element Y oben auf Stapel 2 der einzige Kandidat für das Mehrheitselement ist. In jeder Iteration von Phase 2 werden dann zwei Elemente entfernt, von denen eines (aber niemals beide) gleich Y ist. Y ist also genau dann das Mehrheitselement, wenn am Ende von Phase 2 der Stapel 1 nicht leer ist (d.h. noch mindestens ein Y enthält). Somit folgt die Korrektheit des verfeinerten Majority-Algorithmus. Ferner macht der Algorithmus in Phase 1 höchstens $N - 1$ Vergleiche und in Phase 2 höchstens $\lceil N/2 \rceil - 1$ Vergleiche (davon kann man sich überzeugen, indem man in Betracht zieht, dass in jeder Iteration von Phase 2 mit einem Vergleich zwei Elemente entfernt werden; das lässt sich auch an dem obigen Beispiel gut beobachten). Somit sieht man, dass der Algorithmus insgesamt höchstens $\lceil 3N/2 \rceil - 2$ Vergleiche braucht, um das Mehrheitselement zu bestimmen (oder herauszufinden, dass es keines gibt).

Geht es noch besser? Dieses Mal ist die Antwort nein! Man kann zeigen, dass jeder Algorithmus auf manchen Eingaben mit $N$ Elementen mindestens $\lceil 3N/2 \rceil - 2$ Vergleiche machen muss, um das Mehrheitselement zu bestimmen. Besser als beim verfeinerten Majority-Algorithmus geht es also im Allgemeinen bezüglich der Anzahl der Vergleiche nicht.

## Anwendungen und Erweiterungen

Das Problem der Bestimmung des Mehrheitselements tritt nicht nur bei der Auswertung von Wahlen auf, sondern auch in ganz anderen Anwendungen. Zum Beispiel kann man sicherheitskritische Berechnungen, bei denen ein korrektes Ergebnis extrem wichtig ist, von $N$ verschiedenen Prozessoren unabhängig voneinander ausführen lassen und am Ende unter allen $N$ Ergebnissen das Mehrheitselement bestimmen und als Ergebnis verwenden. Auf diese Weise bekommt man immer das richtige Ergebnis, sofern weniger als die Hälfte der $N$ Prozessoren fehlerhaft sind und ein falsches Ergebnis liefern.

Das Mehrheitselement unter $N$ Elementen ist das Element, das mehr als $N/2$ mal vorkommt. Etwas allgemeiner kann man *häufige* Elemente betrachten, die mehr als $N/K$ mal vorkommen, wobei $K$ eine feste Zahl ist. Für

$K = 10$ etwa sind das die Elemente mit mehr als 10% Häufigkeit. Interessant sind häufige Elemente z. B. bei der Beobachtung des Datenverkehrs im Internet, wo man gerne wissen möchte, welche Anwendungen oder welche Benutzer den meisten Verkehr produzieren. Da die Datenpakete extrem schnell bearbeitet werden müssen, ist hier ein Algorithmus nötig, der jedes neue Datenpaket in kürzestmöglicher Zeit erledigt. Der Majority-Algorithmus lässt sich auf solche Problemstellungen verallgemeinern.

## Welche Lehren können wir aus den Lösungen des Problems der Mehrheitsbestimmung ziehen?

- Die naheliegendste Methode ist nicht automatisch die schnellste.
- Oft gibt es geschickte Algorithmen, die dieselbe Aufgabe mit viel weniger Aufwand lösen können.
- Es ist nicht immer leicht zu sehen, dass ein Algorithmus tatsächlich immer das *richtige Ergebnis* liefert.
- Manchmal kann man zeigen, dass es unmöglich ist, einen noch besseren Algorithmus für ein Problem zu finden.

## Zum Weiterlesen

1. Kapitel 1 (Binäre Suche)

   Hier geht es um Binärsuche, also ein Verfahren, mit dem in einem sortierten Array sehr schnell nach einem Wert gesucht werden kann. Vor einer Wahl könnte man eine Liste aller Kandidaten sortiert in einem Array speichern, zusammen mit einem Zähler für jeden Kandidaten, der anfangs Null ist. Bei der Bearbeitung eines Wahlzettels könnte man dann mit Binärsuche schnell den betreffenden Kandidaten im Array finden und den Stimmenzähler für diesen Kandidaten um Eins erhöhen.

2. Kapitel 3 (Schnelle Sortieralgorithmen)

   Hier werden schnelle Sortieralgorithmen beschrieben. Mit diesen Algorithmen könnte man eine Kandidatenliste schnell in eine sortierte Reihenfolge bringen.

3. Weitere Informationen zum Majority-Algorithmus und seinen Erweiterungen findet man in Originalartikeln, die in Fachzeitschriften oder Konferenzbänden veröffentlicht worden sind. Sowohl der verfeinerte Majority-Algorithmus als auch der Beweis, dass jeder Algorithmus zur Mehrheitsbestimmung auf manchen Eingaben mindestens $\lceil 3N/2 \rceil - 2$ Vergleiche machen muss, werden in folgendem Artikel vorgestellt:

   M.J. Fischer, S.L. Salzberg: *Finding a majority among N votes*. Journal of Algorithms 3(4):375–379, 1982.

Die Verallgemeinerung des Majority-Algorithmus auf das Problem, häufige Elemente in einem Datenstrom zu identifizieren, wird in folgendem Artikel beschrieben:

> J. Misra, D. Gries: *Finding repeated elements*. Science of Computer Programming 2:143–152, 1982.

Weitere Verfeinerungen des Algorithmus zur Identifikation häufiger Elemente eines Datenstroms im Kontext der Analyse von Paketströmen im Internet werden z.B. in diesem Artikel diskutiert:

> E.D. Demaine, A. López-Ortiz, I. Munro: *Frequency Estimation of Internet Packet Streams with Limited Space*. Proceedings of the 10th Annual Symposium on Algorithms (ESA 2002), LNCS 2461, Springer, 2002.

# 25

# Zufallszahlen: Wie kommt der Zufall in den Rechner?

Bruno Müller-Clostermann und Tim Jonischkat

Universität Duisburg-Essen

Algorithmen sind clevere Verfahren, die Probleme verschiedenster Art effizient lösen. Wir haben in den voranstehenden Kapiteln zahlreiche Beispiele für „normale“ Algorithmen gesehen, wie z. B. Sortieren durch Einfügen, Tiefensuche in Graphen und kürzeste Wege. Danach könnte man vermuten, dass Algorithmen trotz aller Cleverness und Effizienz nur sture und gleich ablaufende Verfahren sind, die immer perfekte und eindeutige Lösungen liefern. Mit Zufall haben Algorithmen scheinbar nichts zu tun. Aber Halt! Bei QuickSort wird vorgeschlagen, das Pivot-Element zufällig zu wählen. Beim Verfahren „One-Time-Pad“ werden die Schlüssel zufällig gewählt. Beim „Fingerprinting“-Verfahren werden Zahlen zufällig ausgewählt.

Auch bei Taktik- und Strategiespielen auf dem PC werden Algorithmen verwendet, bei denen Zufall sehr erwünscht oder sogar notwendig ist. Der Rechner übernimmt hier oft die Rolle eines Gegenspielers, wobei die Aktionen von Algorithmen gesteuert werden, die sinnvolle und intelligente Verhaltensvariationen erzeugen sollen. Wir kennen das von interaktiven Computerspielen wie *Siedler*, *Sims*, *SimCity*, oder *WarCraft*. Natürlich soll der Rechner bei gleichen Situationen nicht immer auf die gleiche Art reagieren, sondern es sollen unterschiedliche Effekte und Aktionen erzeugt werden. Dadurch kommt mehr Abwechslung und Spannung ins Spiel.

Eine Erzeugung von Zufallszahlen oder von zufälligen Ereignissen, z. B. durch Werfen eines Würfels (Zahlen 1, 2, ..., 6) oder einer Münze (Kopf oder Zahl) ist zur Verwendung in einem programmierten Algorithmus natürlich nicht geeignet. Kann aber „zufälliges“ Verhalten programmiert werden? Kann „Zufall“ durch einen Algorithmus erzeugt werden? Antwort: Der Zufall wird mit Hilfe von Algorithmen nachgeahmt, die scheinbar zufällige Zahlen erzeugen. Diese Zahlen werden als Zufallszahlen oder genau genommen als Pseudo-Zufallszahlen bezeichnet. Wir betrachten hier bekannte und altbewährte Algorithmen zur Erstellung von Zufallsgeneratoren. Für Zufallszahlen gibt es zahlreiche Anwendungsbereiche; hier werden wir zwei davon kennen lernen: ein Computerspiel und die so genannte Monte-Carlo-Simulation zur Flächenberechnung.

## Ein Taktikspiel: Schnick-Schnack-Schnuck

Als einfaches Beispiel für ein programmiertes Spiel können wir uns überlegen, wie ein Algorithmus für das bekannte Spiel „Papier-Schere-Stein“ („Schnick-Schnack-Schnuck“) aussehen könnte. Das Spiel funktioniert so: Bei jeder Spielrunde wählst du eine der drei Möglichkeiten „Papier“, „Schere“ oder „Stein“.

Dann wird der Algorithmus ausgeführt und liefert als Ergebnis ebenfalls „Papier“ oder „Schere“ oder „Stein“. Danach wird ausgewertet. Es gewinnt Papier gegen Stein, Stein gegen Schere und Schere gegen Papier. Der Sieger bekommt einen Punkt und es folgt die nächste Spielrunde.

Wie soll der Algorithmus arbeiten? Soll z. B. abwechselnd „Schere“ und „Stein“ gewählt werden? Das wird der menschliche Spieler schnell durchschauen! Das Ergebnis des Algorithmus muss also *unvorhersehbar* sein, so wie das Werfen eines Würfels, das Ziehen der Lottozahlen oder der Lauf einer Roulette-Kugel.

Die mechanische Kopplung eines Algorithmus mit einem Würfel oder einem Rouletterad wäre ziemlich umständlich, jedenfalls wäre eine solche Konstruktion nicht effizient und auch nicht clever. Gebraucht wird deswegen ein algorithmisches Verfahren, welches unter den Möglichkeiten „Papier“, „Schere“ oder „Stein“ eine scheinbar zufällige Wahl trifft. Solch ein Algorithmus heißt *Zufallsgenerator* oder auch *Zufallszahlengenerator*.

**Abb. 25.1.** Drei Möglichkeiten: „Papier“, „Schere“ oder „Stein“? (Bildnachweis: Tim Jonischkat)

**Abb. 25.2.** Münze: Kopf oder Zahl; Würfel: 1, ..., 6; Rouletterad: 0, 1, ..., 36 (Bildnachweis: Lukasz Wolejko-Wolejszo, Toni Lozano)

## Hilfsmittel zur Erzeugung von Zufallszahlen: Modulo-Rechnung

Bevor wir uns mit der Berechnung von Zufallszahlen näher beschäftigen, benötigen wir das Konzept der Modulo-Rechnung. Die Modulo-Funktion (auch mod-Funktion genannt) bestimmt den Rest, der bei der Divison zweier natürlicher Zahlen entsteht. Teilen wir z. B. die Zahl 27 durch 12, so bleibt als Rest die Zahl 3.

Betrachten wir zwei beliebige natürliche Zahlen $x$ und $m$ und dividieren $x$ durch $m$, dann erhalten wir einen (ganzzahligen) Quotienten $a$ und einen Rest $r$. Dieser Rest $r$ ist eine natürliche Zahl aus dem Intervall $\{0, 1, \ldots, \text{m-1}\}$.

## Einige Beispiele zur Modulo-Rechnung

- $9 \bmod 8 = 1$
- $16 \bmod 8 = 0$
- $(9 + 6) \bmod 12 = 15 \bmod 12 = 3$
- $(6 \cdot 2 + 15) \bmod 12 = 27 \bmod 12 = 3$
- $1143 \bmod 1000 = 143$
  Bei Division durch 1000 besteht der Rest aus den letzten 3 Stellen

## Veranschaulichung der Modulo-Rechnung

Wir können uns die Modulo-Rechnung wie das Wandern entlang eines Kreises vorstellen, auf dem die Zahlen 0, 1, 2, ..., $m - 1$ aufgetragen sind. Um $x$ modulo $m$, also den Rest von $x$ bei der Division durch $m$ zu bestimmen, starten wir bei 0 und gehen dann $x$ Schritte im Uhrzeigersinn. Die Anzahl der Umrundungen des Kreises ist gegeben durch das Ergebnis der ganzzahligen Division $a$; die Zahl, bei der wir landen, ist der Rest $r$.

Startet man z. B. eine Stundenzählung mit einer Analoguhr am 1. Januar um 0 Uhr, dann sind am 2. Januar um 19 Uhr zwar 43 Stunden vergangen, aber der Stundenzeiger zeigt auf 7 Uhr, das ist der Rest $r = 43 \bmod 12$. Eine Analoguhr zeigt die Stunden modulo 12 an, dies ist der Rest, der sich bei der Division der vergangenen Stunden durch 12 ergibt. Eine Addition entspricht einer Bewegung in Uhrzeigerrichtung. Betrachten wir im Folgenden zwei Beispiele (vgl. Abb. 25.3):

Starten wir z. B. bei $x = 9$ und addieren 6, dann bewegen wir uns 6 Schritte in Uhrzeigerrichtung und erreichen die 3, d.h., es gilt $(9+6) \bmod 12 = 3$ (Abb. 25.3, linkes Bild).

Die Multiplikation einer Zahl $x$ mit einem Faktor $a$ kann man sich als eine Folge von Additionsschritten vorstellen; starten wir z. B. bei dem Wert $x = 2$, und multiplizieren mit $a = 6$, dann addieren wir 5-mal den Wert 2, machen

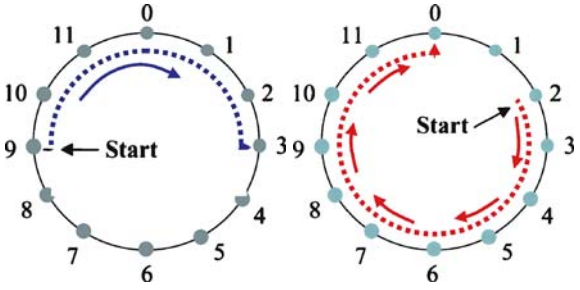



**Abb. 25.3.** Zwei Beispiele zur Modulo-Rechnung: $9 \bmod 6 = 3$ und $(6 \cdot 2) \bmod 12 = 0$

also 5 Zweierschritte im Uhrzeigersinn und erreichen den Wert 0, d.h., es gilt $(6 \cdot 2) \bmod 12 = 0$ (Abb. 25.3, rechtes Bild).

Die Modulo-Arithmetik ist in der Informatik die „natürliche“ Arithmetik, weil Rechner einen begrenzten (d.h. endlichen) Speicher haben und auch die Speicherzellen nur endlich groß sind und daher Zahlen nur bis zu einer bestimmten Größe gespeichert werden können.

## Ein Algorithmus zur Erzeugung von Pseudo-Zufallszahlen

Der folgende Algorithmus erzeugt Zufallszahlen aus dem Intervall $\{0, 1, \ldots, m-1\}$, wobei wir die gerade vorgestellte Modulo-Rechnung verwenden. Das Grundprinzip ist einfach: Wir nehmen einen Startwert $x$, führen eine kleine Rechnung der Form $a \cdot x + c$ aus und berechnen den Rest $r$ durch $r = (a \cdot x + c) \bmod m$. Das Ergebnis ist die erste Zufallszahl $x_1 = r$. Wir verwenden $x_1$ als neuen Ausgangswert und berechnen die zweite Zufallszahl $x_2$. Dadurch ergibt sich eine Folge von Zufallszahlen $x_1, x_2, x_3, \ldots$

Diesen Algorithmus für einen Zufallszahlengenerator können wir auch so aufschreiben:

$$
\begin{aligned}
x_1 &:= (a \cdot x_0 + c) \bmod m \\
x_2 &:= (a \cdot x_1 + c) \bmod m \\
x_3 &:= (a \cdot x_2 + c) \bmod m \\
x_4 &:= (a \cdot x_3 + c) \bmod m \\
&\ldots \text{usw.} \ldots
\end{aligned}
$$

In allgemeiner Schreibweise erhalten wir die iterative Rechenvorschrift:

$$x_{i+1} := (a \cdot x_i + c) \bmod m,\ i = 0, 1, 2, \ldots$$

Um ein konkretes Beispiel für einen Zufallszahlengenerator zu erhalten, müssen wir die Parameter $a$, $c$, $m$ und den Startwert $x_0$ festlegen. Mit den

Festlegungen $a = 5$, $c = 1$, $m = 16$ und dem Startwert $x_0 = 1$ ergibt sich folgende Rechnung.

$$
\begin{aligned}
x_1 &:= (5 \cdot 1 \quad +1) \bmod 16 = 6 \\
x_2 &:= (5 \cdot 6 \quad +1) \bmod 16 = 15 \\
x_3 &:= (5 \cdot 15 \ +1) \bmod 16 = 12 \\
x_4 &:= (5 \cdot 12 \ +1) \bmod 16 = 13 \\
x_5 &:= (5 \cdot 13 \ +1) \bmod 16 = 2 \\
x_6 &:= (5 \cdot 2 \quad +1) \bmod 16 = 11 \\
x_7 &:= (5 \cdot 11 \ +1) \bmod 16 = 8 \\
x_8 &:= (5 \cdot 8 \quad +1) \bmod 16 = 9 \\
&\ldots \text{usw.} \ldots
\end{aligned}
$$

Natürlich ist diese Zahlenfolge offensichtlich deterministisch, d.h., sie besteht bei einem gegebenen $x_0$ immer aus den gleichen fest definierten und reproduzierbaren Elementen. Durch die Wahl eines Startwerts $x_i$ kann der Einstiegspunkt in solch eine Zahlenfolge für jeden Ablauf des Algorithmus neu festgelegt werden.

## Periodisches Verhalten

Rechnen wir weiter, dann fällt auf, dass wir nach 16 Schritten wieder bei dem Ausgangswert 1 ankommen, und dass jede der 16 möglichen Zahlen $0, 1, 2, \ldots, 15$ genau einmal vorkommt. Berechnet man die Werte für $x_{16}, x_{17}, \ldots, x_{31}$, wiederholt sich die Zahlenfolge. Wir sehen also ein periodisches Verhalten, hier mit der Periode 16. Wenn wir $m$ sehr groß wählen und außerdem den Faktor $a$ und die Konstante $c$ geschickt festlegen, dann erhalten wir größere Perioden, im Idealfall erhält man wie in diesem Beispiel die *volle Periode* der Länge $m$. In Programmiersprachen sind manchmal Zufallszahlengeneratoren fest „eingebaut“ oder über eine „Bibliothek“ von Funktionen verwendbar; es gibt z. B. in der Programmiersprache Java einen vollperiodigen Generator mit den Parametern $a = 252149003917$, $c = 11$ und $m = 2^{48}$.

## Simulation von echten Zufallsgeneratoren

Die Erzeugung von Pseudo-Zufallszahlen aus dem Bereich $\{0, 1, \ldots, m-1\}$ ist die Grundlage für viele Anwendungen. Beispiele sind die Simulation eines Münzwurfs mit den Ergebnissen Kopf oder Zahl, das Werfen eines Würfels mit den Ergebnissen $1, 2, 3, 4, 5$ und 6 oder das Drehen eines Rouletterads mit den 37 Möglichkeiten $0, 1, \ldots, 36$.

Nehmen wir an, dass es bei allen Beispielen fair zugeht, d.h., die Wahrscheinlichkeit für Kopf bzw. Zahl ist jeweils $\frac{1}{2}$, beim Würfel haben wir $\frac{1}{6}$ und beim Roulette $\frac{1}{37}$ als Wahrscheinlichkeiten für jeden der möglichen Werte. Zur Simulation des Münzwurfs benötigen wir also ein Verfahren zur Umwandlung einer Zufallszahl $x \in \{0, 1, \ldots, m-1\}$ in eine Zahl $z \in \{0, 1\}$, wobei 0 für „Kopf“ und 1 für „Zahl“ stehen soll. Ein einfaches Verfahren wäre eine Rechnung, bei der „kleine“ Zahlen auf 0 und „große“ Zahlen auf 1 abgebildet werden, d.h. rechnerisch: 0 wenn $x < \frac{m}{2}$, 1 wenn $x \geq \frac{m}{2}$. Beim Roulette hätten wir eine Transformation $z := x \bmod 37$ und beim Würfel $z := x \bmod 6 + 1$.

## Der Schnick-Schnack-Schnuck-Algorithmus

Jetzt endlich zurück zu unserem „Taktikspiel“. Wir benötigen einen Algorithmus, der unter den drei Möglichkeiten „Papier“, „Schere“ oder „Stein“ eine Zufallsauswahl trifft. Zu diesem Zweck verwenden wir einen Zufallszahlengenerator, mit dem wir bei jeder Spielrunde eine Zufallszahl $x$ erzeugen und daraus durch $z := x \bmod 3$ eine neue Zahl bestimmen, die nur die drei Werte 0, 1 und 2 annehmen kann. Abhängig vom Wert von $z$ wählt der Algorithmus „Papier“ (0), „Schere“ (1) oder „Stein“ (2). Dieser Algorithmus ist in einem kleinen Programm implementiert, dem „Schnick-Schnack-Schnuck-Applet“[1].

Es stehen vier Zufallszahlengeneratoren zur Auswahl, wobei der erste durch den extremen Spezialfall einer fest vorgegebenen Folge der Zahlen 0, 1 und 2 definiert ist.

1. Deterministisch: Die feste Zahlenfolge $2, 0, 1, 1, 0, 0, 0, 2, 1, 0, 2$
2. ZZG-016: $a = 5$, $c = 1$, $m = 16$ und Startwert $x_0 = 1$ (Periode 16)
3. ZZG-100: $a = 81$, $c = 1$, $m = 100$, Startwert $x_0 = 10$ (Periode 100)
4. Java-Generator: Der in der Programmiersprache Java vorgegebene Generator „java.util.Random“

Der Algorithmus NAECHSTEZUFALLSZAHL rechnet mit der Eingabe $x$ und den Konstanten $a$, $c$ und $m$. Der mit **return** zurückgegebene Wert ist die auf die parametrisierte Zahl in der jeweiligen Zahlenreihe folgende Zufallszahl im Intervall $\{0, 1, \ldots, m-1\}$.

```
1  procedure NAECHSTEZUFALLSZAHL (x)
2  begin
3     return (a · x + c) mod m
4  end
```

[1] `http://www-i1.informatik.rwth-aachen.de/~algorithmus/Algorithmen/algo38/applet/` (bitte Java aktivieren)

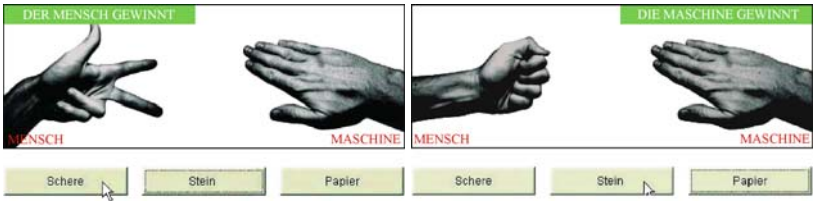



**Abb. 25.4.** Schere schneidet Papier (*links*), Stein wird von Papier eingewickelt (*rechts*)

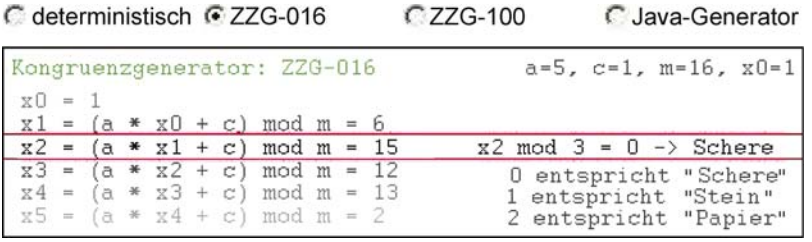



**Abb. 25.5.** Algorithmus der Maschine

In diesem Beispiel wird die Funktion NAECHSTEZUFALLSZAHL $n$-mal aufgerufen; der Startwert wird auf den Wert 1 gesetzt. Bei jeder Wiederholung wird die ermittelte Zufallszahl und zusätzlich ihr Wert modulo 3 ausgegeben.

```
1   procedure ZUFALLSZAHLENBEISPIEL (n)
2   begin
3       a := 5; c := 1; m := 16;
4       x := 1;
5       for i := 1 to n do
6           x := NAECHSTEZUFALLSZAHL(x);
7           print(x);
8           print(x mod 3)
9       endfor
10  end
```

Als *Zufallszahlen* erhalten wir
$x_i : 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, \ldots$
$x_i$ modulo $3 : 0, 0, 0, 1, 2, 2, 2, 1, 2, 1, \ldots$

Die Bilder zeigen einen möglichen Spielverlauf. „Mensch“ und „Maschine“ wählen gleichzeitig „Schere“, „Stein“ oder „Papier“. Tabelle 25.1 zeigt den Spielstand und den Ablauf des Algorithmus.

In Abb. 25.5 kann man die Arbeit des ausgewählten Zufallsgenerators beobachten (ausgewählt ist hier ZZG-016). Der aktuelle Rechenschritt ist her-

**Tabelle 25.1.** Spielstand nach drei Runden: Mensch – Maschine 2:1

| Runde | Mensch | Maschine | PMe | PMa |
|---|---|---|---|---|
| 1 | Schere | Stein | 0 | 1 |
| 2 | Stein | Schere | 1 | 1 |
| 3 | Stein | Schere | 2 | 1 |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |

vorgehoben; aber auch die zukünftigen Werte sind bereits sichtbar! Man kann also feststellen, was die Maschine als nächstes tun wird (ein kleiner „Cheat"!).

## Monte-Carlo-Simulation: Flächenberechnung mit „Zufallsregen"

Ein wichtiger Anwendungsbereich von Zufallszahlen ist die so genannte Monte-Carlo-Simulation, die nach dem Spielcasino in Monte Carlo benannt ist. Monte-Carlo-Simulation kann z. B. verwendet werden, um eine Flächenbestimmung von unregelmäßigen geometrischen Figuren durch „Zufallsregen" durchzuführen. Von Zufallsregen spricht man, wenn viele zweidimensionale Zufallspunkte $(x, y)$ auf eine Ebene treffen. Ein Zufallspunkt in der Ebene wird durch ein $(x, y)$-Paar von zufälligen Koordinatenwerten bestimmt, wobei der $x$-Wert und der $y$-Wert jeweils durch eine Zufallszahl aus dem Intervall $[0, 1]$ festgelegt werden. Zu diesem Zweck werden aus dem Intervall $\{0, 1, \ldots, m-1\}$ gezogene Zufallszahlen so transformiert, dass reellwertige Zufallszahlen zwischen 0 und 1 entstehen. Dies geschieht durch die Transformation $x := x/(m-1)$.

Legt man eine beliebige Fläche in ein Quadrat mit Kantenlänge 1 (wie in Abb. 25.6) und wirft Zufallspunkte in dieses Quadrat, dann fällt ein Teil der Punkte auf diese Fläche und der Rest daneben.

Eine Schätzung für den Flächeninhalt erhält man durch die Berechnung des Quotienten $F$ = Trefferanzahl/Punkteanzahl. Um eine gute Schätzung zu bekommen, muss man Millionen oder sogar Milliarden von Punkten werfen. Eine Durchführung dieses Algorithmus von Hand wird man deswegen niemandem zumuten, aber für einen programmierten Algorithmus auf einem Rechner ist es natürlich kein Problem, einmal schnell 1 Million Zufallspunkte in ein Quadrat zu werfen!

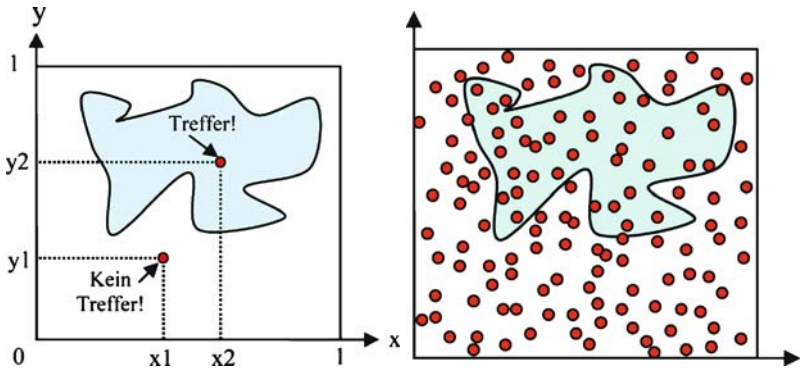



**Abb. 25.6.** Zwei Zufallspunkte $(x_1, y_1)$ und $(x_2, y_2)$ (*links*), Zufallsregen: Zähle die Treffer auf der Fläche (*rechts*)

Als Beispiel zur praktischen Verwendung der Monte-Carlo-Simulation betrachten wir ein Verfahren zur Bestimmung der berühmten Kreiszahl $\pi = 3{,}14159\ldots$ Eine näherungsweise Bestimmung der Zahl kann durch das „Werfen“ von zufälligen Punkten $(x, y)$ in den Einheitskreis durchgeführt werden. Wir betrachten ein Quadrat mit Seitenlänge 1 (Fläche = 1), wobei im 1. Quadranten ein Viertelkreis eingebettet ist. Da der Kreis den Radius $r = 1$ hat, ist seine Fläche $F = r^2 \cdot \pi = \pi$ und die Fläche des Viertelkreises ist $\frac{\pi}{4}$.

Sei $T$ die Zahl der Treffer im Viertelkreis und $N$ die Gesamtzahl der Würfe in dem Quadrat, dann können wir $\pi$ näherungsweise berechnen durch $\pi \approx 4 \cdot \frac{T}{N}$. In dem Bild sind 130 Zufallspunkte dargestellt, davon fallen 102 in den Viertelkreis, d.h., wir rechnen $\pi \approx 4 \cdot \frac{102}{130} \approx 3{,}1384$. Das ist noch kein

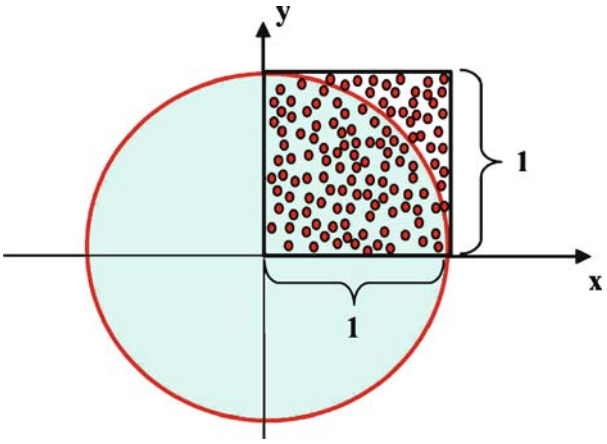



**Abb. 25.7.** Wie viele Zufallspunkte fallen in den Kreis?

sehr guter Wert für $\pi$, aber mit 100000 oder 1 Million oder sogar 1 Milliarde Punkten sollte das Ergebnis wesentlich genauer werden. Dieses rein mit Worten beschriebene Verfahren ist in folgendem Algorithmus ZUFALLSREGEN zusammengefasst.

Der Algorithmus ZUFALLSREGEN berechnet die Kreiszahl $\pi$.
Hierbei setzen wir ein Einheitsquadrat (Kantenlänge = 1) voraus, welches den Flächeninhalt 1 besitzt.

```
1   procedure ZUFALLSREGEN (n)
2   begin
3       a := 1103515245; c := 12345; m := 4294967296;      # Parameter
4       z := 1; treffer := 0;      # Startwerte
5       for i := 1 to n do
6           z := (a · z + c) mod m;
7           x := z/(m − 1);
8           z := (a · z + c) mod m;
9           y := z/(m − 1);
10          if INFLAECHE(x,y) then
11              treffer := treffer + 1
12          endif
13      endfor
14      return 4 · treffer/n
15  end
```

*Anmerkung*: Die Funktion INFLAECHE prüft, ob sich der Punkt mit den als Parameter übergebenen Koordinaten $(x, y)$ in der Fläche befindet. Da es sich um eine Kreisfläche handelt, wird in der Funktion INFLAECHE die Kreisgleichung $x^2 + y^2 = r^2$ verwendet. Ein Punkt $(x, y)$ liegt genau dann im Einheitskreis, wenn gilt: $x^2 + y^2 \leq 1$.

Es gibt zahlreiche Anwendungen der Monte-Carlo-Simulation in den Ingenieur- und Naturwissenschaften. In der Informatik hat sich aus den Techniken zur Monte-Carlo-Simulation das Gebiet der so genannten zufallsgesteuerten Algorithmen entwickelt, die für manche Aufgabenstellungen wesentlich schneller und auch einfacher sind als konventionelle Algorithmen.

## Zum Weiterlesen

1. Das hier vorgestellte Verfahren zur Berechnung von Pseudo-Zufallszahlen unter Verwendung der Modulo-Operation ist auch grundlegend für viele andere Bereiche der Informatik, z. B. für die in diesem Buch behandelten Techniken „Public-Key-Kryptographie“ (Kap. 16), „Fingerprinting“ (Kap. 19) und „One-Time-Pad“ (Kap. 15).
2. Mehr zum Thema Zufallszahlen findet man bei Wikipedia unter:
   `http://de.wikipedia.org/wiki/Pseudozufall`

3. Eine Einführung in Methoden zum Entwurf von zufallsgesteuerten Systemen für Einsteiger findet sich in dem Lehrbuch „Randomisierte Algorithmen“ von J. Hromkovič.[2]
4. Pseudo-Zufallszahlen sind auch von großer Bedeutung für die stochastische Simulation von komplexen Informatiksystemen, wie z. B. bei der Planung von Rechner- und Datennetzen im Umfeld von Internet, WorldWideWeb und Mobilkommunikation.
5. Weitere Anwendungsmöglichkeiten für die Verwendung von (Pseudo-)Zufallszahlen finden sich im Bereich der genetischen Algorithmen und evolutionären Systeme. Als Einstieg in diese so genannten naturimitierenden Modellierungsparadigmen empfehlen wir das in diesem Buch beschriebene Verfahren des „Simulated Annealing“ (Kap. 43).

[2] J. Hromkovič: *Randomisierte Algorithmen – Methoden zum Entwurf von zufallsgesteuerten Systemen für Einsteiger*. Teubner, Stuttgart Leipzig Wiesbaden, 2004.

# 26

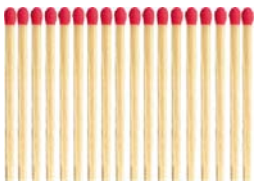
# Gewinnstrategie für ein Streichholzspiel

Jochen Könemann

University of Waterloo, Kanada

Ich bin gerade ein bisschen verärgert und, zugegeben, auch etwas verwirrt. Gestern Abend kommt mein Bruder freudestrahlend auf mich zu – und das alleine ist schon mal kein gutes Zeichen –, um mir von einem *sehr interessanten* neuen Intelligenztest zu erzählen. Natürlich erwarte ich nach dieser Ankündigung nichts Gutes. Aber irgendwie bin auch ein bisschen neugierig, von was er mir da berichten will. „Dann lass mal hören!“

„Prima!“, sagt er und kramt in seiner Hosentasche, nur um Sekunden später eine kleine Streichholzschachtel hervorzuzaubern, die er umgehend vor mir entleert. Jetzt liegen 18 Streichhölzer zwischen mir und ihm auf dem Tisch.



Die Spielregeln sind nun denkbar einfach: Der erste Spieler nimmt 1, 2 oder 3 der auf dem Tisch liegenden Streichhölzer vom Tisch, danach nimmt der zweite Spieler entweder 1, 2 oder 3 der verbleibenden Hölzer usw. Verloren hat der Spieler, der das letzte Streichholz vom Tisch nimmt.

„Verstanden?“, fragt er mich. Er muss auch wirklich immer den großen Bruder raushängen lassen. „Na klar! Ist ja nun auch wirklich kein sehr kompliziertes Spiel.“ Er grinst und sagt: „Na schön. Dann kann's ja los gehen und ich fang' auch gleich mal an. Bin ja schließlich der Ältere von uns beiden.“ Mit diesen Worten nimmt er ein Streichholz vom Tisch; es verbleiben somit 17 Hölzer.

Klasse, also verlieren kann ich in diesem Zug schon mal nicht, denn egal ob ich 1, 2 oder 3 Streichhölzer wegnehme, es verbleiben auf jeden Fall mindestens 14 auf dem Tisch. Also schnappe ich mir 2 der Hölzer und mein Bruder ist am Zug mit 15 verbleibenden Streichhölzern. Der grinst immer noch (warum nur?) und lässt 2 weitere Hölzer verschwinden, womit 13 auf dem Tisch verbleiben. Die linke Spalte der folgenden Tabelle zeigt die augenblickliche Spielsituation: die Anzahl der verbleibenden Streichhölzer. Die rechte Spalte enthält die darauf folgenden Spielzüge.

| Anzahl verbleibender Streichhölzer | Spielzug |
|---|---|
|  (13) | Ich nehme 3 Hölzer. |
|  (10) | Mein Bruder nimmt 1 Holz. |
|  (9) | Ich nehme 2 Hölzer. |
|  (7) | Mein Bruder nimmt 2 Hölzer. |
|  (5) | Ich nehme 1 Holz. |
|  (4) | Mein Bruder nimmt 3 Hölzer. |

Nach dem letzten Zug meines Bruders verbleibt ein Streichholz auf dem Tisch, was ich nach der Spielregeln nehmen muss. Somit habe ich verloren! „Das war natürlich nur Glück!“, sage ich und fordere eine Revanche. Auch das nächste Spiel und die beiden danach gehen an meinen Bruder und so langsam stellt sich Frustration bei mir ein. Mein Bruder scheint mir in diesem Spiel wirklich überlegen zu sein. Aber wie macht er das?

## Lernen anhand kleiner Beispiele

Um das Spiel ein wenig besser zu verstehen, sehe ich mir mal ein paar kleine Beispiele an. Ich weiß, dass ich verloren habe, wenn ich am Zug bin und nur noch ein Holz auf dem Tisch liegt. Aber was soll ich machen, wenn statt einem Holz, zwei vor mir liegen? Na ja, in diesem Fall nehme ich eins der beiden Hölzer und dann ist mein Bruder dran, der das letzte Holz nehmen muss und somit verliert! Das heißt, dass ich in dem Streichholzspiel meines Bruders eine *Gewinnstrategie* habe, wenn zu irgendeiner Zeit 2 Streichhölzer auf dem Tisch liegen und ich am Zug bin. Jetzt ist es nicht schwer, sich zu überlegen, dass ich auch eine Gewinnstrategie habe, wenn 3 oder 4 Hölzer auf dem Tisch liegen und ich am Zug bin. Im ersten Fall nehme ich 2 und im zweiten Fall 3 Hölzer vom Tisch und überlasse meinem Bruder ein einziges Holz.

Halten wir unsere bisher gewonnen Erkenntnisse fest: Wir wissen jetzt also, dass, wenn einer der beiden Spieler des Streichholzspiels 2, 3 oder 4 Hölzer vor sich liegen hat und am Zug ist, der Spieler durch geschicktes Handeln das Spiel für sich entscheiden kann.

| $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $\mathrm{GS}_i$ | Nein | Ja | Ja | Ja |

Die Tabelle links hat eine Spalte für jede der bisher analysierten Spielsituationen. Der Eintrag in der unteren Zeile von Spalte $i$ (den wir mit $\mathrm{GS}_i$ bezeichnen) zeigt an, ob der sich am Zug befindende Spieler eine Gewinnstrategie hat, wenn $i$ Streichhölzer auf dem Tisch liegen.

Soweit so gut, aber was ist, wenn 5 Hölzer auf dem Tisch liegen? Die Antwort auf diese Frage erscheint nicht mehr ganz so einfach, interessiert mich aber sehr, da dies die vorletzte Spielsituation im ersten Spiel gegen meinen Bruder war. Die Spielregeln zwingen mich in dieser Situation mindestens ein und höchstens 3 Streichhölzer vom Tisch zu nehmen. Das heißt, mein Bruder hat nach meinem Zug entweder 2, 3 oder 4 Streichhölzer vor sich. In jeder dieser 3 Situationen hat mein Bruder eine Gewinnstrategie, wie ich von meinen vorherigen Argumenten weiß! Ich kann also in dieser Ausgangssituation nicht gewinnen, wenn mein Bruder sich clever verhält. Allgemeiner kann der sich am Zug befindende Spieler aus einer Anfangssituation mit 5 Streichhölzern nicht gewinnen, wenn der Gegenspieler geschickt spielt und somit $\mathrm{GS}_5 = \text{Nein}$.

Wenn sich 6 Streichhölzer auf dem Tisch befinden, dann kann ich entweder 1, 2 oder 3 Streichhölzer vom Tisch nehmen und meinem Bruder 3, 4 oder 5 Hölzer zurücklassen. Aus der obigen Tabelle weiß ich, dass mein Bruder aus einer Ausgangssituation mit 3 oder 4 Hölzern einen Gewinn erzwingen kann. Wenn ich ihm allerdings 5 Streichhölzer zurücklasse, dann hat er keine Gewinnstrategie ($\mathrm{GS}_5 = \text{Nein}$) und somit kann er nicht gewinnen, wenn ich mich nach seinem Spielzug geschickt verhalte. Das heißt, dass ich eine Gewinnstrategie habe, wenn ich am Zug bin und 6 Streichhölzer vor mir liegen: Ich nehme ein Holz vom Tisch! Somit haben wir $\mathrm{GS}_6 = \text{Ja}$.

## Ein Algorithmus zur Berechnung einer Gewinnstrategie

Die obigen Berechnungen kann ich jetzt problemlos fortführen. Nehmen wir beispielsweise an, ich hätte für alle Ausgangssituationen mit $i \in \{1, \ldots, 14\}$ Streichhölzern bereits bestimmt, ob der sich am Zug befindende Spieler aus dieser Lage einen Sieg erzwingen kann. Wenn nun 15 Hölzer vor mir liegen, kann ich entweder 1, 2 oder 3 vom Tisch entfernen und somit 12, 13 oder 14 Hölzer zurücklassen. Wenn mein Bruder in all diesen Situationen eine Gewinnstrategie hat ($\mathrm{GS}_{12} = \mathrm{GS}_{13} = \mathrm{GS}_{14} = \text{Ja}$) und sich clever verhält, dann habe ich verloren und somit $\mathrm{GS}_{15} = \text{Nein}$. Andererseits, wenn mein Bruder in mindestens einer dieser Situationen keine Gewinnstrategie hat, dann kann ich durch geschicktes Verhalten einen Sieg herbeiführen ($\mathrm{GS}_{15} = \text{Ja}$).

Es ergibt sich der folgende Algorithmus zur Berechnung von $GS_1$ bis $GS_x$:

```
GEWINNSTRATEGIE(x)
 1  GS_1=Nein, GS_2=Ja, GS_3=Ja
 2  i:=3
 3  while i < x do
 4      i:=i+1
 5      if GS_{i-3}=GS_{i-2}=GS_{i-1}=Ja then
 6          GS_i=Nein
 7      else
 8          GS_i=Ja
 9      endif
10  endwhile
```

Der obige Algorithmus kann nun dazu verwendet werden, um $GS_1$ bis $GS_{18}$ zu berechnen. Das Ergebnis des Aufrufs GEWINNSTRATEGIE(18) ist in der folgenden Tabelle zusammengefasst, wobei wir Nein mit N und Ja mit J abkürzen:

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $GS_i$ | N | J | J | J | N | J | J | J | N | J | J | J | N | J | J | J | N | J |

Sehr schön! Jetzt weiß ich, dass mein Bruder aus der Startposition mit 18 Hölzern als Erstziehender eine Gewinnstrategie hat. Aber wie sieht diese Strategie aus?

Sehen wir uns noch einmal das erste Spiel zwischen mir und meinem Bruder an: Mein Bruder beginnt das Spiel und es liegen 18 Streichhölzer vor ihm. Aus der obigen Tabelle entnehmen wir, dass $GS_{18}$ = Ja ist und somit hat mein Bruder eine Gewinnstrategie gegen mich, so lange er alles richtig macht. Wie soll nun sein erster Zug aussehen? Er hat die Wahl 1, 2 oder 3 Streichhölzer vom Tisch zu nehmen und mir somit 15, 16 oder 17 Hölzer zurückzulassen. Was soll er tun? Ein weiterer Blick auf die obige Tabelle verrät es uns: Wenn ich am Zug bin und 15 Hölzer vor mir liegen, dann habe ich eine Gewinnstrategie. Das möchte mein Bruder natürlich vermeiden. Also wird er sich wohl nicht dafür entscheiden, 3 Hölzer vom Tisch zu nehmen. Ganz ähnlich verhält es sich, wenn er 2 Hölzer vom Tisch nimmt: Mit 16 verbleibenden Streichhölzern habe ich laut Tabelle eine Gewinnstrategie! Die einzig verbleibende Wahl ist, ein Holz vom Tisch zu entfernen. Und tatsächlich: Aus der Ausgangssituation mit 17 verbleibenden Hölzern, verliere ich laut Tabelle, wenn mein cleverer Bruder alles richtig macht.

„Was habe ich noch gleich gemacht, nachdem mein Bruder ein Holz vom Tisch genommen hat?“, überlege ich. „Richtig! Ich habe mir 2 Hölzer genommen und 15 Hölzer zurückgelassen.“ Die Tabelle verrät, dass mein Bruder aus dieser Lage gewinnen kann. Durch das Entfernen von 1, 2 oder 3 Hölzern kann er mir nun 12, 13 oder 14 Hölzer zurücklassen. Ein Blick auf die Tabelle zeigt, dass einzig und allein das Entfernen von 2 Hölzern eine Verlustsituation für

mich erzwingt. Und das ist auch genau, was mein Bruder in seinem nächsten Zug gemacht hat. „Von wegen Intelligenztest! Jetzt weiß ich, wie mein Bruder es angestellt hat, immer gegen mich zu gewinnen."

### Die Laufzeit des Algorithmus

Wie viele Elementaroperationen (Vergleiche und Zuweisungen) benötigt der Algorithmus zur Berechnung von $\mathrm{GS}_x$? Die innere **while**-Schleife wird $x-3$ mal ausgeführt. Jede dieser Iterationen greift dabei auf drei zuvor berechnete Werte $\mathrm{GS}_{i-3}$, $\mathrm{GS}_{i-2}$ und $\mathrm{GS}_{i-1}$ zurück, um $\mathrm{GS}_i$ zu berechnen. Somit genügen drei Vergleiche und zwei Zuweisungen für Schritte $4-9$ des Algorithmus und daher ist die Gesamtzahl der Operationen

$$(x-3)\cdot(3+2)+4=5x-11.$$

Die Laufzeit ist also proportional zu der Zahl der Streichhölzer. Das ist sicherlich kein Problem, wenn wir das Spiel tatsächlich spielen und die Streichhölzer auf den Tisch legen. Allerdings kann man mit wenigen Ziffern schon sehr große Zahlen beschreiben und ein Computer kann mit ihren Bits auch sehr schnell umgehen. Wenn wir beispielsweise wissen wollen, wer bei $x = 9876543210$ Streichhölzern gewinnt, dann können wir diese Problemstellung mit 10 Ziffern beschreiben. In diesem Fall ist die Anzahl der Schritte unseres Algorithmus (etwa $5\cdot 9876543210$) schon recht groß.

Besser wäre es, wenn die Anzahl der Schritte unseres Algorithmus proportional zur *Anzahl* der Ziffern der Zahldarstellung wäre und nicht proportional zur Zahl selbst. Die Anzahl Ziffern in der Eingabe $x$ ist abhängig von der gewählten Darstellung. Computer verwenden üblicherweise die sogenannte *Binärdarstellung*, in welcher $x$ einem String

$$a_k a_{k-1},\ldots,a_0$$

von $0,1$-Bits entspricht, so dass

$$x=\sum_{i=0}^{k}2^i a_i$$

gilt. Die Länge der Eingabe unseres Algorithmus ist dann $(k+1)$ und nicht $x$. Wie groß ist nun $k$ im Verhältnis zu $x$? Selbstverständlich gilt

$$2^{\lceil\log_2 x\rceil+1}\geq 2^{\log_2 x+1}=2\cdot x,$$

und daher folgern wir, dass $a_i=0$ für all $i>\lceil\log_2 x\rceil$. Das heißt, dass die Binärdarstellung von $x$ allerhöchstens $\lceil\log_2 x\rceil+1$ Bits benötigt. Die größte Zahl $x$, die sich mit $k+1$ Bits darstellen lässt, ist

$$\sum_{i=0}^{k}2^i=2^0+2^1+\ldots+2^k.$$

Dies ist eine *geometrische Reihe* und man kann zeigen, dass ihr Wert exakt $2^{k+1}-1$ ist (ein Verweis auf einen Artikel mit mehr Informationen zu diesem Thema findet sich am Ende dieses Kapitels). Substituiert man $k = \lfloor \log_2 x \rfloor - 1$ so ergibt sich

$$\sum_{i=0}^{\lfloor \log_2 x \rfloor - 1} 2^i = 2^{\lfloor \log_2 x \rfloor} - 1 \leq 2^{\log_2 x} - 1 = x - 1$$

und daher folgt, dass die Binärdarstellung von $x$ zumindest $\lfloor \log_2 x \rfloor + 1$ Bits benötigt.

Zusammenfassend halten wir fest, dass die Länge $k+1$ der Binärdarstellung von $x$ im Intervall $[\lfloor \log_2 x \rfloor + 1, \lceil \log_2 x \rceil + 1]$ liegt. Die Anzahl der Operationen unseres Algorithmus ist somit

$$5x - 11 \geq 5 \cdot 2^k - 11$$

und dies ist *exponentiell* in der Eingabelänge $k$. Algorithmen, deren Laufzeit zwar proportional zu den Eingabewerten sind, sich aber exponentiell zur Eingabelänge verhalten, werden oft auch als *pseudo-polynomiell* bezeichnet.

Unser Algorithmus kann also nicht wirklich als effizient bezeichnet werden und es ist daher unwahrscheinlich, dass mein Bruder ihn benutzt hat, um seine Gewinnstrategie zu berechnen. In der Tat verrät er mir, dass es nicht schwer sei, zu beweisen, dass $\mathrm{GS}_x = \text{Ja}$ ist, wenn der Rest von $x$ geteilt durch 4 ungleich 1 ist. Mit anderen Worten $\mathrm{GS}_x = \text{Nein}$, wenn der Rest von $x$ durch 4 genau 1 ist. Wenn ich es also einmal in eine Gewinnsituation geschafft habe, dann geht's total einfach weiter: Mein Gegenspieler nimmt $y$ Hölzer, ich nehme $4 - y$, so dass als Summe 4 herauskommt.

Diese handliche Formel deckt sich mit der obigen Tabelle und erklärt auch, warum mein Bruder keine Tabelle bzw. keinen Computer gebraucht hat, um gegen mich zu gewinnen.

## Erweiterungen und Hintergrundinformationen

Das vorgestellte Spiel lässt sich beinahe beliebig erweitern und verkomplizieren. Eine Variante, die als *Nim* bekannt ist, funktioniert wie folgt: Anstatt einer Reihe liegen mehrere Reihen von Streichhölzern auf dem Tisch. Wieder spielen zwei Spieler, die sich abwechseln. Der sich am Zug befindende Spieler wählt zuerst eine der Reihen mit verbleibenden Hölzern aus und entnimmt ihr mindestens ein und beliebig viele Streichhölzer. Wie zuvor verliert der Spieler, der das letzte Streichholz vom Tisch nimmt. Dieses Spiel kann in ganz ähnlicher Weise analysiert werden, und auch für dieses Spiel existiert eine geschlossene Formel, die die Gewinnstrategie beschreibt.

Das Spiel Nim ist sehr alt und stammt wahrscheinlich aus China (es ähnelt dem dort existierenden Spiel *Tsyanshidzi*). In Europa ist das Spiel wohl zuerst

im 16. Jahrhundert aufgetaucht. Seinen Namen *Nim* erhielt das Spiel von Charles Bouton, der im Jahr 1901 auch zuerst die vollständige Analyse dieses Spiels veröffentlichte.

Unser Algorithmus zur Berechnung von $\mathrm{GS}_x$ ist ein einfaches Beispiel für ein Verfahren, das man als *dynamische Programmierung* bezeichnet. Dieses Verfahren wurde in den 1940er Jahren von dem Amerikaner Richard Bellman entwickelt und ermöglicht das Lösen von Optimierungsproblemen, die sich in gleichartige Teilprobleme zerlegen lassen. In unserem Streichholzspiel lässt sich beispielsweise die Frage, ob ein Spieler aus einer Ausgangssituation mit $x$ Streichhölzern eine Gewinnstrategie hat, auf die Antwort auf diese Frage aus einer Ausgangslage mit $x-3$, $x-2$ und $x-1$ Streichhölzern zurückführen.

Ein weitaus komplexeres Beispiel zur dynamischen Programmierung findet sich in Kap. 32, wo mit Hilfe dieser Methode die *Mutationsdistanz* zweier genetischen Strings berechnet wird. Der *Alphabeta*-Algorithmus zur Analyse von Spielbäumen aus Kap. 28 basiert auf dem so genannten *Minimax* Prinzip, was im Wesentlichen wieder nichts anderes ist als dynamische Programmierung.

## Zum Weiterlesen

1. `http://de.wikipedia.org/wiki/Nim-Spiel`

   Ein Artikel zu einer Variante des Nim Spiels, in der derjenige Spieler gewinnt, der den Tisch leer räumt.

2. `http://de.wikipedia.org/wiki/Dynamische_Programmierung`

   Dynamische Programmierung in der Wikipedia.

3. `http://de.wikipedia.org/wiki/Geometrische_Reihe`

   Ein Artikel zu geometrischen Reihen.

4. D. P. Bertsekas: *Dynamic Programming and Optimal Control Vol. 1&2*. Athena Scientific, 3. Auflage, 2005.

   Ein sehr umfassendes Lehrbuch zur dynamischen Programmierung in englischer Sprache.

# 27

# Turnier- und Sportligaplanung

Sigrid Knust

Universität Osnabrück

Die neu gegründete Tischtennisabteilung des TV Schmetterhausen möchte in der kommenden Saison in einer Liga am Punktspielbetrieb teilnehmen und eine Mannschaft mit 6 Spielern melden. Die Mannschaft muss nach Spielstärke aufgestellt sein, d.h., der beste Spieler spielt an Position 1, der zweitbeste an Position 2 usw. Um die Spieler nach Spielstärke ordnen zu können, beschließt der Abteilungsleiter Anton Leiter, ein Turnier durchzuführen. Die 6 ausgewählten Spieler sollen in den nächsten Wochen nach dem System „jeder gegen jeden" gegeneinander spielen, danach soll die Mannschaft gemäß der dort ermittelten Rangfolge aufgestellt werden. An jedem Trainingsabend soll jeder Spieler genau ein Spiel austragen.

Die erste Frage, die sich Anton in diesem Zusammenhang stellt, ist die Frage, wie viele Abende für das Turnier benötigt werden. Jeder der 6 Spieler muss genau einmal gegen jeden der 5 anderen Spieler antreten, d.h., es sind insgesamt $\frac{6 \cdot 5}{2} = 15$ Spiele zu absolvieren (durch 2 muss geteilt werden, da das Spiel $i$ gegen $j$ sowohl für Spieler $i$ als auch für Spieler $j$ gezählt wird). Anton rechnet: Wenn jeder der 6 Spieler an jedem Trainingsabend genau ein Spiel austrägt, finden 3 Spiele pro Abend statt, d.h., es werden somit $\frac{15}{3} = 5$ Abende benötigt, um alle Spiele durchzuführen.

Hochmotiviert legen die Spieler der Abteilung los, wobei sich an jedem Abend jeder Spieler einen Gegner sucht, gegen den er noch nicht gespielt hat. Nach drei Abenden sind folgende Begegnungen absolviert:

| 1. Abend | 2. Abend | 3. Abend |
|---|---|---|
| 1-2 | 1-3 | 1-4 |
| 3-5 | 2-6 | 2-5 |
| 4-6 | 4-5 | 3-6 |

Es ist leicht zu überprüfen, dass die restlichen 6 Spielpaarungen 1-5, 1-6, 5-6, 2-3, 2-4, 3-4 nicht an zwei weiteren Abenden beendet werden können (wenn 1-5 spielt, kann dazu weder das Spiel 1-6 noch das Spiel 5-6 parallel ausgetragen werden, da jeder Spieler nur einmal pro Abend spielen soll). Das Turnier lässt sich nur mit drei weiteren Abenden beenden:

| 4. Abend | 5. Abend | 6. Abend |
|---|---|---|
| 1-5 | 1-6 | 5-6 |
| 2-3 | 2-4 | 3-4 |

Für das Turnier benötigt die Abteilung somit einen Abend länger als ursprünglich geplant, außerdem müssen an jedem der letzten Abende zwei Spieler aussetzen. Da man hinterher bekanntlich immer schlauer ist, fragt sich Anton, ob das so sein muss oder ob die Abteilung mit einem anderen Spielplan ihr Turnier auch schon nach 5 Abenden hätte beenden können. Bei der abendlichen Sportschau stellt Anton fest, dass die Fußballbundesliga eigentlich ein ähnliches Problem hat. Dort müssen 18 Mannschaften in einer Hin- und Rückserie jeweils genau einmal gegen jede andere Mannschaft spielen, wobei in jeder Runde (Wochenende) jede Mannschaft genau ein Spiel austrägt. Anton denkt zurück und erinnert sich, dass in den letzten Jahren jede Saison in $2 \cdot 17 = 34$ Wochen beendet worden konnte und nie eine Mannschaft in einer Runde aussetzen musste. Er fragt sich, ob es immer einen solchen Spielplan gibt oder ob evtl. die Zahl 18 günstiger als die Zahl 6 ist.

Verlassen wir jetzt einmal Anton, den TV Schmetterhausen sowie die Fußballbundesliga und betrachten unser Problem etwas allgemeiner: Gegeben sind eine gerade Anzahl $n$ von Mannschaften (oder Spielern) und $n-1$ Runden (Spieltage). Gesucht ist ein Spielplan, so dass jede Mannschaft genau einmal gegen jede andere spielt und jede Mannschaft in jeder Runde genau ein Spiel austrägt. Insbesondere stellt sich die Frage, ob es für jede gerade Zahl $n$ einen solchen Spielplan gibt und wenn ja, wie man ihn konstruieren kann. Im Folgenden werden wir zeigen, dass für jedes gerade $n$ eine Lösung existiert (also sowohl für $n = 18$ als auch für $n = 6$, aber auch für $n = 100$ oder $n = 1024$) und einen Algorithmus vorstellen, der für jedes $n$ einen solchen Spielplan berechnet.

## Generierung von Spielplänen

Um einen Algorithmus zur Generierung von Spielplänen anschaulich beschreiben zu können, modellieren wir unser Problem zunächst mit Hilfe von so genannten Graphen, die generell in der Informatik eine wichtige Rolle spielen (vgl. auch die Kap. 29, 34, 35, 36 und 42). Ein Graph besteht aus einer Menge von Knoten und Kanten, wobei eine Kante jeweils zwei Knoten miteinander verbindet. Auf diese Weise lassen sich z.B. Straßennetzwerke modellieren, bei denen Straßen den Kanten und Kreuzungen den Knoten entsprechen.

Bei unserem Turnier- oder Sportligaplanungsproblem führen wir für jede Mannschaft einen Knoten ein, die Spiele entsprechen den Kanten. Für $n = 6$ Mannschaften erhält man den Graphen in Abb. 27.1.

Einen solchen Graphen nennt man auch vollständig, da jeder Knoten mit jedem anderen Knoten durch eine Kante verbunden ist (zur Erinnerung: jede Mannschaft soll gegen jede andere spielen). Um einen Spielplan zu erhalten,

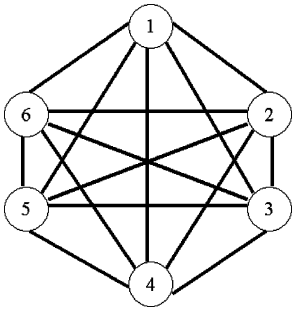



**Abb. 27.1.** Graph für 6 Mannschaften

färben wir nun die Kanten mit den Farben $1, 2, \ldots, n-1$, wobei jede Farbe eine Runde repräsentiert. Eine solche Färbung wollen wir zulässig nennen, wenn sie einem zulässigen Spielplan entspricht. Dazu müssen alle Kanten, die in den gleichen Knoten hineinführen, unterschiedlich gefärbt sein (sonst ist die Bedingung verletzt, dass jede Mannschaft in jeder Runde nur einmal spielt).

Für unseren Graphen mit $n = 6$ Knoten ist z.B. die in Abb. 27.2 dargestellte Kantenfärbung mit $n - 1 = 5$ Farben zulässig. Ein zugehöriger Spielplan ist neben dem Graphen dargestellt, wobei die Farbe „Rot" die erste Runde repräsentiert, die Farbe „Blau" die zweite Runde usw.

Es bleibt die Frage, wie man für jede gerade Zahl $n$ eine zulässige Kantenfärbung für den vollständigen Graphen mit $n$ Knoten finden kann. Betrachten wir in dem Beispiel einmal den Graphen, der entsteht, wenn wir Knoten 6 und alle 5 Kanten, die in ihn hineinführen, entfernen. Es ergibt sich ein Fünfeck, bei dem die 5 Kanten auf dem Rand (1-2, 2-3, 3-4, 4-5, 5-1) alle unterschiedlich gefärbt sind. Zeichnen wir das Fünfeck wie in Abb. 27.3 als regelmäßiges Fünfeck (d.h. die Innenwinkel an allen 5 Ecken sind gleich), fällt auf, dass jede Kante im Inneren des Fünfecks die gleiche Farbe wie die zugehörige parallele Kante auf dem Rand hat. Sind bei einer Kantenfärbung immer nur paralle-

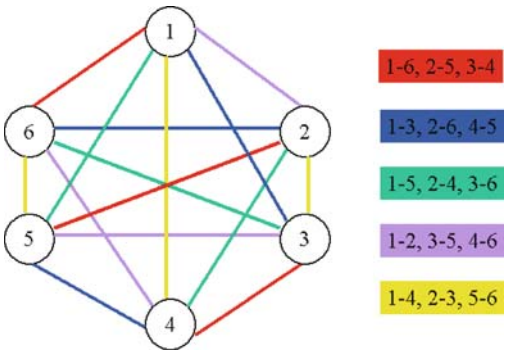



**Abb. 27.2.** Kantenfärbung für den Graphen mit 6 Mannschaften

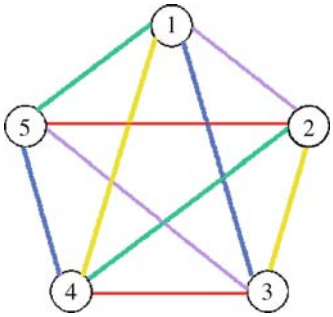



**Abb. 27.3.** Resultierendes Fünfeck

le Kanten (die ja keinen Knoten gemeinsam haben) mit der gleichen Farbe gefärbt, ist unsere obige Bedingung erfüllt, dass Kanten, die in den gleichen Knoten hineinführen, stets unterschiedlich gefärbt sind. Bei unserem Beispiel sehen wir weiterhin, dass bei jedem der 5 Knoten 4 Farben verbraucht sind und jeweils eine andere Farbe unbenutzt ist (die jeweils für die Kante zum Knoten 6 genutzt werden kann).

Diese Beobachtungen lassen sich zu einem Konstruktionsverfahren für eine Kantenfärbung für jeden Graphen mit einer geraden Anzahl $n$ von Knoten umsetzen. Der Ursprung des Verfahrens ist nicht eindeutig bekannt, aber man weiß, dass der englische Pastor und Hobby-Mathematiker Thomas P. Kirkman

**Algorithmus Kantenfärbung (geometrische Version):**

1. Bilde aus den Knoten $1, 2, \ldots, n-1$ ein regelmäßiges $(n-1)$-Eck und platziere den Knoten $n$ links oben neben dem $(n-1)$-Eck.
2. Verbinde den Knoten $n$ mit der „Spitze“ des $(n-1)$-Ecks.
3. Verbinde die übrigen Knoten jeweils mit dem gegenüberliegenden Knoten auf der gleichen Höhe im $(n-1)$-Eck.
4. Die eingefügten $\frac{n}{2}$ Kanten werden mit der ersten Farbe gefärbt (im Beispiel die Kanten 6-1, 5-2, 4-3).
5. Verschiebe die Knoten $1, \ldots, n-1$ des $(n-1)$-Ecks gegen den Uhrzeigersinn zyklisch um eine Position weiter (d.h., Knoten 2 geht auf den Platz von Knoten 1, Knoten 3 ersetzt den alten Knoten 2, ..., Knoten $n-1$ ersetzt den alten Knoten $n-2$ und Knoten 1 ersetzt den alten Knoten $n-1$). Der Knoten $n$ behält seinen Platz neben dem $(n-1)$-Eck und die in den Schritten 2 und 3 eingefügten Kanten behalten ihre Position im $(n-1)$-Eck.
6. Die neu resultierenden $\frac{n}{2}$ Kanten werden mit der zweiten Farbe gefärbt (im Beispiel die Kanten 6-2, 1-3, 5-4).
7. Die Schritte 5 und 6 des Verfahrens werden für die übrigen Farben $3, \ldots, n-1$ wiederholt.

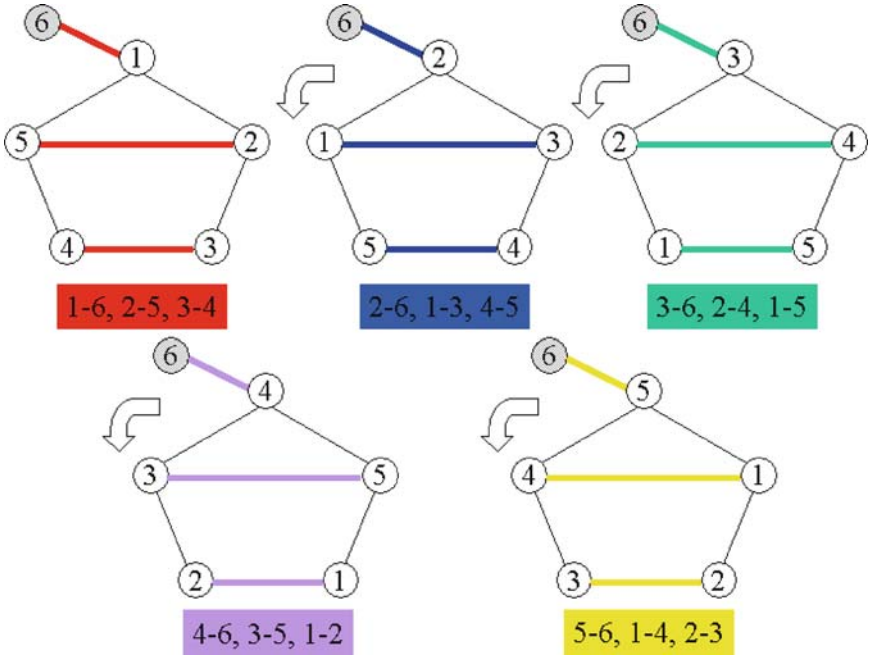



**Abb. 27.4.** Ablauf des Verfahrens

(1806–1895) bereits ähnliche Überlegungen angestellt hat. Auf Seite 278 ist der Algorithmus dargestellt, der in $n-1$ Iterationen jeweils eine Menge von parallelen Kanten mit einer anderen Farbe färbt (vgl. auch Abb. 27.4). Man kann zeigen, dass dieser Algorithmus für jede gerade Zahl $n$ funktioniert und eine zulässige Kantenfärbung liefert.

Um den obigen Algorithmus in ein Computerprogramm umzusetzen, wäre es relativ aufwändig, $(n-1)$-Ecke zu speichern und in jeder Iteration Knoten zyklisch zu verschieben. Wenn man die ganzzahlige Division mit Rest beherrscht (Modulo-Rechnung, vgl. Kap. 12), lässt sich der Algorithmus viel einfacher wie folgt aufschreiben:

Der Algorithmus FärbeKanten färbt alle Kanten des vollständigen Graphen mit gerader Knotenanzahl $n$, wobei $n-1$ Farben benutzt werden.

```
1   procedure FärbeKanten (n)
2   begin
3       for alle Farben i := 1 to n − 1 do
4           färbe die Kante [i, n] mit der Farbe i
5           for k := 1 to n/2 − 1
6               färbe alle Kanten [(i + k) mod (n − 1), (i − k) mod (n − 1)]
7               mit der Farbe i
8           endfor
9       endfor
10  end
```

In diesem Algorithmus bedeutet die Schreibweise $a \bmod b$, dass man die Zahl $a$ ganzzahlig durch die Zahl $b$ teilt und den Rest nimmt, der dabei entsteht. So ist z.B.

- $14 \bmod 4 = 2$, da $14 = 3 \cdot 4 + 2$,
- $9 \bmod 3 = 0$, da $9 = 3 \cdot 3 + 0$ und
- $-1 \bmod 5 = 4$, da $-1 = (-1) \cdot 5 + 4$.

Teilt man in Schritt 6 des Algorithmus ganzzahlig durch die Zahl $n-1$, so entstehen Reste aus der Menge $0, 1, \ldots, n-2$. Da unsere Knoten aber von $1, \ldots, n-1$ durchnummeriert sind (und nicht von $0, 1, \ldots, n-2$), wird der Rest 0 als $n-1$ interpretiert.

Für unser Beispiel erhält man in Schritt 6 für $i = 1$ die Kanten

- $[(1+1) \bmod 5, (1-1) \bmod 5] = [2, 5]$ für $k = 1$, und
- $[(1+2) \bmod 5, (1-2) \bmod 5] = [3, 4]$ für $k = 2$.

Die Werte für $i = 2, 3, 4, 5$ kann der Leser einmal selbst nachrechnen.

## Spielpläne mit Festlegung des Heimrechts

Wir kommen nun noch einmal zur Fußballbundesliga zurück. Im Gegensatz zu unserem Tischtennis-Turnier finden die Spiele nicht alle am gleichen Ort statt, sondern in den Stadien der jeweiligen Mannschaften. Wird das Spiel zwischen den Mannschaften $i$ und $j$ in der Hinserie im Stadion der Mannschaft $i$ ausgetragen, so wird in der Rückserie bei Mannschaft $j$ gespielt. Ein Spielplan besteht somit nicht nur aus den Spielpaarungen pro Runde (wer spielt gegen wen?), sondern es muss zusätzlich für jede Spielpaarung noch das Heimrecht festgelegt werden (wo findet das Spiel statt?).

Aus verschiedenen Gründen (z.B. Fairness, Attraktivität für die Zuschauer) sollten sich Heim- und Auswärtsspiele für jede Mannschaft möglichst abwechseln. Spielt eine Mannschaft zweimal hintereinander zu Hause oder zweimal hintereinander auswärts, so sagt man auch, dass die Mannschaft ein Break hat (die abwechselnde Folge von H- und A-Spielen ist unterbrochen). Sind Breaks unerwünscht, wäre natürlich ein Spielplan am besten, bei dem keine Mannschaft ein Break hat. Betrachtet man jedoch den Spielplan der Fußballbundesliga etwas genauer, erkennt man, dass dort in jeder Saison Breaks auftreten. Wieder können wir uns fragen, ob das so sein muss oder ob es bessere Pläne (ohne Breaks) gibt.

Man kann sich relativ leicht überlegen, dass es bei unseren Voraussetzungen keinen Spielplan ohne Breaks geben kann. Hätten alle Mannschaften kein Break, so müsste jede Mannschaft die Spielfolge HAHA...H oder AHAH...A haben. Zwei Mannschaften mit der gleichen Spielfolge (z.B. HAHA...H) können jedoch nie gegeneinander spielen (da sie beide immer entweder zu Hause oder auswärts spielen). Aus diesem Grund können höchstens zwei Mannschaften kein Break haben (eine Mannschaft mit der Folge HAHA...H, eine andere

mit AHAH...A). Daraus folgt, dass die übrigen n - 2 Mannschaften mindestens ein Break haben müssen, jeder Spielplan für eine Halbserie also mindestens $n-2$ Breaks enthält.

Man kann zeigen, dass es Spielpläne mit genau $n-2$ Breaks gibt und sie sich mit einer Erweiterung des obigen Algorithmus berechnen lassen. Als einzige Erweiterung muss man dabei in den Schritten 4 und 6 des Algorithmus FärbeKanten zusätzlich das Heimrecht wie folgt festlegen:

- Das Spiel $[i, n]$ ist ein Heimspiel für Mannschaft $i$, wenn $i$ gerade ist; sonst ist es ein Heimspiel für $n$.
- Das Spiel $[(i+k) \bmod (n-1), (i-k) \bmod (n-1)]$ ist ein Heimspiel für Mannschaft $(i+k) \bmod (n-1)$, wenn $k$ ungerade ist; sonst ist es ein Heimspiel für $(i-k) \bmod (n-1)$.

In unser Graphenmodell kann man Heim- und Auswärtsspiele integrieren, indem man den Kanten zusätzlich eine Richtung gibt. Bedeutet eine Kante $i \to j$, dass das Spiel zwischen $i$ und $j$ bei Mannschaft $j$ stattfindet, so erhält man für unser Beispiel mit $n = 6$ Mannschaften mit dem erweiterten Algorithmus den Plan aus Abb. 27.5 mit $n - 2 = 4$ Breaks (die Mannschaften 1 und 6 haben kein Break, die Mannschaften 2, 3, 4, 5 jeweils 1 Break).

Das geometrische Konstruktionsverfahren mit dem $(n-1)$-Eck funktioniert auch weiterhin, wenn man den Kanten dort ebenfalls eine Richtung gibt. Während die Kanten im Inneren des $(n-1)$-Ecks immer gleich gerichtet bleiben, wechselt die Orientierung der Kante zu dem äußeren Knoten $n$ in jeder Iteration (vgl. Abb. 27.6).

Zusammengefasst können wir feststellen: Zu jeder geraden Anzahl $n$ von Mannschaften gibt es einen Spielplan für $n-1$ Spieltage mit $n-2$ Breaks, der sich einfach durch den obigen Algorithmus berechnen lässt. Wer mag, kann sich einmal überlegen, was bei einer ungeraden Anzahl von Mannschaften passiert.

Zum Abschluss kommen wir noch einmal zur Fußballbundesliga zurück. Da dort eine Hin- und eine Rückserie gespielt werden, treten in jeder Halbserie

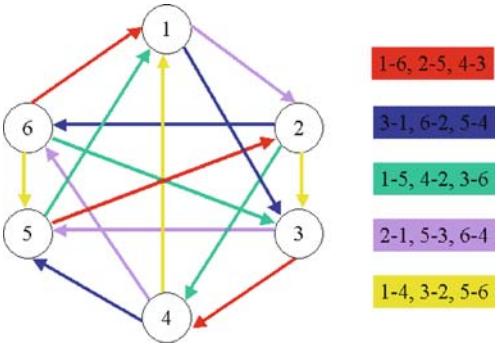



**Abb. 27.5.** Plan mit $n-2$ Breaks

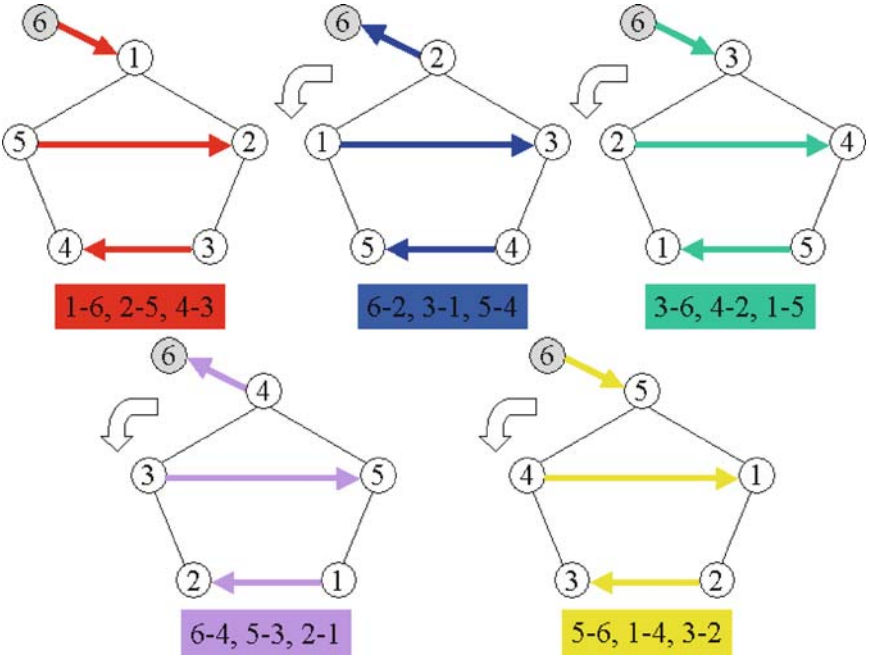



**Abb. 27.6.** Ablauf des erweiterten Verfahrens

mindestens $n-2$ Breaks auf. Beim System der deutschen Bundesliga finden die Spiele der Rückserie in der gleichen Reihenfolge wie die Spiele in der Hinserie statt (mit getauschtem Heimrecht). Für dieses System lässt sich zeigen, dass mindestens $3n - 6$ Breaks auftreten. Ein Plan mit $3n - 6$ Breaks (jeweils $n - 2$ in den beiden Halbserien und $n - 2$ beim Übergang von der Hin- zur Rückserie) lässt sich leicht mit der obigen Methode konstruieren. Durch eine kleine Modifikation lässt sich darüber hinaus erreichen, dass kein Team zwei aufeinander folgende Breaks hat.

In der Praxis ist die Planung einer Sportliga jedoch meist viel schwieriger, da zusätzliche Nebenbedingungen berücksichtigt werden müssen. So ist z.B. zu beachten, dass zwei Mannschaften, die das gleiche Stadion für Heimspiele nutzen, nicht gleichzeitig in einer Runde zu Hause spielen. Außerdem sollten aufgrund von Bahn- oder Polizeikapazitäten nicht zu viele Heimspiele in einer Region stattfinden. Des Weiteren kann es passieren, dass in manchen Runden ein Stadion nicht zur Verfügung steht, da dort bereits eine andere Veranstaltung (Konzert, Messe oder eine andere Sportveranstaltung) geplant ist. In diesem Fall muss die entsprechende Mannschaft in dieser Runde auswärts spielen. Die Medien und Zuschauer möchten eine Saison erleben, die lange spannend bleibt (d.h., Spitzenspiele sollten eher zum Ende der Saison stattfinden), und attraktive Spiele sollten möglichst gleichmäßig über die Saison verteilt sein.

In den meisten Ligen werden Spielpläne per Hand konstruiert. Ein Planer generiert mit der obigen Methode einen Spielplan für seine Ligagröße, wobei er zunächst die Zahlen $1, \ldots, n$ als Platzhalter für die Mannschaften einsetzt. In einem zweiten Schritt wird dann jeder Zahl eine konkrete Mannschaft zugeordnet (z.B. 1 = Werder Bremen, 2 = Hamburger SV, 3 = Bayern München usw.). Dabei wird versucht, möglichst viele zusätzliche Nebenbedingungen zu erfüllen (z.B. dass zwei Mannschaften, die das gleiche Stadion nutzen, Platzhaltern zugeordnet werden, die nie gleichzeitig ein Heimspiel haben).

Wir wollen uns einmal überlegen, wie viele verschiedene solcher Zuordnungen es bei einer Liga mit $n = 18$ Mannschaften gibt. Zunächst hat man für die erste Zahl 18 Mannschaften zur Auswahl, danach für die zweite Zahl nur noch 17 Möglichkeiten (da eine Mannschaft bereits festgelegt ist), dann 16 Möglichkeiten für die dritte Zahl usw. Insgesamt ergeben sich $18! = 18 \cdot 17 \cdot 16 \cdot \ldots \cdot 2 \cdot 1 = 6,4 \cdot 10^{15}$ Möglichkeiten. Unter der Voraussetzung, dass ein Computer eine Milliarde Lösungen pro Sekunde generieren könnte, müssten wir 74 Tage rechnen, um alle Möglichkeiten auszuprobieren. Diese riesige Zahl zeigt, dass ein menschlicher Planer selbst mit Computerunterstützung nur eine kleine Anzahl von möglichen Plänen ausprobieren kann. Ein weiterer Nachteil der gerade beschriebenen Methode besteht darin, dass nur ein möglicher Spielplan als Grundlage für die Zuordnung genommen wird. Es gibt eine Vielzahl von anderen Plänen, die nicht durch die obige Methode generiert werden können. Das bedeutet, dass mit dieser Methode evtl. gute Pläne (d.h. Pläne, die möglichst viele Nebenbedingungen erfüllen) nicht gefunden werden. Aus diesem Grund beschäftigt sich aktuelle Forschung im Bereich Sportligaplanung mit der Entwicklung von neuen Verfahren, mit denen möglichst gute Spielpläne in akzeptabler Rechenzeit berechnet werden können.

## Zum Weiterlesen

1. Kapitel 29 (Die Eulertour), 34 (Kürzeste Wege), 35 (Minimale aufspannende Bäume), 36 (Maximale Flüsse) und 42 (Das Travelling Salesman Problem)

   In diesen Kapiteln finden sich weitere Modelle und Algorithmen, die auf Graphen basieren.

2. Eric W. Weisstein: *Kirkman's Schoolgirl Problem*. From MathWorld – A Wolfram Web Resource:
   `http://mathworld.wolfram.com/KirkmansSchoolgirlProblem.html`

   Ein weiteres kombinatorisches Problem, bei dem man Lösungen durch gefärbte Graphen darstellen kann.

3. Thomas Bartsch, Andreas Drexl: *Fußballbundesliga-Spielpläne aus dem Computer*. OR News, Sonderausgabe 2006, 125–129.

   Dieser Artikel beschreibt, wie für die Österreichische Fußballbundesliga mit Hilfe von Computerverfahren Spielpläne erstellt wurden, die möglichst viele praktische Nebenbedingungen erfüllen.

4. Dieter Jungnickel: *Graphen, Netzwerke und Algorithmen*. BI-Verlag, Mannheim, 3. Auflage, 1994.

   Dieses Buch führt in die Grundbegriffe der Graphentheorie ein, insbesondere enthält es auch ein Kapitel zu Liga- und Turnierplänen.

5. Manfred Nitzsche: *Graphen für Einsteiger – Rund um das Haus vom Nikolaus*. Vieweg-Verlag, 2. Auflage, 2005.

   Eine insbesondere für Schüler und Nicht-Fachleute geschriebene Einführung in die Welt der Graphen.

# 28

# Der Alphabeta-Algorithmus für Spielbäume: Wie bringe ich meinen Computer zum Schachspielen?

Burkhard Monien, Ulf Lorenz und Daniel Warner

Universität Paderborn

Schon seit langer Zeit sind die Menschen von der Idee begeistert, Maschinen Gesellschaftsspiele spielen zu lassen. Unermüdlich sollen sie sein, aber auch spielstark, damit es nicht so schnell langweilig wird. Besonders das Schachspiel scheint hierbei magische Anziehungskraft zu besitzen.

Ende des 18. Jahrhundert war z.B. der Schachtürke von Wolfgang von Kempelen eine Sensation in Europa. Es handelte sich dabei um einen angeblich vollautomatischen mechanischen Schachspieler, dessen Funktionsgeheimnis lange Zeit gewahrt wurde. Jahrzehntelang soll er die Menschen seiner Zeit in Erstaunen versetzt haben, darunter prominente Namen wie Napoleon, Edgar Allen Poe und Kaiserin Maria Theresia. 1854 kam das gute Stück bei einem Brand „ums Leben“. Natürlich handelte es sich bei dieser Maschine um einen Trick: ein kleinwüchsiger Mensch konnte sich so geschickt im Inneren

**Abb. 28.1.** Der Schachtürke (Bildnachweis: Jan Braun/Heinz Nixdorf MuseumsForum)

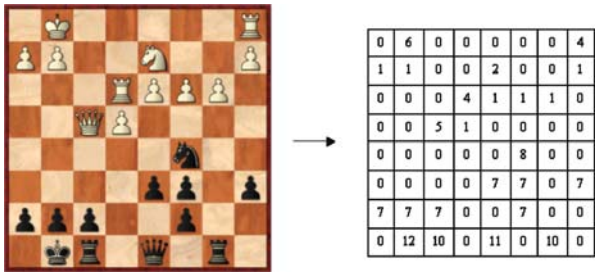
verstecken, dass es sehr schwierig war ihn zu entdecken, selbst wenn man den Spieltisch scheinbar aufmachte.

Mit der Erfindung von Computern gab es neue Hoffnung auf eine echte Schachmaschine. Viele Jahre hatte man es für nahezu unmöglich gehalten, dass ein Computer besser Schach spielen könnte als die besten Menschen. Aber heute wissen wir es besser. Mit Garry Kasparov musste sich Ende des 20. Jahrhunderts erstmals ein Schachweltmeister einem Schachprogramm, der IBM-Maschine Deep Blue, in einem Kampf über sechs Spiele knapp geschlagen geben. Mittlerweile gibt es mehrere Schachprogramme, die so stark oder sogar stärker spielen als die spielstärksten menschlichen Spieler. Sie haben klangvolle Namen wie Shredder, Fritz oder Rybka. Einen bisher einmaligen Vorsprung vor allen anderen Programmen und Menschen konnte sich das Schachprogramm Hydra erarbeiten, das bereits im Jahre 2005 den Supergroßmeister Michael Adams mit 5,5 zu 0,5 vernichtend schlug.

## Grundbausteine eines Schachprogramms

Die Frage ist nun, wie funktioniert eigentlich ein Schachprogramm?

Zunächst muss man natürlich ein Modell des Schachspiels in den Rechner hineinbringen. Das kann man z.B. tun, indem man das $8 \times 8$ Felder große Schachbrett als 2-dimensionales $8 \times 8$ Feld darstellt und den einzelnen Figuren Zahlen zuordnet. Ein leeres Feld entspricht einer 0, ein weißer Bauer einer 1 usw.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 6 | 0 | 0 | 0 | 0 | 0 | 4 |
| 1 | 1 | 0 | 0 | 2 | 0 | 0 | 1 |
| 0 | 0 | 0 | 4 | 1 | 1 | 1 | 0 |
| 0 | 0 | 5 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 |
| 0 | 0 | 0 | 0 | 7 | 7 | 0 | 7 |
| 7 | 7 | 7 | 0 | 0 | 7 | 0 | 0 |
| 0 | 12 | 10 | 0 | 11 | 0 | 10 | 0 |

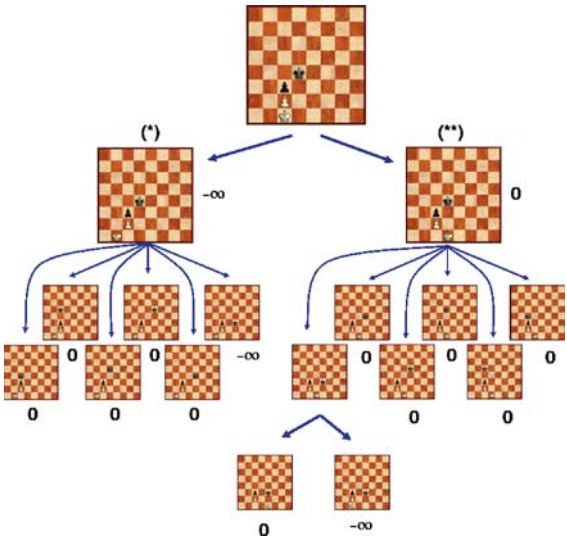
Das Schachprogramm lässt sich nun grob in drei „Baugruppen“ unterteilen: den Zuggenerator, die Bewertungsprozedur und den Suchalgorithmus. Der Zuggenerator erzeugt zu einer vorgegebenen Stellung eine Liste aller dort möglichen Züge. Die Bewertungsprozedur ordnet einer ihr vorgegebenen Schachstellung eine Zahl zu. Je größer diese Zahl ist, desto besser steht Weiß, negative Zahlen geben einen Vorteil für Schwarz wieder.

Die Abbildung unten zeigt, wie eine Stellungsbewertung aufgebaut sein kann. Typische Werte für die so genannten „Materialwerte“ sind +100 Punkte für einen weißen, bzw. −100 Punkte für einen schwarzen Bauern. Dann ±300 für Springer und Läufer, ±500 für Türme und ±1000 für Damen. Da die Abbildung ein ausgeglichenes Materialverhältnis zeigt, ist die Summe der

Figurenwerte gleich 0. Dem gegebenen Schema nach hätte Weiß also einen Vorteil von 124 Punkten.


Außerdem vergeben wir noch die Werte $-\infty$ für „Schwarz gewinnt ganz sicher“ und $\infty$ für „Weiß gewinnt ganz sicher“. Das ist z.B. dann der Fall, wenn wir eine Stellung betrachten, in der einer der Spieler matt gesetzt wurde. Das Zeichen „$\infty$“ wird auch als „unendlich“ bezeichnet, ihr braucht euch aber nicht den Kopf über „Unendlichkeit“ zu zerbrechen. Denkt euch stattdessen einfach, dass es sich um eine sehr große Zahl handelt; so groß, dass „Schwarz gewinnt ganz sicher“ oder „Weiß gewinnt ganz sicher“ damit ausgedrückt werden können.

Der wichtigste Teil eines Spielprogramms ist aber der so genannte „Suchalgorithmus“, der eine intelligente Vorausschau organisiert, indem er einen „Spielbaum“ auswertet. Ein Spielbaum ist dabei die Struktur, die entsteht, wenn wir uns vorstellen, welche Züge wir in einer während des Spiels entstandenen Situation ausführen könnten, welche Antworten unser Gegner daraufhin ausspielen könnte, wie unsere eigene Antwort wiederum aussehen könnte usw. Die nachfolgende Abbildung skizziert diese Vorstellung.



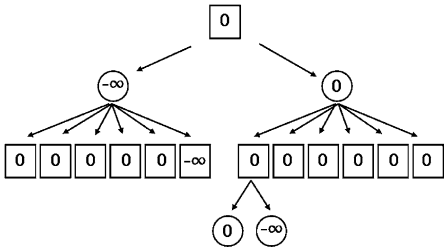
Beispiele dieser Art werden schnell unübersichtlich. Deshalb gehen wir im Folgenden davon aus, dass uns unsere Bewertungsprozedur für diejenigen Stellungen, für die keine Nachfolgestellungen abgebildet sind, die Werte 0 bzw. $-\infty$ liefert. Für die anderen Stellungen kennen wir die Werte zunächst nicht. Wohin soll Weiß dann ziehen? Falls er nach links zu (*) geht, darf Schwarz den nachfolgenden Zug wählen, und natürlich ginge der zu dem am weitesten rechts liegenden Nachfolger von (*). Schwarz würde in der Stellung (*) gewinnen, und deshalb hat die Stellung den Wert $-\infty$. Der Wert der Stellung (*) kann also dadurch gebildet werden, dass wir ihr den kleinsten Wert der Nachfolgestellungen von (*) zuordnen.

Geht man von der Stellung (**) nach links, darf Weiß den nachfolgenden Zug wählen. Einer davon hat den Wert 0, der andere $-\infty$. Käme es zu dieser Stellung würde Weiß natürlich nach links ziehen, so dass die Stellung ausgeglichen bliebe. Der Wert des linken äußeren Nachfolgers der Stellung (**) wird deshalb gebildet, indem man den größten Wert von dessen Nachfolgestellungen heranzieht. Betrachten wir nun wieder die Stellung (**). Offenbar haben alle Nachfolgestellungen den Wert 0, und weil der kleinste Wert dementsprechend auch 0 ist, hat (**) den Wert 0. An der Ausgangsstellung darf Weiß wählen, und er sollte dorthin ziehen, wo er am meisten erreichen kann, also nach rechts. Die Ausgangsstellung ist eine ausgeglichene Stellung, was sich auch dadurch äußert, dass der größte (größere) Wert der Stellungen (*) und (**) gleich 0 ist.

Man nennt das vorgestellte Schema, den Stellungen Werte zuzuordnen das „Minimax-Prinzip“. Der Begriff bedeutet „Minimum und Maximum bilden“, und die Begriffe *Minimum* bzw. *Maximum* kommen aus dem Lateinischen und bedeuten „das Größte“ bzw. „das Kleinste“. Eine einfache Möglichkeit, die Werte der Stellungen im Spielbaum zu ermitteln ist es, von unten nach oben, ausgehend von den Endstellungen ohne Nachfolger, den jeweiligen Vorgängern Werte gemäß diesem Minimax-Prinzip zuzuordnen. Den besten Zug findet man dann an der Ausgangsstellung, indem man die möglichen Folgestellungen und deren Werte betrachtet und einen Zug zu demjenigen Nachfolger auswählt, der einem gemäß dieser Werte am besten gefällt.

## Der Suchalgorithmus

Im Folgenden geht es darum, einen pfiffigen Algorithmus zu erklären, der den besten Zug an der Ausgangsstellung ermittelt, *ohne* den ganzen Baum zu untersuchen. Und weil Spielbäume sehr schnell sehr unübersichtlich werden, sollten wir uns im Folgenden auf die für uns wichtigen Aspekte von Spielstellungen konzentrieren. Anstatt uns Spielbäume wie in obiger Abbildung als „Schachbäume“ vorzustellen, stellen wir uns die Stellungen besser als Quadrate und Kreise vor, je nachdem ob Weiß oder Schwarz am Zug ist. Der Spielbaum des obigen Beispiels sieht dann wie folgt aus:




Außerdem sollten wir die Ausdrücke „Stellungen, bei denen Weiß am Zug ist“ und „Stellungen, bei denen Schwarz am Zug ist“ sinnvoll abkürzen. Nennen wir sie doch einfach *Max-Stellungen* (Quadrate) und *Min-Stellungen* (Kreise). So, jetzt sieht der Spielbaum doch schon viel übersichtlicher aus, und man kann schön sehen, dass Weiß an der Wurzel nach rechts ziehen sollte, um das Spiel in eine ausgeglichene Stellung zu bringen.

Der Algorithmus der Wahl, um einen Min-/Max-Spielbaum zu durchsuchen, ist der *Alphabeta-Algorithmus*. Der ist etwas trickreich, ihr solltet genau aufpassen!

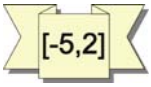
Stell dir vor, du erhältst den folgenden Auftrag:

1. Du bekommst von einem Auftraggeber eine *Schachstellung* und die Information, *wer am Zug ist*, sowie ein seltsames *Zettelchen, auf dem zwei Zahlen stehen*. Das sieht folgendermaßen aus: `[Zahl1,Zahl2]` und bedeutet, dass dein Auftraggeber sich nur dann für den exakten Wert der vorgegebenen Spielstellung interessiert, wenn der Wert zwischen den zwei gegebenen Zahlen liegt.
   Dazu ein kleines Beispiel: Nehmen wir an, auf dem Zettelchen steht $[-5, 2]$. Falls der echte Wert z.B. $-1$ ist, sollst du genau diesen Wert $-1$ liefern. Falls er aber außerhalb dieses so genannten Fensters liegt und z.B. $-6$ ist, reicht es zu sagen, dass der Wert kleiner oder gleich $-5$ ist. Liegt der Wert auf der anderen Seite außerhalb des Fensters, (ist er also z.B. 10), so genügt es, zu sagen, dass der Wert größer oder gleich 2 ist.
   *Du sollst nun den Wert der Schachstellung bestimmen, sofern dieser in dem gegebenen Fenster liegt. Ansonsten sollst du herausfinden, ob der Wert der Schachstellung kleiner als Zahl1 oder größer als Zahl2 ist. Du darfst dir dazu einen Helfer heranziehen.*
   Außerdem hilft dir dein Lehrer, der zu manchen Schachstellungen den Wert aus seiner langjährigen Schachspielerfahrung kennt.
2. *Falls* der Lehrer den Wert kennt, gibst du diesen Wert an deinen Auftraggeber und bist fertig.

*Alternativ:*

3. Schreibe alle möglichen Züge, die in deiner Spielstellung möglich sind, auf ein Stück Papier. Bearbeite die Züge auf deiner Liste einen nach dem anderen auf folgende Weise:

4. *Falls* Weiß am Zug ist, gehst du so vor:
   a. Zunächst gibst du deinem Helfer diejenige Stellung, die entsteht, wenn der Zug, den du gerade betrachtest, ausgeführt wird. Außerdem machst du ihm eine Kopie deines Zettels mit `[Zahl1,Zahl2]`. Du erklärst ihm genau, worum es geht, und dass er nach demselben Schema seine Spielstellung bewerten soll, wie du es für deine Stellung tust. Dann wartest du, bis er mit dem Ergebnis zu dir kommt.
   *Falls* der Wert, den er dir zurück gibt, größer ist, als deine Zahl1, überpinselst du die Zahl1 mit Tipp-Ex und ersetzt sie durch das Ergebnis deines Helfers. Sah z.B. dein Zettel so aus




   und liefert dir dein Helfer eine 0 zurück, sieht dein Zettel jetzt so aus:


   b. *Falls* deine neue Zahl1, also die linke Zahl auf deinem Zettel größer oder gleich der Zahl2 ist, *gibst du Zahl1* als Ergebnis an deinen Auftraggeber *zurück* und hörst auf. Wenn du schlau warst, hast du mit dem Auftraggeber eine Belohnung ausgehandelt und gehst jetzt kassieren.
   *Falls* die neue Zahl1 immer noch kleiner als die Zahl2 ist, nimmst du dir den nächsten Zug deiner Liste und gehst wieder zu Zeile 4.

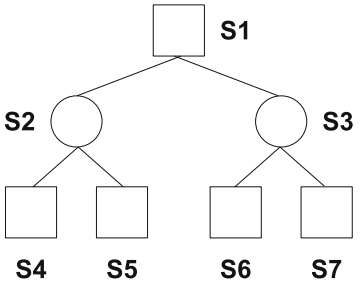
5. *Falls* in deiner Stellung Schwarz am Zug ist und nicht Weiß, wie in Zeile 4 angenommen, verhältst du dich ganz ähnlich:
   a. Zunächst gibst du deinem Helfer diejenige Stellung, die entsteht, wenn der Zug, den du gerade betrachtest, ausgeführt wird. Außerdem machst du ihm eine Kopie deines Zettels mit `[Zahl1,Zahl2]`. Du erklärst ihm genau, worum es geht, und dass er nach demselben Schema seine Spielstellung bewerten soll, wie du es für deine Stellung tust. Dann wartest du, bis er mit dem Ergebnis zu dir kommt.
   *Falls* der Wert, den er dir zurück gibt, kleiner ist, als deine Zahl2, überpinselst du die Zahl2 und ersetzt sie durch das Ergebnis deines Helfers.

   b. *Falls* deine neue Zahl2, also die rechte Zahl auf deinem Zettel kleiner oder gleich der Zahl1 ist, *gibst du Zahl2* als Ergebnis an deinen Auftraggeber *zurück* und hörst auf. Vergiss nicht die Belohnung zu kassieren.
   *Falls* die neue Zahl2 immer noch größer als die Zahl1 ist, nimmst du dir den nächsten Zug deiner Liste und gehst wieder zu Zeile 5.

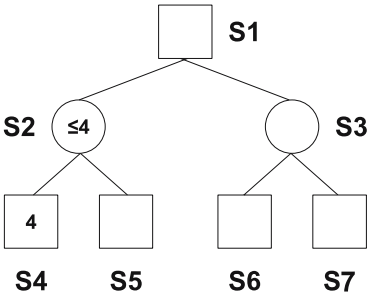
6. An diese Stelle kommst du nur, wenn du für alle Züge deiner Liste Ergebnisse von deinem Helfer bekommen hast. *Falls* Weiß am Zug ist, gibst

du Zahl1 als Ergebniswert an deinen Auftraggeber. *Falls* Schwarz am Zug ist, gibst du ihm deine Zahl2.

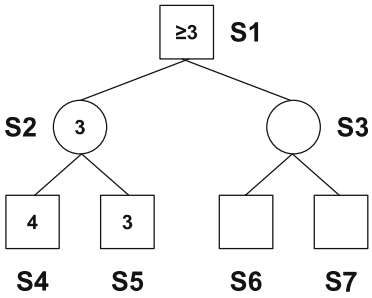
Die Zahl1 wird typischerweise „alpha“ genannt und die Zahl2 „beta“. Deshalb besitzt der hier vorgestellte Algorithmus den Namen *Alphabeta-Algorithmus*. Der Algorithmus verwendet das Prinzip der *Rekursion* (vgl. Kap. 3), und im Wesentlichen handelt es sich um eine *Tiefensuche* (vgl. Kap. 7), die einen Spielbaum von links nach rechts durchläuft. Das Besondere an dieser Tiefensuche ist, dass der Algorithmus in linken Teilen des Baums Informationen sammeln kann, die er dazu benutzt, um in rechten Teilbäumen Knoten nicht untersuchen zu müssen. Es gibt dabei keine Qualitätsverluste. Schauen wir uns dazu zunächst ein sehr kleines Beispiel an. Der Spielbaum soll wie folgt aussehen:



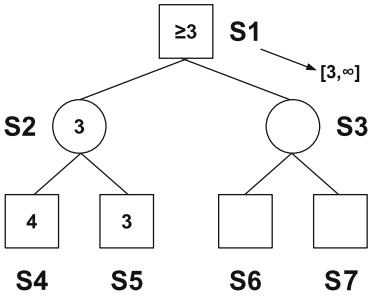

Die Stellungen sind jetzt mit S1 bis S7 bezeichnet. Der Algorithmus läuft zuerst links herunter bis zu S4, denn der Auftraggeber an S1 beauftragt seinen Helfer, sich S2 anzusehen, und der beauftragt wiederum seinen Helfer zu allererst einmal, sich S4 anzusehen. Weil S4 keine Nachfolger besitzt, kann der zweite Helfer unseren Lehrer nach dem Wert fragen. Der sagt uns z.B. den Wert 4.




Für S2 wissen wir nun, dass der Wert von S2 kleiner oder gleich 4 sein muss, da Schwarz einen Nachfolger mit minimalem Wert wählt. Danach geht es zu Stellung S5, wo wir eine 3 bekommen, und wir wissen für S2, dass der Wert 3 ist. Zurück in Stellung S1, wissen wir, dass der Wert von S1 größer oder gleich 3 ist, denn wenn Weiß an S1 nach links geht, wird er den Wert 3 erreichen. Was ihn erwartet, wenn er nach rechts geht, wissen wir noch nicht.

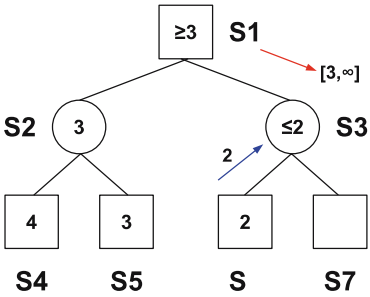



Jetzt kommt die Besonderheit des Algorithmus. Wir sind während des ganzen Beispiels noch gar nicht auf die „seltsamen“ Parameter Zahl1 und Zahl2 (bzw. auch *alpha* und *beta* genannt) eingegangen, weil sie bis hierher noch nicht interessant waren. Mit Hilfe dieser Parameter wird dem Sachbearbeiter für Stellung S3 die Information mitgegeben, dass der gesamte Unterbaum unter S3 für den Auftraggeber in S1 (und damit für die gesamte Rechnung) nur interessant ist, wenn der Wert von S3 größer als 3 ist. Das Zahlenpaar `[Zahl1,Zahl2]`, bzw. $[3, \infty]$ wird dem Helfer, der S3 bearbeiten soll, also von oben mitgegeben. In der Algorithmus-Beschreibung geschieht das mit Hilfe der Zettelchen. Im Detail sieht das so aus, dass derjenige Sachbearbeiter, der für die Spielstellung S1 ursprünglich $[-\infty, \infty]$ auf seinem Zettel stehen hatte, seine *Zahl1* für S1 mit Hilfe von Zeile 4a (vgl. oben) auf 3 erhöht hat. Da die Bedingung (*„Falls deine neue Zahl1, also die linke Zahl auf deinem Zettel größer oder gleich der Zahl2 ist...“*) in Zeile 4b nicht wahr ist und die Stellung S3 noch untersucht werden muss, nimmt sich der Sachbearbeiter von Stellung S1 auch noch den zweiten möglichen Zug vor und geht wieder zu Zeile 4. Er beauftragt dann seinen Helfer S3 zu untersuchen und gibt ihm diesmal einen Zettel mit $[3, \infty]$ mit auf den Weg.




Wenn S3 untersucht wird und ausgehend von dort S6, und in S6 z.B. der Wert 2 gefunden wird, dann wissen wir, dass der Wert von S3 kleiner oder gleich 2 ist. Wenn der für S3 verantwortliche Helfer Zeile 5a bearbeitet, wartet er dort auf seinen eigenen Helfer und bekommt von dem den Ergebniswert 2 mitgeteilt. Er setzt seine Zahl2 ebenfalls auf 2. Zusammenfassend ist dem Sachbearbeiter von S3 jetzt bekannt, dass der Wert von S3 höchstens 2 ist

und dass der genaue Wert von S3 für seinen Auftraggeber in S1 nur von Interesse wäre, wenn der Wert größer als 3 wäre.




Das aber heißt, dass, egal was in S7 für ein Wert gefunden würde, die Stellung S3 die Entscheidung von Weiß an S1 nicht mehr beeinflussen kann. *Dann brauchen wir S7 auch nicht zu untersuchen!* Man sagt, der Knoten S7 wird durch die Anweisung in Zeile 5b „weggeschnitten".

## Zusammenfassung

Der entscheidende Clou eines Schachprogramms ist es, mit Hilfe eines Spielbaums möglichst weit „in die Zukunft" zu schauen, d.h. nachfolgende Spielzüge möglichst weit im voraus zu planen. Dabei bewertet das Programm Stellungen, die es aus Zeitgründen nicht weiter verfolgen kann, mit Hilfe von Schätzwerten einer Bewertungsprozedur und rechnet diese Werte dem Minimax-Prinzip folgend zur Ausgangsstellung zurück. In der Tat kann man aufgrund der Baumbeschneidungen mit Hilfe des Alphabeta-Algorithmus doppelt so weit „in die Zukunft schauen" als wenn man den ganzen Baum durchsuchte. Das ist besonders beeindruckend, wenn man bedenkt, dass die Spielstärke von Schachprogrammen maßgeblich davon abhängt, wie weit sie vorausrechnen können. Zur Einschätzung: Wenn ein modernes Schachprogramm bis zu einer nominellen Tiefe von 20 rechnet, schlägt es jeden Menschen beim Turnierschach mehr oder weniger regelmäßig. Ohne Alphabeta-Algorithmus könnte es nur 10 Züge vorausschauen und hätte damit gerade einmal die Spielstärke eines Spielers der 2. Bundesliga.

## Algorithmus im Pseudocode

Für diejenigen von euch, die bereits Programmiererfahrung besitzen, haben wir den Algorithmus zusätzlich in einer Schreibweise ähnlich zu einem Computerprogramm aufbereitet.

Der Algorithmus ALPHABETA

```
1  function ALPHABETA(KNOTEN v, INT a, INT b): INT
2     if (v hat keinen Nachfolger) then return f(v);
        // Blattbewertung
3     für alle Nachfolger w von v do
4        if (MAX-Spieler ist bei v am Zug) then
5           a := max(a, ALPHABETA(w, a, b));
6           if (a >= b) then return a;      // Beta-Cutoff
7        else
8           b := min(b, ALPHABETA(w, a, b));
9           if (a >= b) then return b;      // Alpha-Cutoff
10       end if
11    end do
12    if (MAX-Spieler ist bei v am Zug) then
13       return a;
14    else
15       return b;
16    end if
17 end function
```

## Zum Weiterlesen

1. Kapitel 3 (Schnelle Sortieralgorithmen)

   Hier wird das Prinzip der Rekursion eingeführt.

2. Kapitel 7 (Tiefensuche)

   Dieses Kapitel erläutert die Tiefensuche.

3. Videos und Textmaterial zum Schachtürken finden sich auf den Seiten des Heinz Nixdorf MuseumsForum:
   `http://www.hnf.de/Schachtuerke/index.html`

4. `http://de.wikipedia.org/wiki/Alpha-Beta-Suche`

   Der Wikipedia-Artikel liefert einen gut zugänglichen Ausgangspunkt für alle, die mehr erfahren möchten über die Alphabeta-Suche.

5. `http://de.wikipedia.org/wiki/Hydra_(Schach)`

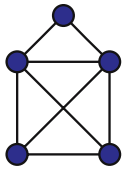
   Hier erhält man Informationen zum Schachcomputer Hydra.

# 29

# Die Eulertour

Michael Behrisch, Amin Coja-Oghlan und Peter Liske

Humboldt-Universität zu Berlin



Ein ziemlich unterhaltsamer Zeitvertreib ist bekanntlich, seine Mitschüler mit Rätseln zu ärgern, deren Lösung man bereits kennt. Ein schönes Beispiel dafür ist das „Haus des Nikolaus“:

Dabei handelt es sich um eine Figur aus fünf *Knoten* (die dicken Punkte) und acht *Kanten* (die Linien, die je zwei Knoten miteinander verbinden). *Kann man die rechts stehende Figur zeichnen, ohne den Stift abzusetzen und ohne eine Linie doppelt zu ziehen?*

Zwar hat sich die Lösung ziemlich schnell herumgesprochen (zumal es sogar 44 verschiedene Lösungen gibt), aber zum Glück gibt es andere schöne Figuren, die man auch in einem Zug zeichnen kann – wenn man weiß, wie es geht:

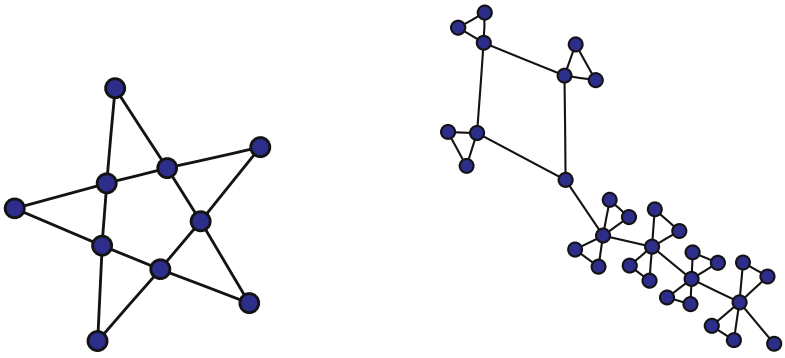


**Abb. 29.1.** Stern und Drachen zum Ausprobieren

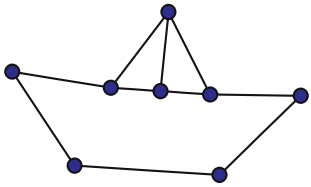


**Abb. 29.2.** Das Segelschiff als Gegenbeispiel

Bei manch anderen Figuren muss man vielleicht ziemlich lange probieren – um am Ende festzustellen, dass es gar nicht geht (vgl. Abb. 29.2).

Wir wollen hier einen *Algorithmus* vorstellen, der eine Figur *immer* in einem Zug zeichnet, wenn das möglich ist. Dazu beschäftigen wir uns zunächst nur mit Figuren, die man derart in einem Zug durchzeichnen kann, dass *am Ende der Stift wieder an dem Ausgangspunkt ankommt*. Den Weg, den der Stift dabei zurücklegt, nennt man eine *Eulertour*, denn die Frage, wann man eine Figur ohne abzusetzen zeichnen kann, wobei der Endpunkt gleich dem Startpunkt ist, wurde erstmals von dem Mathematiker *Leonhard Euler* im Zusammenhang mit dem so genannten *Königsberger Brückenproblem* (siehe letzter Abschnitt) beantwortet. Dabei war Euler vorrangig an der Frage interessiert, ob das Problem eine Lösung besitzt oder nicht; wir werden uns später auch mit der Frage beschäftigen, wie man solch eine Lösung schnell finden kann.

Wir werden im Folgenden sehen, warum man beispielsweise den Stern mit einer Eulertour zeichnen kann, das Schiff und das Nikolaushaus jedoch nicht und auch warum man das Nikolaushaus mit verschiedenem Start- und Endpunkt in einem Zug zeichnen kann, beim Schiff jedoch nicht einmal das klappt. Hierzu ist es wichtig, sich die *Knoten* der Figur anzuschauen, also diejenigen Stellen, an denen der Stift die Richtung wechseln kann (in den Bildern als dicke Punkte gezeichnet).

## Wann gibt es überhaupt eine Eulertour?

Der *Grad* eines Knotens ist die Anzahl der Linien, die sich dort treffen, so ist z. B. der Grad der Mastspitze des Schiffes gleich 3. Wenn man eine Figur so in einem Zug zeichnet, dass Start- und Endpunkt übereinstimmen und keine Kante mehr als einmal benutzt wird, dann führt man den Stift dabei natürlich in jeden Knoten genau so oft hinein, wie man ihn auch wieder hinausführt. Das bedeutet: *der Grad jedes Knotens muss eine gerade Zahl sein.* Aber das Schiff hat sogar vier Knoten mit ungeradem Grad (alle am Segel). Deshalb ist es unmöglich, diese Figur in einem Zug mit gleichem Start- und Endpunkt zu zeichnen. (Dies gilt auch für das „Haus des Nikolaus“, weil es in dieser Figur

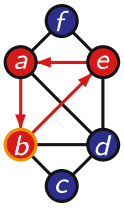
zwei Knoten mit Grad 3 gibt. Hier hilft jedoch ein kleiner Trick, der später verraten wird, um wenigstens ohne abzusetzen mit *verschiedenen* Start- und Endpunkten zu zeichnen.)

Im Gegensatz dazu haben beim Stern alle Knoten einen geraden Grad. *Gibt es in allen solchen Figuren eine Eulertour? Falls ja, wie können wir eine finden?*

## Wie man eine Eulertour findet

Wenn einem nichts besseres einfällt, könnte man einfach *irgendwo anfangen und „draufloszeichnen“*. Wir beginnen dabei an einem beliebigen Knoten und folgen von dort irgendeiner Kante, so dass wir einen neuen Knoten erreichen. Dort folgen wir wieder einer Kante zu einem neuen Knoten usw. Wir dürfen dabei allerdings jede Kante nur einmal benutzen.

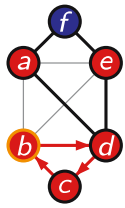
Jedesmal, wenn wir einen Knoten betreten und verlassen haben, können wir zwei seiner Kanten nun nicht mehr benutzen. Hatte der Knoten vor dem Betreten eine gerade Anzahl von verfügbaren Kanten, so hat er nachher ebenso eine gerade Anzahl. Es kann uns also nicht passieren, dass wir in einer Sackgasse stecken bleiben. Daher werden wir früher oder später mit unserer „Drauf-los“-Strategie wieder im Ausgangsknoten ankommen.




Im Bild links ist dabei ein „Kreis“ (also ein Weg über mehrere Kanten, der am Ausgangspunkt ankommt) mit drei Kanten entstanden. Die durchlaufenen Knoten sind `b`, `e` und `a`.

Meistens gelingt es so aber noch nicht, die ganze Figur zu zeichnen, da noch einige Kanten übrigbleiben. Wir sind also beim Einzeichnen des Kreises an irgendeiner Stelle „falsch abgebogen“ und haben dadurch einen Teil der Figur ausgelassen. Deshalb müssen wir versuchen, *den Kreis, den wir schon eingezeichnet haben, noch zu „erweitern“*. Die Kanten, die wir schon besucht haben, können wir dazu einfach löschen.

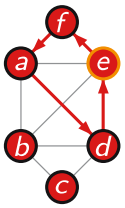
Dies geht z. B., indem man vom Start-/Zielknoten des schon erstellten Kreises nochmals losläuft. Dabei erhalten wir einen weiteren Kreis. Im Beispiel ist dies das Dreieck zwischen den drei unteren Knoten `b`, `d` und `c` (rechtes Bild).




Die beiden Kreise können wir nun zu einem längeren zusammenfügen und erhalten einen Kreis `(b,e,a,b,d,c,b)`. Aber damit sind wir immer noch nicht fertig. Die Kanten vom Startknoten sind nun verbraucht, von diesem Knoten geht es also nicht weiter.

Aber: Wenn wir die Kanten, die wir schon nachgezeichnet haben, aus der Figur löschen, haben immer noch alle Knoten einen geraden Grad. Deshalb können wir *auf dem übriggebliebenen Teil wieder einen neuen Kreis finden*: Wir suchen uns dazu einen Knoten auf unserem „alten“ Kreis, der eine Kante

hat, die wir noch nicht nachgezeichnet haben. Von diesem Knoten aus zeichnen wir wieder drauf los, bis wir einen neuen Kreis eingezeichnet haben. Im Beispiel erhalten wir ein Viereck mit den Eckpunkten `a`, `d`, `e` und `f` wie im linken Bild.

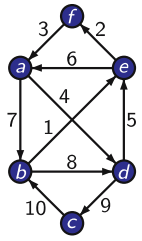



Wir haben nun einen „alten“ Kreis und einen „neuen“, der an irgendeinem Knoten des alten Kreises beginnt und endet. Die Idee ist, den neuen Kreis in den alten „einzuhängen“, um einen längeren Kreis zu erhalten. Dazu folgen wir zunächst dem ersten Kreis bis zu der Stelle, an der wir den zweiten Kreis begonnen haben, im Beispiel ist das der Knoten `e`. Von dort an folgen wir dem zweiten Kreis, bis wir wieder an dessen Ausgangspunkt sind. Schließlich setzen wir den ersten Kreis fort bis zu Ende und erhalten so die Tour `(b,e,f,a,d,e,a,b,d,c,b)`.

Somit sind jetzt die beiden „kleinen“ Kreise vereinigt und daraus ein „größerer“ Kreis geworden, und damit ist, in unserem Beispiel, die ganze Figur gezeichnet. Was aber, wenn auch der „größere“ Kreis noch immer nicht die ganze Figur abdeckt. Was spricht dagegen, das Verfahren einfach zu wiederholen?

Also: Wir suchen uns einen Knoten auf unserem „größeren“ Kreis, von dem eine Kante ausgeht, die wir noch nicht besucht haben. Dann finden wir in der Figur, aus der der „größere“ Kreis gelöscht ist, einen neuen Kreis und verknüpfen diesen mit dem „alten“ Kreis wie vorher. Da unsere Figur aus einem Stück besteht, ist auch klar, dass, solange es ungezeichnete Kanten gibt, mindestens eine dieser Kanten einen „alten“ Kreis berührt.

Das wiederholen wir, bis alle Kanten der Figur verbraucht sind. Auf diese Weise bekommen wir eine Eulertour.




Angewandt auf unser Beispiel bedeutet dies, dass wir auch das Hintereinanderhängen der ersten beiden Kreise `(b,e,a,b)` und `(b,d,c,b)` als Verknüpfung sehen können und so im dritten Schritt mit dem Kreis `(e,f,a,d,e)` unsere endgültige Tour `(b,e,a,d,e,f,a,b,d,c,b)`, wie im rechten Bild angedeutet, konstruieren, wobei die Kanten in der Reihenfolge ihrer Nummern durchlaufen werden.

## Der Algorithmus

Der unten angegebene Algorithmus geht ganz ähnlich vor wie im obigen Beispiel, nur dass er das Zusammenkleben sofort erledigt. Das heißt, nachdem eine Teiltour gefunden wurde (z. B. `(b,e,a,b,d,c,b)`), wird die nächste gleich an Ort und Stelle eingefügt. Angenommen die aktuelle Tour ist `(b,e,a,b,d,c,b)` und der in Zeile 4 gewählte Knoten $u =$ `a`, dann könnte die gewählte Kante `(a,d)` sein und so die Tour vorläufig zu `(b,e,a,d,b,d,c,b)` erweitert werden. Dies ist natürlich noch keine gültige Tour, aber der Algorithmus fährt ja auch fort, bis er wieder bei `a` ankommt.

Der Algorithmus EULERTOUR findet in einer Figur, in der jeder Knoten geraden Grad hat, einen Weg, diese in einem Zug zu zeichnen, und gibt die Reihenfolge, in der die Knoten abzuarbeiten sind, aus.

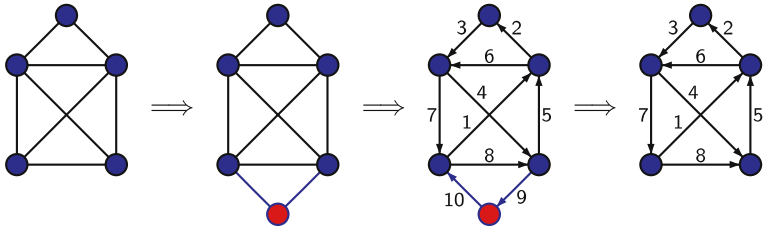
```
1   function EULERTOUR(Figur F)
2   begin
3       Tour := (s), für einen beliebigen (Anfangs-)Knoten s aus F
4       while es gibt einen Knoten u in der Tour, von dem noch eine Kante
        ausgeht
5           v := u
6           repeat
7               nimm eine Kante v − w, die in v beginnt
8               füge den anderen Endknoten w hinter v in der Tour ein
9               v := w
10              entferne die Kante aus F
11          until v = u      {Kreis geschlossen}
12      endwhile
13      return Tour
14  end
```

## Das Haus des Nikolaus

Bisher haben wir uns nur mit Figuren beschäftigt, die eine Eulertour haben, die also so in einem Zug gezeichnet werden können, dass Start- und Endpunkt übereinstimmen. Wie wir schon wissen, funktioniert das aber nur, wenn alle Knoten in der Figur einen geraden Grad haben. Leider trifft das auf das „Haus des Nikolaus“ nicht zu: Die beiden unteren Knoten haben Grad 3. Trotzdem kann man die Figur in einem Zug zeichnen, nur eben nicht mit gleichem Start- und Endpunkt.

*Wie können wir also unseren Algorithmus anpassen, so dass er auch für das „Haus des Nikolaus“ funktioniert?*

Ein einfacher Trick ist, einen neuen Knoten einzubauen und ihn mit den beiden Knoten vom Grad 3 zu verbinden: In dieser Figur haben alle Knoten einen geraden Grad, deshalb funktioniert der Algorithmus und liefert uns eine Eulertour.




Zum Schluss lassen wir einfach in dieser Eulertour den „künstlich eingebauten“ Knoten weg: Wir beginnen den Weg mit seinem linken „Nachbarknoten“

und hören bei seinem rechten Nachbarn auf – und bekommen eine Lösung für das „Haus des Nikolaus“! Im Beispiel funktioniert das besonders schön, weil die hinzugefügten Kanten als letzte in der Tour liegen. Anderenfalls muss man die gefundene Tour so „verschieben“ (d. h., die Kanten nicht in der Reihenfolge 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 sondern z. B. 4, 5, 6, 7, 8, 9, 10, 1, 2, 3 durchlaufen), dass diese am Schluss liegen.

Dieser Trick funktioniert immer dann, wenn die Figur, die wir zeichnen wollen, genau zwei Knoten von ungeradem Grad hat. Aber andere Figuren (mit mehr als zwei Knoten von ungeradem Grad) können gar nicht in einem Zug gezeichnet werden, selbst wenn man erlaubt, dass Start- und Zielknoten verschieden sind.

## Von Postboten und Müllmännern

Abgesehen davon, dass man mit dem beschriebenen Algorithmus seine Mitschüler beeindrucken kann (insbesondere damit, dass man schon durch „Draufschauen“ und schnelles Testen der Grade der Knoten entscheidet, ob sich eine Figur in einem Strich zeichnen lässt), gibt es auch ernsthaftere Anwendungen. Nimmt man z. B. an, dass die Kanten Straßen sind und die Knoten Kreuzungen, so ist ein Weg, der alle Straßen genau einmal durchläuft, gerade eine Eulertour. Wenn ein Straßennetz Eulertouren hat, stellen diese schnelle und benzinsparende Routen für die Postzustellung und die Müllabfuhr dar. Unser Algorithmus findet solche Touren auf einfache Weise. Für Netze mit Hunderten von Straßen übernimmt natürlich ein Computer die Arbeit.

Tatsächlich gibt es in Städten aber mehr als zwei Kreuzungen mit einer ungeraden Zahl von Straßen. Dadurch muss bei der Planung der Müllbeseitigung berücksichtigt werden, dass einige Straßen mehrfach genutzt werden. Dabei spielt auch die Länge der Straßen eine Rolle, was zum so genannten Briefträgerproblem (Chinese Postman Problem) führt.

So erlaubt die Beschäftigung mit Eulerkreisen nicht nur eine Rundreise durch Kanten und Knoten, sondern führt auch von einem vorwiegend angenehmen Transporteur (dem Nikolaus) zum nächsten (dem Briefträger).

## Zum Weiterlesen

1. Mathematische Basteleien:
   `http://www.mathematische-basteleien.de/nikolaushaus.htm`

   Eine Website mit vielen historischen und künstlerischen Zusatzinformationen sowie einer Liste aller Lösungen zum Haus des Nikolaus.

2. `http://de.wikipedia.org/wiki/Königsberger_Brückenproblem`

   Im Wikipedia-Artikel erfährt man alles über den Ausgangspunkt, den dieses Problem als „touristische“ Fragestellung an Leonhard Euler vor über 270 Jahren genommen hat.

3. `http://de.wikipedia.org/wiki/Leonhard_Euler`

   Leben und Werk des Namensgebers, nicht nur des Eulerkreises, sondern auch der Eulerschen Zahl und noch vieler anderer wichtiger mathematischer Errungenschaften.

4. Kapitel 9 (Zyklensuche in Graphen)

   Im Gegensatz zum Eulerkreis, der alle Kanten der Figur enthält, geht es in diesem Kapitel „nur“ darum zu entscheiden, ob es einen Kreis gibt (der eventuell einen bestimmten Knoten enthält). Das Verfahren ist unserem „einfach loslaufen“ recht ähnlich.

5. Kapitel 42 (Das Travelling Salesman Problem)

   Obwohl man nur ein winziges Detail ändert, nämlich jeden Knoten genau einmal besuchen möchte und dabei nicht unbedingt jede Kante benutzen muss, wird das Problem gleich viel schwieriger. Will man noch dazu einen kürzesten Weg finden, der das leistet, landet man bei einem der berühmtesten bislang ungelösten Optimierungsprobleme, das (vermutlich) nicht so einen schnellen Algorithmus wie das Eulerkreisverfahren besitzt.

6. Ulrich Kortenkamp: *Modellieren, Simulieren, Optimieren: Die Müllabfuhr.* `http://kortenkamps.net/material/EulerTour/`

   Diese Lerneinheit für den computergestützten Mathematikunterricht lässt einen interaktiv die Müllabfuhr der Stadt Schriesheim modellieren und planen.

7. Zum Weiterdenken:

   Wir haben gesehen, dass der Algorithmus für Figuren funktioniert, bei denen alle Knotenpunkte eine gerade Anzahl von ausgehenden Linien haben, dass man zwei „ungerade Knotenpunkte“ gut reparieren kann und es bei vieren schief geht. Was passiert aber bei einem oder drei „ungeraden Knotenpunkten“?

# 30

# Kreise zeichnen mit Turbo

Dominik Sibbing und Leif Kobbelt

RWTH Aachen

Wenn du dir ein Fernseh- oder Computer-Bild mal von ganz nahe ansiehst, wirst du feststellen, dass es eigentlich aus einer Menge ganz kleiner Lichtpunkte, den so genannten Pixeln (= Picture Elements), besteht. Um mit dem Computer ein Bild zu erzeugen, muss man also berechnen, welche Pixel dazu in welcher Farbe leuchten müssen. Das ist vergleichbar mit dem Ausmalen einzelner Kästchen auf einem karierten Blatt Papier (Abb. 30.1). Bei der großen Anzahl von Pixeln, die dabei verarbeitet werden müssen, ist klar, dass sehr effiziente Algorithmen notwendig sind, wenn z. B. ein Computer-Bild mit einer Million Pixel bei 30 Bildern pro Sekunde dargestellt werden soll. In der Computergrafik geht man daher so vor, dass jede darzustellende Szene in viele kleine Elemente (z. B. Dreiecke oder Linien) zerlegt wird, d. h., eine komplizierte Form wird durch viele einfache Formen angenähert. In vielen Anwendungen, wie z. B. Computerspielen stellt sich die Frage, wie solche geometrischen Grundbausteine möglichst schnell, d. h. mit möglichst wenigen und einfachen Rechenoperationen, gezeichnet werden können. Wir wollen dir dies am Beispiel von Kreisen verdeutlichen.

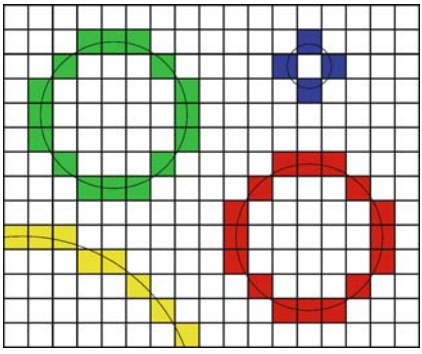


**Abb. 30.1.** Zeichnung mit Kreisen

## Kreise zeichnen mal ganz simpel

Um einen Kreis zu zeichnen, wird einem wohl zuerst der Zirkel als ideales Hilfsmittel einfallen, der auf dem einfachen Prinzip beruht, den Abstand der Zirkelspitze zum Mittelpunkt konstant zu lassen, während sie sich um 360° um den Mittelpunkt herumdreht. Möchten wir die Methode des Zirkels nachahmen, so müssen wir berechnen können, wo sich die Zirkelspitze zu jedem Zeitpunkt aufhält. Kennt man den Radius $R$ des Kreises und den Winkel $\alpha$, den die $x$-Achse und die Verbindungslinie der Zirkelspitze einschließen, so lässt sich die Position der Zirkelspitze über die Sinus- und Kosinus-Funktionen bestimmen (Abb. 30.2):

$$(x, y) = (R \cdot \cos(\alpha), R \cdot \sin(\alpha))$$

Nehmen wir an, wir hätten einen Befehl „*plot* $(x, y)$“ der das Pixel mit den Koordinaten $(x, y)$ auf dem Monitor anschalten kann. Ein Kreis entsteht nun, indem mit Hilfe der oben genannten Formel eine Menge von Pixeln auf dem Monitor angeschaltet wird. Hierzu starten wir einen ersten Versuch, um einen Kreis mit einer gewissen Zahl $N$ von Pixeln zu zeichnen:

Naiver Algorithmus zur Berechnung eines Kreises

```
1  for i := 0 to N − 1 do
2      x = R · cos(360 · i/N)
3      y = R · sin(360 · i/N)
4      plot(x, y)
5  endfor
```

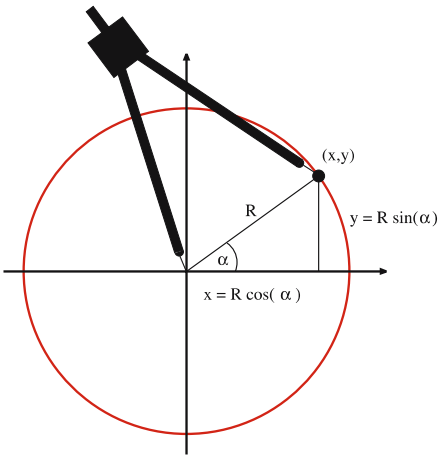



**Abb. 30.2.** Parametrisierung des Kreises

Wie man sieht, läuft der Winkel von 0 bis 360°, was genau wie beim Zeichnen mit dem Zirkel zu einem vollständigen Kreis führt.

Bleibt noch die Frage, wie man die Anzahl $N$ der Punkte bestimmt: Auf der einen Seite wollen wir möglichst wenige Punkte berechnen, da bei der Betrachtung von zu vielen Punkten unter Umständen immer wieder dieselben Pixel gesetzt werden, was unnötig Rechenzeit in Anspruch nehmen würde. Auf der anderen Seite dürfen aber auch nicht zu wenig Punkte betrachtet werden, weil sonst Lücken im Kreis entstehen könnten. Bei einer Pixelbreite von 1 Längeneinheit lässt sich mit Hilfe der Formel für den Kreisumfang

$$U = 2\pi R \leq 7R$$

ungefähr abschätzen, dass man nicht mehr als $7R$ Pixel benötigt, um den Kreis lückenlos zu zeichnen.

Obwohl der Algorithmus korrekt funktioniert, erfordern die vielen Multiplikationen, Additionen und auch die Auswertung der Sinus- und Kosinus-Ausdrücke einen hohen Rechenaufwand, was bei Bildern mit sehr vielen Kreisen zu langen Wartezeiten vor dem PC führt.

***Kreise sind symmetrisch.*** Können wir vielleicht Rechenzeit einsparen und unser Zeichenprogramm beschleunigen?

Als erstes fällt sicherlich die Symmetrie des Kreises auf, die wir unbedingt ausnutzen wollen. Zeichnet man nur den oberen Halbkreis, so lässt sich durch Spiegelung der Punkte an der $x$-Achse leicht der andere Halbkreis ergänzen. Diese Spiegelung ist besonders einfach, weil nur das Vorzeichen der $y$-Koordinate geändert werden muss. Es gibt sogar noch mehr Symmetrieachsen, die man ausnutzen kann, wie in der Abb. 30.3 zu erkennen ist. Die Spiegelung an der $y$-Achse ist ebenso einfach, weil hier nur die $x$-Koordinate des berechneten Punktes mit einem „Minus“ versehen werden muss. Nun reicht es schon aus, nur ein Viertel des Kreises zu berechnen. Das Zeichnen eines

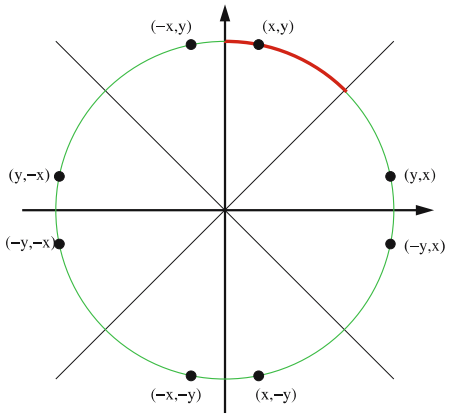



**Abb. 30.3.** Symmetrie des Kreises

Achtels des Kreises genügt, wenn man die beiden Diagonalen im Koordinatensystem zu Symmetrieachsen macht. Hier werden nur die Koordinaten des Punktes $(x, y)$ vertauscht. Aus $(x, y)$ wird also ganz einfach $(y, x)$. So berechnen wir anschaulich gesagt nur noch den Bereich zwischen 12:00 und 13:30 Uhr und ergänzen den Rest des Kreises (Abb. 30.3). Ist also $(x, y)$ bekannt, so sind auch die sieben weiteren Punkte

$$(y, x), (-y, x), (x, -y), (-x, -y), (-y, -x), (y, -x) \text{ und } (-x, y)$$

quasi ohne Rechenaufwand ableitbar.

Ein Algorithmus, der sich die Symmetrie des Kreises zu Nutze macht, ist fast um den Faktor 7 schneller als der „Naive Algorithmus“ und sieht wie folgt aus:

Verbesserter Naiver Algorithmus zur Berechnung eines Kreises

```
1  N = 7R
2  for i := 0 to N/8 do
3      x = R · cos(360 · i/N)
4      y = R · sin(360 · i/N)
5      plot(  x,  y); plot(  y,  x)
6      plot(−x,  y); plot(  y, −x)
7      plot(  x, −y); plot(−y,  x)
8      plot(−x, −y); plot(−y, −x)
9  endfor
```

## Der Bresenham-Algorithmus für Kreise

Nun wenden wir uns der Berechnung der Punkte $(x, y)$ zu, die immer noch recht aufwändig ist. Die Frage ist, kann diese Berechnung auf einfachere Rechenoperationen reduziert werden? Das heißt, ist es möglich, auf die Berechnung der Sinus- und Kosinusterme zu verzichten? Ein erster Schritt wäre es, sich den Satz des Pythagoras zu Nutze zu machen. Mit ihm lässt sich die Höhe $y$ in Abhängigkeit von $x$ sofort ausrechnen (siehe auch Abb. 30.2):

$$x^2 + y^2 = R^2, \text{ also } y = \sqrt{R^2 - x^2}$$

Ein Algorithmus, der diesen Ansatz umsetzt, müsste die $x$-Koordinate nur noch aufzählen und nicht mehr berechnen. Zur Bestimmung der zugehörigen $y$-Koordinate wird lediglich eine Wurzel ausgewertet.

Leider ist die Berechnung der Wurzel ähnlich rechenaufwändig wie die von Sinus und Kosinus. Dazu kommt, dass Sinus-, Kosinus- und Wurzelausdrücke vom Computer nur näherungsweise berechnet werden, was bedeutet, dass kleine Abweichungen von der korrekten Lösung möglich sind.

Können wir vielleicht auch auf die schwierige Berechnung der Wurzel verzichten und gänzlich mit einfachen und exakten Multiplikations- und Additionsoperationen auskommen? Der Algorithmus mit Turbo heißt „Bresenham-Algorithmus für Kreise“ und und wurde in seiner ursprünglichen Form für das Zeichnen von Linien von Jack E. Bresenham (1962) entwickelt. Für Kreise soll der Algorithmus im Folgenden beschrieben werden.

*Die Idee.* Man beginnt mit dem Zeichnen des obersten Pixels mit ganzzahligen Koordinaten $(0, R)$. In jedem Schritt des Algorithmus wird die $x$-Koordinate um eins erhöht. Da im ersten Achtel des Kreises die Steigung zwischen 0 und $-1$ liegt (Abb. 30.4), kann die $y$-Koordinate entweder gleich bleiben, d. h., wir bewegen uns nach „Osten“ oder um eins verringert werden, was der Bewegung nach „Süd-Osten“ entspricht. Diese Entscheidung wird so getroffen, dass der Mittelpunkt des neu berechneten Pixels (blauer Punkt in Abb. 30.4) möglichst nahe am Kreis (rot) liegt. Das wird solange gemacht, bis der Achtelkreis und damit über Spiegelungen der komplette Kreis berechnet wurde.

Um zu entscheiden, ob das Zentrum des östlichen oder das des südöstlichen Nachbarpixels näher am Kreis liegt, schaut man sich die im Bild grün eingezeichneten Punkte an. Diese Punkte sind immer um eine Einheit nach rechts und eine halbe Längeneinheit nach unten gegenüber der aktuellen Position des blauen Punktes verschoben. Da die Startposition bei $(0, R)$ liegt, müssen wir für den ersten Schritt den grünen Punkt an der Stelle $(1, R - \frac{1}{2})$ testen,

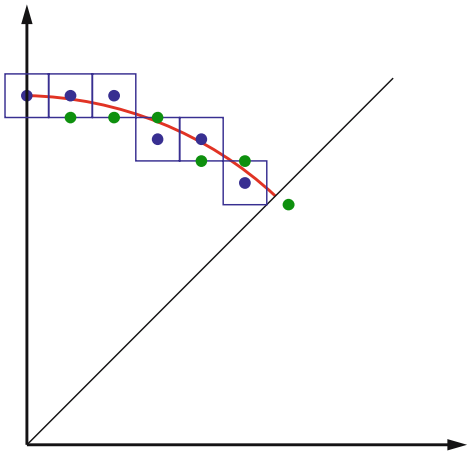


**Abb. 30.4.** Bresenham-Algorithmus für Kreise

der genau in der Mitte zwischen dem östlichen und südöstlichen Nachbarn von $(0, R)$ liegt.

Die Entscheidung sieht nun wie folgt aus:
*Fall 1*: Liegt der grüne Punkt innerhalb des Kreises, so bewegen wir uns im nächsten Schritt nur nach Osten:

$$(x, y) \leftarrow (x + 1, y)$$

*Fall 2*: Liegt er außerhalb des Kreises, so bewegen wir uns nach Süd-Osten:

$$(x, y) \leftarrow (x + 1, y - 1)$$

*Entscheidung: innen oder außen?* Wir benötigen eine Möglichkeit um festzustellen, ob ein Punkt innerhalb oder außerhalb des Kreises liegt. Nur so können wir unsere Laufrichtung („Osten“ oder „Süd-Osten“) festlegen.

Ein Punkt $(x, y)$ liegt immer dann innerhalb eines Kreises, wenn der Abstand zwischen dem Punkt und dem Kreismittelpunkt kleiner als der Kreisradius ist. Das ist gleichbedeutend damit, dass die Funktion

$$F(x, y) = x^2 + y^2 - R^2$$

einen Wert $< 0$ liefert. Das Gute ist: Dieser Test erfordert keine Wurzelberechnung mehr!

Der Funktionswert des ersten grünen Punktes ist schnell gefunden:

$$\begin{aligned} F\left(1, R - \frac{1}{2}\right) &= 1 + R^2 - R + \frac{1}{4} - R^2 \\ &= \frac{5}{4} - R \end{aligned}$$

Entscheiden wir uns dafür, von einem Punkt $(x, y)$ nach Osten zu laufen, so ändert sich der Wert der Entscheidungsfunktion $F$ in folgender Weise:

$$\begin{aligned} F(x + 1, y) &= (x + 1)^2 + y^2 - R^2 \\ &= x^2 + 2x + 1 + y^2 - R^2 \\ &= F(x, y) + 2x + 1 \end{aligned}$$

Bei der Entscheidung ein Pixel nach Süd-Osten zu laufen, ergibt sich der neue Wert von $F$ als

$$\begin{aligned} F(x + 1, y - 1) &= (x + 1)^2 + (y - 1)^2 - R^2 \\ &= F(x, y) + 2x - 2y + 2 \end{aligned}$$

Danach wird das neue Pixel gesetzt und das Ganze beginnt von vorne. Der Funktionswert von $F$ wird also nicht in jedem Schritt neu berechnet, sondern nur leicht verändert, indem wir etwas hinzuaddieren. Man spricht auch von „inkrementeller“ Berechnung der Funktionswerte, die deutlich schneller durchgeführt werden kann als die komplett neue Berechnung.

Mit diesen Informationen können wir bereits eine erste Version des Bresenham-Algorithmus für Kreise angeben.

Bresenham-Algorithmus für Kreise

```
1   (x, y) = (0, R)
2   F = 5/4 − R
3   plot(0,  R); plot( R, 0)
4   plot(0, −R); plot(−R, 0)
5   while (x < y) do
6       if (F < 0) then
7           F = F + 2 · x + 1
8           x = x + 1
9       else
10          F = F + 2 · x − 2 · y + 2
11          x = x + 1
12          y = y − 1
13      endif
14      plot( x,  y); plot( y,  x)
15      plot(−x,  y); plot( y, −x)
16      plot( x, −y); plot(−y,  x)
17      plot(−x, −y); plot(−y, −x)
18  endwhile
```

*Es geht noch etwas schneller.* In den beiden Inkrementen $(2x+1)$ und $(2x-2y+2)$ ist man noch auf Multiplikationen angewiesen. Etwas schneller wäre unser Algorithmus, wenn wir nur Additionen benötigen würden.

Um das zu erreichen, führen wir zwei neue Variablen $d_E$ und $d_{SE}$ ein, damit wir die Änderungen der Inkremente verfolgen können. Je nachdem, ob wir uns entscheiden nach Osten oder nach Süd-Osten zu gehen, erhöhen wir die Funktion $F$ entweder um $d_E$ oder um $d_{SE}$. Die Startwerte für beide Variablen lassen sich leicht bestimmen, indem man die Startwerte für $x$ und $y$ in die Inkremente einsetzt:

$$\begin{aligned} d_E(0,R) &= 2 \cdot 0 + 1 = 1 \\ d_{SE}(0,R) &= 2 \cdot 0 - 2 \cdot R + 2 = 2 - 2 \cdot R \end{aligned}$$

Nun müssen wir uns noch überlegen, wie sich $d_E$ und $d_{SE}$ ändern, wenn wir uns für eine der Richtungen entscheiden. Im Prinzip kann das genauso

berechnet werden wie der neue Werte für die Funktion $F$ selbst. Gehen wir nach Osten, so ändern sich $d_E$ und $d_{SE}$ in folgender Weise:

$$\begin{aligned} d_E(x+1,y) &= 2\cdot(x+1)+1 = d_E(x,y)+2 \\ d_{SE}(x+1,y) &= 2\cdot(x+1)-2\cdot y-2 = d_{SE}(x,y)+2 \end{aligned}$$

Bei der Entscheidung nach Süd-Osten zu gehen ändern sich $d_E$ und $d_{SE}$ zu

$$\begin{aligned} d_E(x+1,y-1) &= 2\cdot(x+1)+1 = d_E(x,y)+2 \\ d_{SE}(x+1,y-1) &= 2\cdot(x+1)-2\cdot(y-1)+2 = d_{SE}(x,y)+4 \end{aligned}$$

*Der Bruch ist unnötig.* Etwas unschön erscheint uns noch der Bruch in dem Startwert von $F$. Es wäre wünschenswert, nur mit ganzen Zahlen zu rechnen, weil diese im Computer exakt dargestellt werden können. Würde $F$ zu Beginn der Berechnung eine ganze Zahl sein, so würde dies auch nach den oben aufgeführten Änderungen so bleiben, weil eben auch nur ganze Zahlen addiert werden.

Um auf den Bruch verzichten zu können, fragen wir uns, was es bedeutet, wenn $F < 0$ ist. Aus dem Startwert

$$F = \frac{5}{4} - R$$

und der Tatsache, dass $F$ nur um ganze Zahlen verändert wird, erkennt man, dass $F$ zu jedem Zeitpunkt gleich $K + 1/4$ für eine ganze Zahl $K$ ist. Damit liegt $F$ in folgender Menge:

$$F \in \left\{ \dots, -\frac{3}{4}, \frac{1}{4}, \frac{5}{4}, \dots \right\}$$

Für alle Zahlen dieser Menge ist, wie man sieht, $F < 0$ gleichbedeutend mit $F - 1/4 < 0$. Also können wir im Algorithmus statt mit dem bisherigen $F$ auch mit dem um $1/4$ kleineren Wert, der eine ganze Zahl ist, die Abfrage auf $F < 0$ ausführen. Dies erreichen wir dadurch, dass wir bereits den Startwert von $F$ um $1/4$ kleiner machen, also Zeile 2 im Algorithmus durch $F = 1 - R$ ersetzen, ohne dass die Entscheidung nach Osten oder Süd-Osten zu gehen, falsch getroffen werden kann.

Nun können wir eine Version des Bresenham-Algorithmus für das Zeichnen von Kreisen angeben, die nur noch mit einfachen Additionen ganzer Zahlen auskommt:

Verbesserter Bresenham-Algorithmus für Kreise

```
 1  (x, y) = (0, R)
 2  F = 1 − R
 3  d_E = 1
 4  d_SE = 2 − 2 · R
 5  plot(0,  R); plot( R, 0)
 6  plot(0, −R); plot(−R, 0)
 7  while (x < y) do
 8      if (F < 0) then
 9          F = F + d_E
10          x = x + 1
11          d_E = d_E + 2
12          d_SE = d_SE + 2
13      else
14          F = F + d_SE
15          x = x + 1
16          y = y − 1
17          d_E = d_E + 2
18          d_SE = d_SE + 4
19      endif
20      plot( x,  y); plot( y,  x)
21      plot(−x,  y); plot( y, −x)
22      plot( x, −y); plot(−y,  x)
23      plot(−x, −y); plot(−y, −x)
24  endwhile
```

## Ein kleines Wettrennen

Mit diesem Algorithmus ist das Zeichnen von Kreisen ein Leichtes: Lässt man die Algorithmen in einem kleinen Beispielprogramm zur Zeitmessung gegeneinander antreten, so zeigt sich, dass mit diesem Ansatz das Zeichnen von Kreisen mehr als 14-mal so schnell geht, wie mit dem ersten vorgeschlagenen Verfahren! Außerdem werden nur sehr einfache ganzzahlige Additionen benötigt, was von speziellen Prozessoren zusätzlich ausgenutzt werden kann. Probier es zu Hause doch selbst einmal aus.

Ähnliche Algorithmen existieren auch für andere geometrische Grundbausteine, wie z. B. die Gerade oder das Dreieck. Diese sind oft auf einer sehr niedrigen Ebene in Grafiktreibern oder direkt auf der Graphikkarte implementiert, so dass Bilder sehr schnell berechnet werden können, was von enormer Wichtigkeit für viele Anwendungen, wie z. B. 3D-Computerspiele, ist. In dieser speziellen Anwendung sollen oft mehr als 40 Bilder pro Sekunde erzeugt werden, die ihrerseits wiederum aus sehr vielen dieser geometrischen Grundbausteine bestehen können.

Die hier gezeigte Vorgehensweise ist typisch für die Entwicklung von Algorithmen in der Informatik. Zunächst muss eine vorgegebene Aufgabenstellung in Formeln übersetzt werden, um das Problem präzise zu beschreiben und einen ersten, wenn vielleicht auch ineffizienten Algorithmus zu finden. Dies hilft oft auch dabei, das Problem genauer zu verstehen (z. B.: Wie viele Pixel sind für einen lückenlosen Kreis notwendig?). Dann überlegt man sich, wie man den Algorithmus beschleunigen kann. Hierzu kann es helfen, zusätzliches Wissen einzubringen (z. B. Symmetrie) oder das Problem durch mathematische Transformationen anders zu formulieren (z. B. inkrementell). Schließlich kann man Kenntnisse über die Architektur und Funktionsweise von Computern ausnutzen (z. B., dass einfache Rechenoperationen schneller sind), um die Geschwindigkeit weiter zu steigern. Durch diese Vorgehensweise hat man am Ende vielleicht nicht nur eine optimale Lösung für das gegebene Problem gefunden, sondern kann die gefundene Lösung oft auch auf andere Probleme anwenden. Unser Kreis-Algorithmus mit Turbo kann z. B. auch so angepasst werden, dass man mit ihm Ellipsen, Parabeln, Hyperbeln und auch die meisten anderen in der Computergrafik verwendeten Kurven zeichnen kann.

## Zum Weiterlesen

1. Kapitel 8 (Der Pledge-Algorithmus) und 38 (Kleinster umschließender Kreis)

   Hier werden Verfahren vorgestellt, deren Resultate graphisch dargestellt werden können. Dazu müssen geometrische Grundbausteine schnell gezeichnet werden können, was mit dem Bresenham-Algorithmus für Linien und Kreise möglich ist.

2. Kapitel 11 (Multiplikation langer Zahlen)

   Der Bresenham-Algorithmus für Kreise kommt ohne Multiplikationen aus. Wenn man dennoch auf Multiplikationen angewiesen ist, sollten diese auch möglichst schnell durchgeführt werden können. Wie das am besten geht, beschreibt dieses Kapitel.

3. Klaus Zeppenfeld: *Lehrbuch der Grafikprogrammierung*. Spektrum Akademischer Verlag, 1. Auflage, 2003.

   Dieses Buch beschreibt grundlegende Algorithmen der Grafikprogrammierung anschaulich und mit Beispielen.

4. Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner: *The OpenGL Programming Guide*. Addison-Wesley Professional, 5. Auflage, 2005.

   Dieses Buch geht weit über das Zeichnen von geometrischen Primitiven hinaus und beschreibt mit vielen Beispielen zum Nachprogrammieren die Grafikprogrammierung in OpenGL.

# 31

# Gauß-Seidel Iteration zur Berechnung physikalischer Probleme

Christoph Freundl und Ulrich Rüde

Universität Erlangen-Nürnberg

## Zum Aufwärmen: Fußball

Bei diesem Algorithmus wird es um die Berechnung von physikalischen Effekten gehen. Computer können nämlich auch verwendet werden, um natürliche Vorgänge zu simulieren. Das wird in der Wissenschaft immer wichtiger, weil man damit oft besser verstehen kann, wie die Natur funktioniert. Unsere Wettervorhersage beruht z.B. auf einer Simulation, bei der es auf die möglichst genaue Abbildung der Natur im Computer ankommt. Neue Automodelle und Flugzeuge werden auch simuliert, lange bevor sie zum ersten Mal gebaut werden. Viele Wissenschaftler sind sogar ganz auf die Simulation angewiesen. Astronomen, die wissen wollen was geschieht, wenn zwei schwarze Löcher zusammenstoßen, müssen Computersimulationen verwenden, denn Experimente sind dazu natürlich nicht möglich. Auch Computerspiele sind übrigens oft Simulationen, nur dass es bei Spielen nicht unbedingt darauf ankommt, dass die Berechnungen mit der realen Welt übereinstimmen.

Mit unserem Algorithmus werden wir zumindest ein wichtiges Problem simulieren können, nämlich das der Wärmeverteilung, wie man sie z.B. für die Wettervorhersage auch braucht. Allerdings werden wir uns auf feste Gegenstände wie beispielsweise eine zweidimensionale Platte beschränken, weil es für die Simulation am Anfang leichter ist, wenn man die Strömung der Luft nicht mit berücksichtigen muss. Bevor wir zur eigentlichen Aufgabe kommen, machen wir erst mal eine „Aufwärmübung“ und schauen uns beim Fußball um.

Bei der letzten WM hat zwar schon nicht viel gefehlt, aber bei der kommenden Europameisterschaft steht die Nationalmannschaft endlich doch im Finale. Die Spieler sind ganz aufgeregt, so aufgeregt, dass der Trainer davon überzeugt ist, dass sie es schon nicht schaffen werden, sich zur Nationalhymne in einer Reihe, nach Rückennummern geordnet, aufzustellen.

Nachdem der Trainer das Spielfeld nicht betreten darf, wo er die Spieler ohne Weiteres selbst auf ihre Plätze in der Reihe stellen könnte, überlegt er sich voller Verzweiflung folgende Vorgehensweise: Den beiden Spielern mit den Nummern 1 und 11, die am weitesten rechts und links stehen werden, schärft

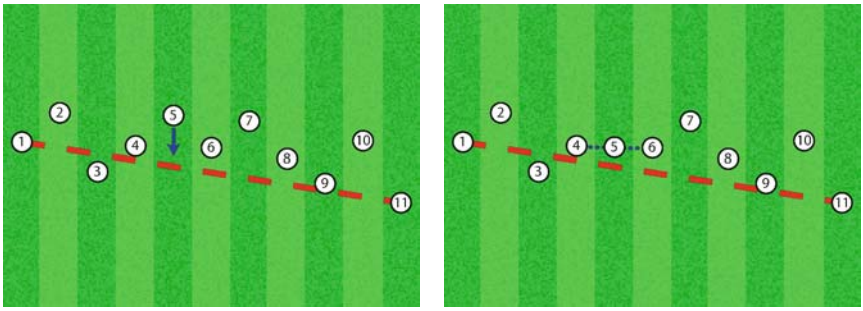
er ein, nur darauf Acht zu geben, dass die übrigen Spieler noch zwischen ihnen Platz haben. Ansonsten sollen sie sich nicht mehr von der Stelle rühren.

Die restlichen Spieler mit den Nummern 2 bis 10 erhalten folgende Anweisung: Wenn sie aufgerufen werden, sollen sie sich so bewegen, dass sie genau in der Mitte zwischen ihrem rechten und linken Nachbarn stehen (s. Abb. 31.1). Der Trainer ruft nacheinander alle Spieler in der Reihenfolge ihrer Trikotnummern auf, und nachdem sich der Spieler mit der höchsten Nummer bewegt hat, fängt der Trainer wieder von vorne an.

Man kann das selbst schnell ausprobieren, z.B. mit Spielfiguren. Auf der Webseite zu diesem Algorithmus befindet sich ein Programm, dass diese Vorgehensweise veranschaulicht. Man erkennt dabei: Nach einigen Durchläufen dieses Verfahrens haben sich die Spieler tatsächlich so bewegt, dass sie zumindest einigermaßen in einer Reihe stehen. Ganz genau stimmt es zwar nicht, aber es ist gut genug, dass keiner etwas merkt.

Um die Spieler ganz genau auf die Linie zu bekommen, müsste man den Algorithmus unendlich lange laufen lassen. Deshalb bekommt man in der Praxis nie das ganz richtige Resultat. Aber das macht nichts, denn nach genügend vielen Schritten kommt das Ergebnis des Algorithmus der richtigen Lösung beliebig nahe. Solche Algorithmen findet man häufig, wenn es um so genannte numerische Probleme geht, also dort, wo echte Kommazahlen ausgerechnet werden, wie man sie z.B. als Physiker oder Ingenieur braucht.

Wenn sich die Spieler in der Reihenfolge ihrer Trikotnummern aufstellen, dann werden sie immer wieder von links nach rechts aufgerufen, aber das hätten wir auch anders machen können. Eine beliebte Variante ist die sogenannte *Rot-Schwarz-Sortierung* der Spieler. Dabei belegt die Hälfte der Spieler mit den niedrigen Trikotnummern zunächst jeden zweiten Platz in der Reihe, anschließend füllen die Spieler mit den hohen Trikotnummern die restlichen Plätze auf. Aber auch, wenn die Spieler sich komplett zufällig aufstellen, funk-




(a) Spieler 5 ist an der Reihe (b) Spieler 5 ist auf der neuen Position

**Abb. 31.1.** Situation während der Aufstellung: Spieler 5 wird vom Trainer aufgerufen; er bewegt sich genau in die Mitte zwischen seinen Nachbarn, den Spielern 4 und 6. Die rot gestrichelte Linie ist durch die beiden Spieler am Rand vorgegeben, auf ihr sollen am Ende idealerweise alle Spieler stehen

tioniert das Verfahren, es muss nur immer für jeden Spieler klar sein, welches seine beiden Nachbarn sind.

Dieses Verfahren ist der Algorithmus von Gauß und Seidel, den wir im Folgenden für physikalische Probleme anwenden wollen.

## Temperaturberechnung in einem Stab (1D)

Kommen wir nun also wirklich zur Berechnung einer Temperaturverteilung. Ist es nicht ein bisschen erstaunlich, dass wir auch dazu das Prinzip des Sich-in-einer-Reihe-Aufstellens verwenden können? Betrachtet man z.B. die Temperaturverteilung in einem dünnen Stab, so stellt man nämlich fest, dass die Temperatur an jedem Punkt des Stabes dem Mittelwert der Temperaturen in der Umgebung des Punktes entspricht. Fixiert man die Temperatur an den beiden Enden des Stabes, dann verläuft die Temperatur zwischen den beiden Enden „in einer Reihe“, also linear vom einen zum anderen Ende.

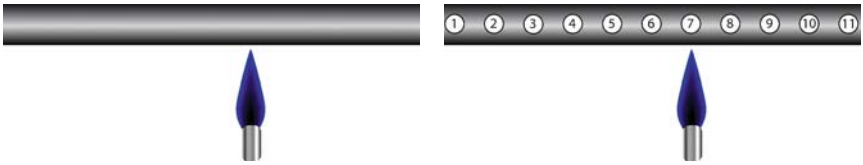
Um diese lineare Verteilung zu berechnen, braucht man mit Sicherheit noch keinen Computer, genau so wenig, wie ein Fußballtrainer normalerweise keinen komplizierten Algorithmus braucht, um die Spieler in einer Reihe aufzustellen. Aber wir sehen, dass die Probleme verwandt sind und sich vielleicht auf gleiche Weise lösen lassen. Die Position des Spielers entspricht dem Temperaturwert, und ansonsten verläuft die Berechnung auf gleiche Weise wie bei unserem Fußballproblem.

Als Nächstes machen wir die Aufgabe noch etwas interessanter: Wie sieht es denn mit dem Temperaturverlauf aus, wenn man den Stab zusätzlich an einer Stelle in der Mitte erhitzt? Dann gilt für die Temperatur an dieser Stelle natürlich nicht mehr, dass sie dem Durchschnitt der benachbarten Temperaturen entspricht, hinzu kommt ja noch die zusätzliche Erwärmung.

Jetzt gibt es keine offensichtliche Möglichkeit mehr, den resultierenden Temperaturverlauf zu bestimmen, also überlegen wir uns, wie wir den Computer zur Lösung dieses Problems benutzen können. Eine erste Schwierigkeit ergibt sich aus der Tatsache, dass entlang des Stabes unendlich viele Punkte liegen, jeder Computer aber nur endlich viele Objekte in endlicher Zeit bearbeiten kann. Wir wählen daher entlang des Stabes eine endliche Anzahl an Punkten aus (s. Abb. 31.2), an denen wir die Temperatur berechnen wollen. Diese Vorgehensweise nennt man auch *Diskretisierung*, da man ein kontinuierliches Problem auf ein diskretes Problem abbildet.

Sind die Punkte gleichmäßig über den Stab verteilt und bezeichnet $u_i$ den Temperaturwert sowie $f_i$ die Erhitzung an Punkt $i$, dann wird die Temperatur an einem Punkt aktualisiert über die Formel

$$u_i := \frac{1}{2}\left(u_{i-1} + u_{i+1}\right) + f_i$$




(a) Kontinuierlicher Stab (b) Diskretisierter Stab

**Abb. 31.2.** Diskretisierung eines kontinuierlichen eindimensionalen Stabes mit 11 Punkten

Der Algorithmus GaussSeidel1D

```
1  procedure GaussSeidel1D (n, u, f)
2  begin
3      for i := 2 to n − 1 do
4          u[i] := 1/2 (u[i − 1] + u[i + 1]) + f[i]
5      endfor
6  end
```

Wie im Beispiel der Fußballspieler werden nur die Punkte im Inneren des Stabes laufend neu berechnet, an den beiden Punkten am Rand des Stabes nimmt man die Temperatur als gegeben an. Je nach dem physikalischen Experiment kann es aber auch sein, dass die Temperatur an den Stabenden nicht fixiert ist. Wir haben auch noch nicht überlegt, dass der Stab in der Praxis Wärme an seine Umgebung abgibt. Das kann man durch entsprechend kompliziertere Formeln und Berechnungen mit berücksichtigen, die uns jetzt aber zu weit führen würden. Statt dessen sehen wir uns eine andere Schwierigkeit an, die sich ergibt, wenn wir nicht die Temperaturverteilung in einem eindimensionalen Stab, sondern in einer zweidimensionalen Platte berechnen wollen.

## Temperaturberechnung auf einer Platte (2D)

Wir können die Aufgabenstellung erweitern, indem wir keinen eindimensionalen Stab, sondern eine zweidimensionale Kochplatte betrachten, die vielleicht wiederum an bestimmten Stellen erhitzt wird.

Die Diskretisierung funktioniert ähnlich wie im obigen Fall, diesmal ergibt sich ein zweidimensionales Gitter von Punkten. An diesen Punkten soll die Temperatur berechnet werden. Für die Mittelung der Nachbartemperaturen an einem Punkt darf man jetzt nicht mehr nur die rechten und linken Nachbarn betrachten, sondern muss natürlich auch die oberen und unteren Nachbarn mit dazunehmen (s. Abb. 31.3).

Aufgrund dieser Darstellung der Abhängigkeiten eines Punktes von seinen Nachbarpunkten spricht man von dem *Stern*, den man auf den Punkt und

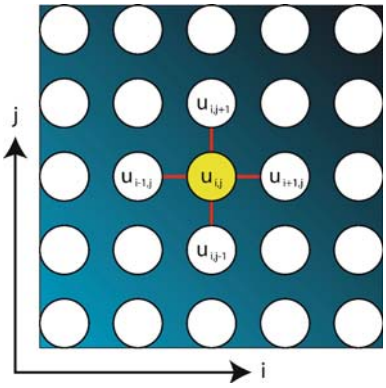



**Abb. 31.3.** Schematische Darstellung einer diskretisierten zweidimensionalen Platte und des Fünf-Punkt-Sterns

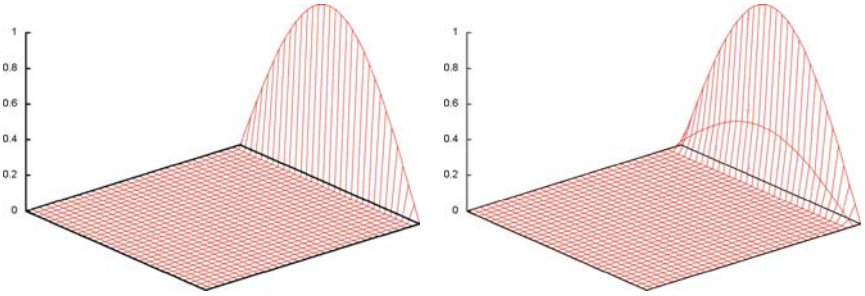
seine Umgebung anwenden muss, um seinen Wert neu zu berechnen. In diesem Fall handelt es sich also um einen Fünf-Punkt-Stern, und der neue Wert eines Punktes im Inneren der Platte berechnet sich zu

$$u_{i,j} := \frac{1}{4}\left(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}\right) + f_{i,j}$$

Der Algorithmus GaussSeidel2D

```
1  procedure GaussSeidel2D (n, u, f)
2  begin
3     for i := 2 to n − 1 do
4        for j := 2 to n − 1 do
5           u[i, j] := 1/4 (u[i − 1, j] + u[i + 1, j] + u[i, j − 1] + u[i, j + 1])
6                 +f[i, j]
7        endfor
8     endfor
9  end
```

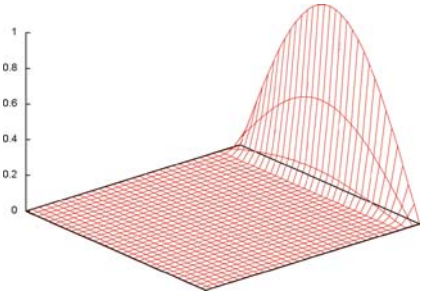
Betrachten wir nun eine quadratische Platte, deren linke untere Ecke sich im Punkt $(0, 0)$ und deren rechte obere Ecke sich im Punkt $(1, 1)$ befinden. Die Temperaturwerte an den Randpunkten der Platte sind festgelegt; am rechten Rand se i ein kurvenförmiger Temperaturverlauf, beschrieben durch die Funktion $\sin(\pi y)$, und an allen anderen Rändern die Temperatur Null vorgegeben. Dabei stellen wir die Temperatur als Wert in der dritten Dimension in Abhängigkeit von der Position auf der Platte dar. In diesem dreidimensionalen Bild sieht man also eine Landschaft, deren Höhe der Temperatur an diesem Punkt entspricht. Das erste Bild (Abb. 31.4(a)) zeigt damit eine Temperaturverteilung, bei der an allen Punkten eines Gitters mit $33 \times 33$ Punkten die Temperatur 0 ist, ausgenommen an dem rechten Rand. Dort ist eine Temperaturkurve vorgegeben.



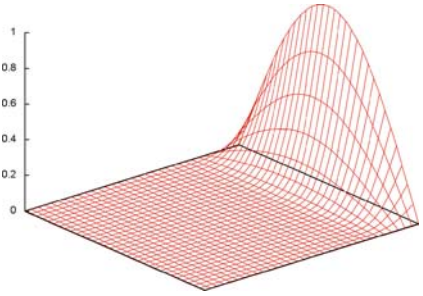
(a) Nach 0 Iterationen



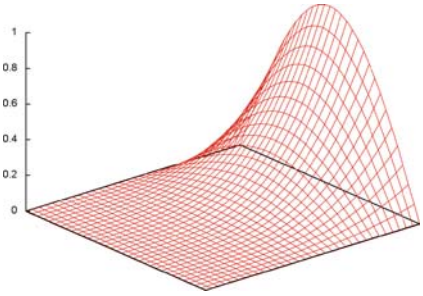
(b) Nach 1 Iteration



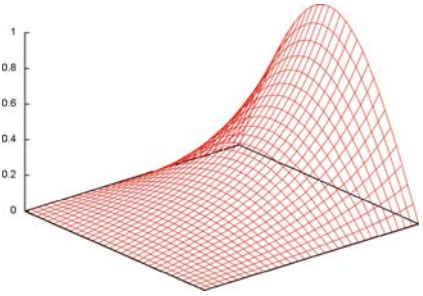
(c) Nach 2 Iterationen



(d) Nach 10 Iterationen



(e) Nach 100 Iterationen



(f) Nach 1000 Iterationen

**Abb. 31.4.** Verlauf der Gauß-Seidel-Iterationen

Dies entspricht nicht der richtigen Temperaturverteilung, die wir ja erst mit unserem Gauß-Seidel-Verfahren ausrechnen wollen. Am Ende muss eine glatte Temperaturverteilung herauskommen, die aber anders als im eindimensionalen Stab nicht mehr linear ist, selbst wenn es keine zusätzlichen Temperaturquellen gibt, wovon wir hier ausgehen.

Führt man den Algorithmus GaussSeidel2D einmal aus, so spricht man von einer ausgeführten *Iteration*. Dies drückt schon aus, dass man den Algorithmus mehrfach hintereinander anwenden muss, um eine Lösung zu erhalten. Lässt man also einige Gauß-Seidel-Iterationen laufen (Abb. 31.4(b)

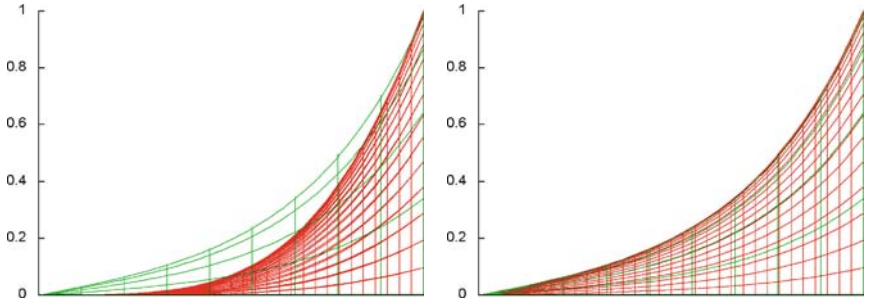
bis Abb. 31.4(f)), dann erkennt man, wie die am Rand vorgegebene Temperatur in das Innere der Platte hineinwandert, bis die Temperaturverteilung über der ganzen Platte schließlich schön glatt aussieht. Der Rechenaufwand ist übrigens nicht zu unterschätzen, denn jeder Durchlauf des Gauß-Seidel-Verfahrens muss ja an $31 \times 31 = 961$ Punkten den Durchschnitt von vier Zahlen berechnen. Bei tausend Durchläufen hat der Computer deshalb fast schon 5 Millionen Rechenoperationen zu bewältigen.

Auch wenn nach 100 Iterationen (Abb. 31.4(e)) die berechnete Lösung auf den ersten Blick schon recht gut erscheint, darf man das Verfahren an dieser Stelle noch nicht abbrechen, wie die Abb. 31.5 zeigen. Im ersten Bild (Abb. 31.5(a)) werden die exakte Lösung des Problems und die berechnete Lösung nach 100 Gauß-Seidel-Durchläufen übereinander gezeigt, so dass man schön sehen kann, dass das Ergebnis noch nicht wirklich stimmt. (Die exakte Lösung kann man in diesem Sonderfall mit mathematischen Formeln ausrechnen, sie lautet $u(x, y) = \frac{1}{\sinh \pi} \sinh(\pi x) \cdot \sin(\pi y)$, das ist die einzige Funktion, für die die vorgegebenen Randbedingungen erfüllt werden und für die die Summe der zweiten Ableitungen nach $x$ und $y$ Null ergibt.)

Erst nach ungefähr 1000 Durchläufen (Abb. 31.4(f) und Abb. 31.5(b)) zeigt die Überblendung keinen Unterschied zwischen exakter und berechneter Lösung mehr.

Auch wenn es so aussieht, als ob bei der Temperaturverteilung der zeitliche Fortschritt der Erhitzung dargestellt werden würde, ist dies nicht der Fall. Was hier berechnet wird, ist der Zustand, wenn das System bei der gegebenen Erhitzung im Gleichgewicht ist. Mit anderen, etwas komplizierteren Verfahren kann man auch berechnen, wie sich der zeitliche Verlauf der Erwärmung oder Abkühlung verhält.

Eine interessante Frage ist natürlich, wie viele Durchläufe des Gauß-Seidel-Verfahrens benötigt werden, um eine gute Näherung der physikalischen Lösung zu berechnen. Durch die Erfahrung (oder besser durch eine mathematische Analyse des Verfahrens) bekommt man heraus, dass bei einem Gitter mit $N \times N$ Punkten ungefähr $N \times N$ Durchläufe gemacht werden müssen. Weil an-




(a) Unterschied nach 100 Iterationen (b) Unterschied nach 1000 Iterationen

**Abb. 31.5.** Unterschiede zwischen angenäherter (rot) und exakter (grün) Lösung

dererseits die Temperatur in der Platte um so genauer wiedergegeben werden kann, je feiner die Gitterpunkte liegen, bekommt man schnell einen Rechenaufwand, der auch moderne PCs überfordert. Das gilt vor allem dann, wenn man nicht zweidimensionale Gebilde (wie die Platte), sondern dreidimensionale Objekte simulieren möchte, und zusätzliche physikalische Phänomene — wie bei der Wettervorhersage — die Berechnungen komplizierter machen. Dann braucht man nicht nur um ein Vielfaches teurere Supercomputer (mit Kosten in der Größenordnung von Millionen Euro), sondern auch erheblich verbesserte Algorithmen, die das gleiche Resultat viel schneller ausrechnen können.

Für diejenigen, die sich dafür interessieren, sind hier noch zwei Ideen, wie man das Verfahren beschleunigen kann: In den Bildern sieht man, dass die richtige Lösung von einer Seite angenähert wird, nämlich von unten. Anders gesagt, jeder Gauß-Seidel-Durchlauf bringt uns näher an die richtige Lösung, aber er geht nicht weit genug. Das kann man ausnützen, indem man in jeder der einzelnen Berechnungen die Änderung entsprechend vergrößert, d.h. mit einer Zahl größer 1 (aber kleiner 2) multipliziert. Diese Berechnungsweise ist das so genannte SOR-Verfahren (engl.: successive over relaxation). Die zweite Idee ist noch komplizierter und beruht darauf, dass Gitter mit verschiedenen Punktabständen kunstvoll zusammenarbeiten. Dieses so genannte Mehrgitterverfahren kommt dann mit einer sehr geringen Anzahl von Durchläufen aus und gilt als das schnellste Verfahren für diese Art von Problemen.

Zu guter Letzt sei gesagt, dass das Gauß-Seidel-Verfahren von dem berühmtesten aller Mathematiker, nämlich Carl Friedrich Gauß, 1823 erfunden wurde und von einem seiner Kollegen, Philipp Ludwig Seidel, in der Folge weiterentwickelt wurde. Damals wurde natürlich von Hand gerechnet. Gauß schreibt in einem Brief „Das Verfahren lässt sich halb im Schlaf ausführen oder man kann während dessen an andere Dinge denken.“ Wenn man das Verfahren heute programmiert, können auch wir an andere Dinge denken, während Computer die Rechenarbeit machen.

## Zum Weiterlesen

1. Kapitel 10 (PageRank)

   Der Artikel zum Page-Rank-Algorithmus zeigt, wie man schrittweise ein Gleichungssystem näherungsweise löst: Das ist genau das Gauß-Seidel-Verfahren, hier eben für ein anderes Gleichungssystem.

2. Das Gauß-Seidel-Verfahren als Algorithmus der Woche:
   `http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo39.php`

   In der Online-Version dieses Artikels gibt es Java-Applets, die die Durchführung des Verfahrens veranschaulichen.

3. `http://de.wikipedia.org/wiki/Gauß-Seidel-Verfahren`

   Im Wikipedia-Artikel findet man eine exakte mathematische Beschreibung des Gauß-Seidel-Verfahrens für beliebige Gleichungssysteme.

4. Video des Vortrags „Simulieren geht über Probieren. Virtuelle Welten mit Supercomputern“:
   `http://giga.rrze.uni-erlangen.de/movies/collegium_alexandrinum/ss06/20060518-Ruede-XVid.avi`
   (MPEG-4-Video, für andere Formate s. Online-Version dieses Artikels)

   Der Vortrag zeigt Simulationen auf Höchstleistungsrechnern und warum diese ein wichtiges Werkzeug für die Wissenschaft sind.

5. “Why Multigrid Methods Are So Efficient” von Irad Yavneh:
   `http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1717310`

   Dieser Artikel, erschienen in der Zeitschrift *Computing in Science & Engineering*, erklärt die Idee der oben erwähnten Mehrgitterverfahren, erfordert aber auch etwas mehr Mathematik zum Verständnis.

6. TOP500 – die Liste der 500 schnellsten Computer in der Welt:
   `http://www.top500.org/`

   Diese Liste wird zwei Mal im Jahr vorgestellt. Auf ihr befinden sich die Computer, auf denen man die Lösung wirklich großer Probleme berechnen kann.

7. `http://de.wikipedia.org/wiki/Carl_Friedrich_Gauß`
   `http://de.wikipedia.org/wiki/Ludwig_Seidel`

   Diese Artikel beschreiben in Kürze das Leben der beiden deutschen Mathematiker Carl Friedrich Gauß und Philipp Ludwig von Seidel, auf die das vorgestellte Rechenverfahren zurückgeht.

# 32

# Dynamische Programmierung: Evolutionäre Distanz

Norbert Blum und Matthias Kretschmer

Rheinische Friedrich-Wilhelms-Universität Bonn

Vor über 150 Jahren wurden nahe bei Düsseldorf im Neandertal erstmals Skelettteile des so genannten Neandertalers gefunden. Seit diesem Fund stellte sich u.a. die Frage, inwiefern der Homo sapiens, von dem wir abstammen, und der Neandertaler miteinander verwandt sind. Mittlerweile haben Wissenschaftler herausgefunden, dass die Entwicklungslinie des Homo neanderthalensis sich vor ca. 315.000 Jahren von jener trennte, die schließlich zum Homo sapiens führte. Dies hat man aufgrund der Unterschiede im Erbgut des Homo sapiens und des Homo neanderthalensis festgestellt. Neue Verfahren ermöglichen signifikante Anteile des Erbguts aus den mehr als 30.000 Jahre alten Knochen zu extrahieren. Dieses erhält man in Form von DNA-Sequenzen, die man sich als Baupläne für Tiere und Menschen vorstellen kann. Während der evolutionären Geschichte haben sich DNA-Sequenzen aufgrund von Mutationen geändert. Hat man DNA-Sequenzen verschiedener Spezies, dann kann man die Ähnlichkeit dieser Sequenzen mit Hilfe des Computers berechnen. Wir messen die Ähnlichkeit zweier Sequenzen, indem wir beide zeichenweise vergleichen. Das Ergebnis dieses Vergleiches bezeichnen wir als Distanz zwischen den beiden Sequenzen. Umso ähnlicher zwei DNA-Sequenzen sind, umso geringer ist die Distanz zwischen diesen.

Wie kann man einen Algorithmus entwickeln, der die Distanz zweier DNA-Sequenzen berechnet?

Mit dieser Frage wollen wir uns zunächst beschäftigen. Hierzu benötigen wir als erstes ein mathematisches Modell. Wir werden DNA-Sequenzen und Mutationen modellieren, um die Distanz zweier Sequenzen formal definieren zu können.

## Mathematische Modellierung

Eine DNA-Sequenz setzt sich aus Folgen von Basen zusammen, die die Aminosäuren kodieren. Dazu werden immer drei aufeinander folgende Basen benötigt. Es gibt vier verschiedene Basen, die mit den Zeichen $A$, $G$, $C$ und

$T$ bezeichnet werden. Eine DNA-Sequenz kann somit als eine Folge aus den Zeichen $A$, $G$, $C$ und $T$ dargestellt werden. Man sagt, eine DNA-Sequenz ist ein String (Zeichenkette) über dem Alphabet $\Sigma = \{A, G, C, T\}$. So ist z. B.

$$CAGCGGAAGGTCACGGCCGGGCCTAGCGCCTCAGGGGTG$$

ein Ausschnitt aus der DNA-Sequenz des Huhnes. In der Natur verändern sich DNA-Sequenzen durch Mutationen. Eine *Mutation* kann mathematisch als eine Überführung einer DNA-Sequenz $x$ in eine DNA-Sequenz $y$ modelliert werden. Wir gehen davon aus, dass es nur drei verschiedene Mutationen, die wir auch Operationen nennen werden, vorkommen können:

1. Streichen eines Zeichens,
2. Einfügen eines Zeichens und
3. Ersetzen eines Zeichens durch ein anderes.

Wenn z. B. $x = AGCT$, dann würde die Mutation Ersetzen von $G$ durch $C$ die DNA-Sequenz $x$ in die DNA-Sequenz $y = ACCT$ transformieren. Die Position, an der ein Zeichen gestrichen, eingefügt oder ersetzt wird, ist, wie wir später sehen werden, aus dem Kontext direkt ersichtlich. Darum werden wir die Position nicht explizit angeben. Wir verwenden $a \rightarrow b$, um das Ersetzen des Zeichens $a$ durch das Zeichen $b$ darzustellen. $a \rightarrow$ bezeichnet das Streichen des Zeichens $a$ und $\rightarrow b$ das Einfügen des Zeichens $b$.

Um ein Maß für die Distanz zweier DNA-Sequenzen zu erhalten, ordnen wir jeder Mutation $s$ Kosten $c(s)$ zu. Die Kosten einer Mutation korrespondieren zu der Wahrscheinlichkeit, dass diese Mutation auftritt. Je wahrscheinlicher die Mutation umso geringer die Kosten. Wir ordnen den drei verschiedenen Mutationen folgende Kosten zu:

- Streichen: Kosten 2
- Einfügen: Kosten 2
- Ersetzen: Kosten 3

Wenn wir zwei verschiedene DNA-Sequenzen $x$ und $y$ miteinander vergleichen, dann benötigen wir in der Regel mehrere Mutationen, um $x$ in $y$ zu überführen. Wählen wir z. B. $x = AG$ und $y = T$, dann sehen wir, dass eine einzelne Mutation $x$ nicht nach $y$ transformieren kann. Wir sehen aber auch, dass die Mutationenfolge $S = A \rightarrow, G \rightarrow T$ (Streichen von $A$ und Ersetzen von $G$ durch $T$) $x$ nach $y$ transformiert. Wir definieren die Kosten $c(S)$ einer Mutationenfolge $S = s_1, \ldots, s_t$ als die Summe der Kosten der einzelnen Mutationen. Als Formel ausgedrückt erhalten wir

$$c(S) := c(s_1) + \cdots + c(s_t).$$

Wir wollen die Kosten einer Mutationenfolge verwenden, um die Distanz zweier DNA-Sequenzen zu definieren. Dabei müssen wir beachten, dass es mehrere verschiedene Mutationenfolgen geben kann, die eine DNA-Sequenz $x$ in eine DNA-Sequenz $y$ transformieren. Im obigen Beispiel $x = AG$ und $y = T$ gibt es unter anderem folgende Möglichkeiten:

- $S_1 = A \to, G \to T$; $c(S_1) = c(A \to) + c(G \to T) = 2 + 3 = 5$
- $S_2 = A \to T, G \to$; $c(S_2) = c(A \to T) + c(G \to) = 3 + 2 = 5$
- $S_3 = A \to, G \to, \to T$; $c(S_3) = c(A \to) + c(G \to) + c(\to T) = 2 + 2 + 2 = 6$
- $S_4 = A \to C, G \to, C \to, \to T$;
  $c(S_4) = c(A \to C) + c(G \to) + c(C \to) + c(\to T) = 3 + 2 + 2 + 2 = 9$

Es gibt sehr viele weitere Mutationenfolgen, die allerdings alle keine niedrigeren Kosten haben als $S_1$ und $S_2$. Wir verwenden zur Bestimmung der Distanz einfach eine Mutationenfolge mit den geringsten Kosten. Wir definieren für unsere Kostenfunktion $c$ die Distanz $d_c(x, y)$ zweier Sequenzen $x$ und $y$ durch

$$d_c(x, y) := \min\{c(S) \mid S \text{ transformiert } x \text{ nach } y\}.$$

Im obigen Beispiel ist die Distanz $d_c(x, y) = 5$, da $S_1$ und $S_2$ die Mutationenfolgen mit den geringsten Kosten sind, die $x$ nach $y$ transformieren.

## Berechnung der evolutionären Distanz

Wie berechnet man nun die evolutionäre Distanz? Das heißt, wie berechnen wir $d_c(x, y)$? Wir haben nur die Operationen Streichen, Einfügen und Ersetzen zur Verfügung, um $x$ nach $y$ zu transformieren. Die Operationen und das Kostenmodell sind so definiert, dass eine Folge von mehreren Operationen an derselben Position kostengünstiger durch eine einzelne Operation ersetzt werden kann. So kann z. B. das Streichen eines Zeichens $a$ und das Einfügen eines Zeichens $b$ an derselben Position durch das Ersetzen von $a$ durch $b$ kostengünstiger durchgeführt werden (Kosten $2 + 2 = 4$ für eine Streiche- und eine Einfüge-Operation gegenüber Kosten 3 für eine Ersetze-Operation). Mutationen an verschiedenen Positionen beeinflussen sich gegenseitig nicht. Wir können somit Mutationen in beliebiger Reihenfolge durchführen. Wir gehen nun davon aus, dass die letzte Mutation immer an den letzten Positionen der beiden Sequenzen stattfindet.

Seien $x = a_1 a_2 \dots a_m$ und $y = b_1 b_2 \dots b_n$. Das heißt, $x$ besteht aus $m$ und $y$ aus $n$ Zeichen. Gemäß unseren drei Operationen gibt es drei Möglichkeiten, $x$ nach $y$ zu transformieren:

1. *Streichen*: Zunächst wird $a_1 a_2 \dots a_{m-1}$ nach $b_1 b_2 \dots b_n$ transformiert. Dann wird $a_m$ gestrichen.
2. *Einfügen*: Zunächst wird $a_1 a_2 \dots a_m$ nach $b_1 b_2 \dots b_{n-1}$ transformiert. Dann wird $b_n$ eingefügt.
3. *Ersetzen*: Zunächst wird $a_1 a_2 \dots a_{m-1}$ nach $b_1 b_2 \dots b_{n-1}$ transformiert. Dann wird $a_m$ durch $b_n$ ersetzt.

Für die Berechnung der evolutionären Distanz nehmen wir die kostengünstigste der drei Varianten.

Wir wollen jetzt obige Beobachtung zu einem Algorithmus ausarbeiten. Sei nun $x[i]$ die Zeichenkette bestehend aus den ersten $i$ Zeichen von $x$. Diese

Zeichenkette nennt man auch *Präfix der Länge i* von $x$. Das Präfix der Länge 0 von $x$ ist der leere String $x[0]$. Das Präfix der Länge $m$ von $x$ ist $x$ selber, da $x$ gerade $m$ Zeichen lang ist. Entsprechend ist $y[j]$ das Präfix der Länge $j$ von $y$. Wir können nun die obige Beobachtung wie folgt umformulieren:

1. $x[m-1]$ wird nach $y$ transformiert und dann $a_m$ gestrichen.
2. $x$ wird nach $y[n-1]$ transformiert und dann $b_n$ eingefügt.
3. $x[m-1]$ wird nach $y[n-1]$ transformiert und dann $a_m$ durch $b_n$ ersetzt.

Jetzt stellt sich die Frage: Wie transformieren wir $x[m-1]$ nach $y$, $x$ nach $y[n-1]$ und $x[m-1]$ nach $y[m-1]$? Hierzu können wir dasselbe Schema verwenden. Hierzu wenden wir diese Betrachtungsweise auf beliebige Präfixe $x[i]$ und $y[j]$ an. Zur Bestimmung der evolutionären Distanz $d_c(x[i], y[j])$ von $x[i]$ und $y[j]$ benötigen wir somit die evolutionären Distanzen $d_c(x[i-1], y[j])$, $d_c(x[i], y[j-1])$ und $d_c(x[i-1], y[j-1])$. Wir erhalten dann $d_c(x[i], y[j])$ durch

$$d_c(x[i], y[j]) := \min \begin{cases} d_c(x[i-1], y[j]) + c(a_i \rightarrow) & \text{(Streichen)}, \\ d_c(x[i], y[j-1]) + c(\rightarrow b_j) & \text{(Einfügen)}, \\ d_c(x[i-1], y[j-1]) + c(a_i \rightarrow b_j) & \text{(Ersetzen)} \end{cases}$$

und haben somit das Problem der Berechnung von $d_c(x[i], y[j])$ in drei neue Teilprobleme überführt, wobei allerdings die Gesamtlänge der beiden Sequenzen jeweils abgenommen hat.

Falls einer der Präfixe die Länge Null hat, dann können nicht alle drei obigen Alternativen auftreten. Das rührt daher, dass wir in einem leeren String kein Zeichen streichen und auch keines ersetzen können. Um aus $x[i]$ den leeren String $y[0]$ mit minimalen Kosten zu erhalten, werden wir niemals ein Zeichen einfügen oder ersetzen. Jedes Zeichen, welches eingefügt wird, muss auch wieder gestrichen werden. Das bedeutet, dass eine Einfüge-Operation immer zusätzliche Kosten von 2+2 (Einfügen und Streichen) verursacht und somit nicht in einer Lösung minimaler Kosten vorkommen kann. Entsprechend kann man eine Ersetze-Operation einfach weglassen. Es macht keinen Unterschied, welches Zeichen gelöscht wird, da die Kosten für eine Streiche-Operation nicht von dem zu streichenden Zeichen abhängen. Somit erhöhen sich die Kosten durch die Verwendung einer Ersetze-Operation und diese Operation kann nicht in einer Lösung minimaler Kosten auftreten. Ähnlich verhält es sich, wenn man den leeren String $x[0]$ in den String $y[j]$ mit minimalen Kosten transformieren möchte. Jede Ersetze- oder Streiche-Operation ist mit zusätzlichen Kosten verbunden. Konkret bedeutet dies, dass für $i = 0$ und $j > 0$ nur eine Einfüge-Operation und für $j = 0$ und $i > 0$ nur eine Streiche-Operation in Frage kommen. Im Fall $i = 0$ und $j = 0$ müssen wir den leeren String $x[0]$ in den leeren String $y[0]$ transformieren. Dafür benötigen wir keine einzige Operation, da beide Strings identisch sind. Somit ergibt sich $d_c(x[0], y[0]) = 0$.

Betrachten wir das Beispiel $x = AGT$ und $y = CAT$. Nehmen wir an, wir hätten $d_c(x[1], y[2]) = d_c(A, CA)$, $d_c(x[2], y[1]) = d_c(AG, C)$ und

$d_c(x[1], y[1]) = d_c(A, C)$ bereits bestimmt. Dann können wir daraus die evolutionäre Distanz von $d_c(x[2], y[2]) = d_c(AG, CA)$ nach obigem Muster berechnen. Dazu ermitteln wir das Minimum von

- $d_c(x[1], y[2]) + c(a_2 \rightarrow) = d_c(A, CA) + c(G \rightarrow) = d_c(A, CA) + 2$,
- $d_c(x[2], y[1]) + c(\rightarrow b_2) = d_c(AG, C) + c(\rightarrow A) = d_c(AG, C) + 2$ und
- $d_c(x[1], y[1]) + c(a_2 \rightarrow b_2) = d_c(A, C) + c(G \rightarrow A) = d_c(A, C) + 3$.

Da wir $d_c(A, CA)$, $d_c(AG, C)$ und $d_c(A, C)$ bereits vorher bestimmt haben, können wir somit $d_c(AG, CA)$ berechnen.

## Der Algorithmus

Wie können wir nun aus diesen Überlegungen einen Algorithmus entwickeln? Die evolutionären Distanzen von Präfixen von $x$ und $y$ werden mehrfach benötigt. So wird z. B. $d_c(x[i-1], y[j-1])$ zur Berechnung von $d_c(x[i], y[j-1])$, $d_c(x[i-1], y[j])$ und $d_c(x[i], y[j])$ verwandt. Wir möchten die Berechnung von $d_c(x[i-1], y[j-1])$ nur einmal durchführen. Demzufolge müssen wir diesen Wert speichern, so dass wir ihn mehrfach verwenden können. Wir ziehen hierzu eine Tabelle heran, in der wir die bisher berechneten evolutionären Distanzen zwischenspeichern. In dieser Tabelle verwenden wir die Zelle $(i, j)$ (Zeile $i$ und Spalte $j$) zur Speicherung der evolutionären Distanz $d_c(x[i], y[j])$. Der Vorteil dieser Methode ist, dass es wesentlich effizienter ist, die Werte direkt aus der Tabelle zu lesen, als sie jedes Mal erneut zu berechnen. Wir benötigen die evolutionären Distanzen für alle $0 \leq i \leq m$ und $0 \leq j \leq n$. Also besteht unsere Tabelle aus $m + 1$ Zeilen und aus $n + 1$ Spalten.

Seien z. B. $x = ATGAACG$ und $y = TCAAT$. Die dazugehörige Tabelle, bei Verwendung unserer Kostenfunktion $c$ (Streichen und Einfügen kosten 2, Ersetzen kostet 3), ist in der Abbildung auf der nächsten Seite gegeben. Wir fangen bei der Berechnung der Distanzen mit $d_c(x[0], y[0])$ an. Wie wir uns oben überlegt haben, gilt $d_c(x[0], y[0]) = 0$. Also notieren wir uns in der Zelle $(0, 0)$ den Wert 0. Wenn wir die Spalte 0 von oben nach unten durchgehen, dann können wir jeweils nur eine Streiche-Operation durchführen. Das haben wir oben bereits bemerkt ($j$ ist in diesem Fall gleich 0). Wir haben uns bereits überlegt, dass wir in diesem Fall nur Zeichen streichen, da wir $x[i]$ in den leeren String $y[0]$ überführen. Somit ergeben sich die Kosten $0, 2, 4, 6, \ldots$ für die erste Spalte. Entsprechend können wir in der Zeile 0 nur Einfüge-Operationen durchführen. Somit ergeben sich die Kosten $0, 2, 4, 6, \ldots$ für die erste Zeile.

Die anderen Tabelleneinträge berechnen sich nun nach der oben entwickelten Formel. Betrachten wir z. B. die Zelle $(2, 1)$. Wir wollen den String $x[2] = AT$ in den String $y[1] = T$ transformieren. Wir sehen direkt, dass es mit einer Streiche-Operation erledigt ist. Wir müssen nur $A$ entfernen. Das ist auch intuitiv die Lösung mit den geringsten Kosten. Der Algorithmus muss demnach auch diese Distanz (eine Streiche-Operation: Kosten = 2) berechnen.

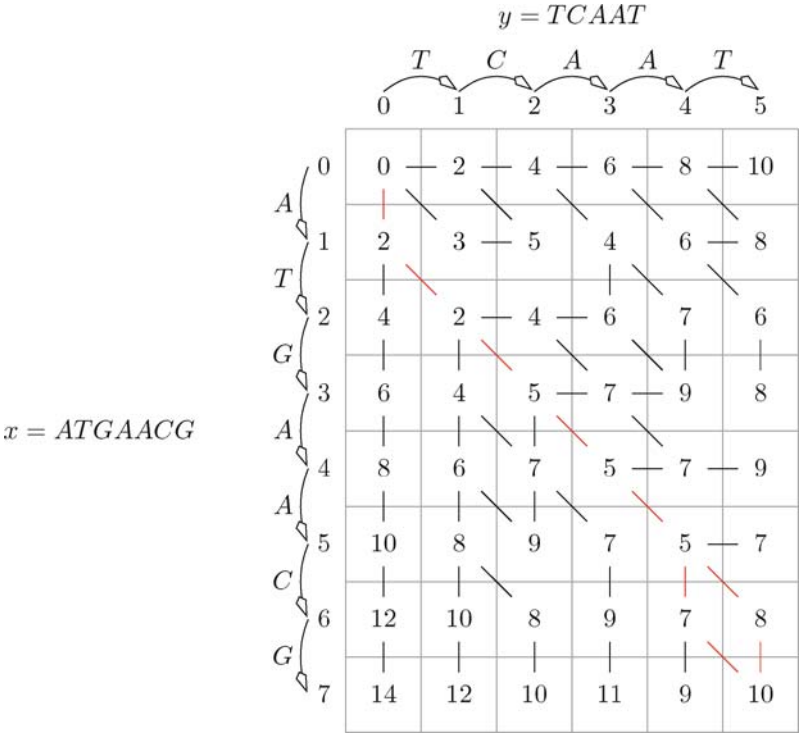



**Abb. 32.1.** Tabelle zur Berechnung der evolutionären Distanz von $x = ATGAACG$ und $y = TCAAT$

Dies ist auch der Fall, denn er bildet das Minimum aus den drei Werten, die den drei Operationen entsprechen:

1. $d_c(x[1], y[1]) + c(a_2 \rightarrow) = d_c(A, T) + c(a_2 \rightarrow) = 3 + 2 = 5$ (Streichen von $T$),
2. $d_c(x[2], y[0]) + c(\rightarrow b_1) = d_c(AT, y[0]) + c(\rightarrow b_1) = 4 + 2 = 6$ (Einfügen von $T$) und
3. $d_c(x[1], y[0]) + c(a_2 \rightarrow b_1) = d_c(A, y[0]) + c(a_2 \rightarrow b_1) = 2 + 0 = 2$ (Ersetzen von $T$ durch $T$).

Die Operation Ersetzen von $T$ durch $T$, ist keine Mutation und wird nur der Einfachheit halber verwendet. In Wirklichkeit ersetzen wir $T$ nicht durch $T$, sondern tun nichts. Daher betragen die Kosten hierfür 0. Die Operation Streiche $A$ ist hier nicht explizit aufgeführt. Dies liegt daran, dass diese Operation implizit bei dem Schritt $x[1] = A$ nach $y[0]$ zu transformieren enthalten ist. Dieser wird durch den Tabelleneintrag $(1, 0)$ repräsentiert, der bereits vorher erfolgt ist.

Die kleinen Striche innerhalb der Tabelle sollen verdeutlichen, über welche „Wege“ eine optimale Transformation von $x[i]$ nach $y[j]$ ablaufen kann. In dem Fall der Zelle $(2,1)$ sind wir gerade diagonal von $(1,0)$ über eine Ersetzen-Operation zu den minimalen Kosten gekommen. Diese Wege sind nicht notwendigerweise eindeutig, wie man anhand der Tabelle sehen kann. Die Striche können direkt bei dem Aufbau der Tabelle bestimmt werden, da sie nur angeben, welche der drei möglichen Schritte an den letzten Positionen der Präfixe zu minimalen Kosten führen.

Die rot markierten Kanten geben die „Wege minimaler Kosten“ für die Transformation von $x$ nach $y$ an. Das heißt, sie geben die Wege an, über die man mit minimalen Kosten $x$ nach $y$ transformieren und somit die evolutionäre Distanz bestimmen kann.

Da wir zwischendurch nicht wissen, welche Tabelleneinträge zu einem „Weg minimaler Kosten“ für die Transformation von $x$ nach $y$ beitragen, müssen wir alle bestimmen. Wir benötigen zur Berechnung des Eintrages der Zelle $(i,j)$ die Werte der Zellen $(i-1,j)$, $(i,j-1)$ und $(i-1,j-1)$. Um sicherzustellen, dass diese evolutionären Distanzen bereits berechnet sind, wenn wir $d_c(x[i],y[j])$ in die Tabelle eintragen wollen, bauen wir die Tabelle zeilenweise von oben nach unten oder spaltenweise von links nach rechts auf. Wenn wir die Tabelle zeilenweise aufbauen, dann müssen wir jede Zeile von links nach rechts durchlaufen. Entsprechend muss jede Spalte beim spaltenweisen Aufbau von oben nach unten durchlaufen werden. Am Ende steht dann in Zelle $(m,n)$ das gewünschte Ergebnis, die evolutionäre Distanz $d_c(x[m],y[n])$ von $x[m]=x$ und $y[n]=y$.

## Zusammenfassung

Beginnend bei dem einfachsten und kleinsten Teilproblem, der Berechnung von $d_c(x[0],y[0])$, haben wir immer größer werdende Teilprobleme gelöst. Dabei haben wir die Längen der Präfixe von $x$ und $y$ immer größer werden lassen und für diese die evolutionäre Distanz berechnet. Zur Bestimmung von $d_c(x[i],y[j])$ wurden die drei evolutionären Distanzen $d_c(x[i-1],y[j])$, $d_c(x[i],y[j-1])$ und $d_c(x[i-1],y[j-1])$ benötigt. Somit haben wir optimale Lösungen kleiner Teilprobleme zur optimalen Lösung von größeren Teilproblemen verwendet.

Besteht die Aufgabe darin, für ein gegebenes Problem eine Lösung minimaler Kosten zu berechnen, dann bezeichnet man das Problem auch als ein *Optimierungsproblem*. Somit ist das hier betrachtete Problem „Berechnung der evolutionären Distanz zweier DNA-Sequenzen“ ein Optimierungsproblem. Es stellt sich nun die Frage, unter welchen Voraussetzungen die oben entwickelte Methode auch zur Lösung von anderen Optimierungsproblemen verwendet werden kann.

Implizit haben wir bei der Entwicklung unseres Verfahrens folgende Eigenschaft unseres Optimierungsproblems angewandt:

- Jede Teillösung einer optimalen Lösung, die Lösung eines Teilproblems ist, ist selbst eine optimale Lösung des betreffenden Teilproblems.

Viele Optimierungsprobleme besitzen diese Eigenschaft, die häufig auch *Optimalitätsprinzip* genannt wird. Wenn für ein Optimierungsproblem das Optimalitätsprinzip gültig ist, dann kann zur Lösung dieses Problems die oben entwickelte Methode, die so genannte *dynamische Programmierung* angewandt werden. Bei der dynamischen Programmierung teilen wir unser Problem in Teilprobleme auf. Die kleinsten Teilprobleme lösen wir direkt. In unserem Fall war dies gerade die Berechnung von $d_c(x[0], y[0])$. Die Lösung hierfür war sehr einfach, da immer $d_c(x[0], y[0]) = 0$ gilt. Aus den optimalen Lösungen von kleinen Teilproblemen werden dann optimale Lösungen für größere Teilprobleme berechnet. Dies wird solange wiederholt, bis man eine optimale Lösung für das eigentliche Problem berechnet hat.

Die dynamische Programmierung ist ein wichtiges allgemeines Verfahren, das beim Algorithmenentwurf häufig seine Anwendung findet.

Das hier vorgestellte Verfahren kann nicht nur zur Berechnung der evolutionären Distanz zweier DNA-Sequenzen verwendet werden, sondern auch, um die Ähnlichkeit zweier Wörter der deutschen Sprache zu bestimmen. Dies wird unter anderem bei Rechtschreibkorrekturprogrammen angewendet. Zum Beispiel wird dem Benutzer jedes Wort, dass eine Distanz kleiner oder gleich fünf zu einem dem Rechtschreibkorrekturprogramm unbekannten Wort hat, als Alternative für dieses unbekannte Wort angeboten.

## Zum Weiterlesen

Die angegebenen Kapitel aus den hier vorgestellten Büchern geben eine allgemeine Einführung in die Thematik der dynamischen Programmierung. Dort wird die Anwendung dieses Lösungsverfahren theoretisch und anhand von Beispielen erläutert.

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms, Second Edition*. MIT Press, 2001. Kap. 3.
2. Jon Kleinberg, Éva Tardos: *Algorithm Design*. Addison-Wesley, 2005. Kap. 6.

# 33

# Faires Teilen: Eine Weihnachtsstollengeschichte

Raimund Seidel

Universität des Saarlandes

„Die Tage nach Weihnachten sind am coolsten." Katja lungert auf dem Sofa und spielt mit ihren Haaren herum. „Die Erwachsenen sind vom Weihnachtsstress ganz erschöpft und lassen einen in Ruhe. Man muss nicht in die Schule, kann in der Früh ausschlafen und sonst faulenzen."

„Stimmt", sagt ihr Bruder Holger, „aber es wird auch schnell langweilig. Draußen nur Regenwetter. Fast alle Freunde im Urlaub. Was soll man denn da noch tun? Gut, dass wenigstens Gert noch da ist!"

Gert hockt am Boden und arrangiert die Fransen am Teppichrand. „Ja", sagt der, „Schnee wäre wirklich besser als Regen. Was gibt's denn im Fernsehen?"

„Immer nur Fernsehen! Euch Jungs fällt auch nichts anderes ein.", motzt Katrin und ärgert sich still über die Bücher, die sie zu Weihnachten bekommen hat. Für Bibi Blocksberg ist sie nun ja wirklich schon zu alt.

„Wo ist denn schon wieder die Fernbedienung?" Holger sucht Tisch und Regal ab. „Die kann doch nicht einfach weg sein!"

„Ich glaube, die ist in der Küche", meint schließlich Katja. „Die liegt gleich neben dem Toaster."

„Na, hoffentlich nicht im Toaster", murmelt Holger und geht in die Küche. „Das wäre einmal was Neues."

„Du, Katja", ruft er, „was ist denn das für ein Kuchen auf dem Tisch?"

„Das ist der Pochelsteiner Weihnachtsstollen – von Tante Hedwig. Du weißt doch, sie bringt jedes Jahr einen."

„Kuchen?", stöhnt Gert und macht mit einer Handbewegung sein Teppichfransenarrangement kaputt. „Jetzt, nach Advent und Weihnachten, mag ich schon das Wort kaum noch hören."

„Nein, nein, Gert", sagt Holger und kommt mit einer Kuchenplatte zurück. Darauf liegt der ganze Stollen. Er ist nicht besonders groß, aber sieht appetitlich aus und verströmt einen feinen Duft. „Nein, Gert. Tante Hedwigs Pochelsteiner Weihnachtsstollen ist was ganz Besonderes. Da hätte sich sogar die Prato noch was abschneiden können."

„Wer, wie, was abschneiden?“, fragt Gert verwundert. „Ach, ignorier ihn!“ rät Katja. „Seitdem er im Sommer in Österreich war, redet Holger manchmal ganz wirres Zeug.“

Die drei stehen um den Kuchen herum. Der Duft löst selbst bei Gert gewaltigen Kuchenhunger aus. „Also sehr groß ist der ja leider nicht“, meint Holger. „Ein ordentliches Stück für jeden, und der ist weg.“ Er verschwindet in die Küche und kommt mit einem Messer zurück.

„Holger, du hättest schon auch Kuchenteller mitbringen können“, mahnt Katja vorwurfsvoll. „Du weißt doch, wie fröhlich Mama wird, wenn wir hier im Wohnzimmer alles vollkrümeln.“

„Wir können ja nachher staubsaugen.“

„Du und staubsaugen! Das möchte ich sehen!“ schnaubt Katja und holt Kuchenteller.

„Und wie kriegen wir den Kuchen in drei Teile?“ fragt jetzt Gert und greift sich an die Brille.

„Ja mit dem Messer natürlich“, antwortet Holger.

„Nein, ich meine, wie kriegen wir ihn in drei gleich große Teile? Wir wollen doch fair teilen, oder?“

Nach kurzer Stille sagt Katja: „Wir könnten doch mit einem Zentimetermaß messen, wie lang der Stollen ist. Dann teilen wir das durch drei – . Ach, ich sehe schon, das wird nichts. Der Stollen ist ja dünner an den beiden Enden und dicker in der Mitte.“

„Ja, aber wir könnten eine Waage verwenden“, wirft Gert ein.

„Und wie soll das helfen?“, fragt Holger. „Wir schneiden den Stollen in drei Stücke und wiegen sie? Die sind dann sicher nicht alle gleich schwer, also müssen wir vom schwersten Stück wieder etwas wegschneiden und an die anderen Stücke verteilen. Aber dann haben wir vielleicht zu viel weggeschnitten und wir müssen diesen Fehler korrigieren, was wieder zu einem neuen Fehler führt, und so fort. Am Schluss haben wir dann wahrscheinlich nur noch Krümel und keinen Kuchen. Schade um den guten Stollen!“

„Ja, müssen denn die drei Teile wirklich genau gleich schwer werden?“, fragt Gert.

„Natürlich!“, antwortet Holger. „Wenn du wirklich fair teilen willst, dann müssen doch die drei Teile gleich schwer werden. Alles andere wäre doch unfair!“

„Unser alter Nachbar“, beginnt Katja nach einiger Zeit, „der Herr Maier, der hat mir einmal erzählt, wie er im Krieg lang mit einem Kameraden unterwegs war und sie nur mehr wenig zu essen hatten. Die haben ihr Brot dann immer so geteilt, dass der Eine das Stück in zwei Hälften geschnitten hat und der Andere sich dann die Hälfte ausgesucht hat. So musste sich der Eine beim Schneiden immer bemühen, möglichst gleich große Hälften zu erzeugen, weil ihm sonst nach der Wahl des Anderen nur die kleinere Hälfte übriggeblieben wäre.“

„Kleinere Hälfte!“, stänkert Holger, „wenn unser Mathelehrer das hört, sagt er immer: ‚So ein Blödsinn! Es gibt keine kleinere Hälfte. Hälften sind per definitionen gleich groß!‘“

„Du weißt schon, was ich meine“, antwortet Katja. „Und übrigens heißt es per definitioneM“ korrigiert sie dazu.

„DefinitioneM, DefinitioneN, DefinitionENG! Mir egal!“, entfährt es Holger. „Schwesterchen, was soll denn diese Kriegsgeschichte überhaupt? Wir wollen doch den Stollen in drei Teile schneiden und nicht in zwei.“

„Aber vielleicht können wir diese Idee verwenden“, wirft Gert ein. „Ich könnte doch den Stollen in drei Stücke schneiden. Dann sucht Katja sich ein Stück aus, dann du, Holger, und ich muss das Stück nehmen, das übrig bleibt.“

„Klingt doch gut“, sagt Katja.

„Nein, nein, nein!“, erwidert ihr Bruder. „Nehmen wir einmal an, Gert ist in dich verknallt. Dann schneidet er den Stollen in ein großes Stück und zwei kleinere. Du kannst dir dann das große Stück nehmen. Ich bekäme nur ein kleines, was mir gar nicht recht wäre. Und er kriegt natürlich auch nur ein kleines Stück; aber wenn er in dich verknallt ist, ist das dem galanten Helden ja egal. Hauptsache, er hat dir ein großes Stück zugespielt.“

Gert errötet unmerklich. „Vielleicht sollte dann Holger zuerst ein Stück nehmen, nachdem ich geschnitten habe“, meint er.

„Und wie kann ich dann sicher sein, dass du nicht ihm ein großes Stück zuschanzst? Damit du dir seinen neuen Super-MP3-Spieler ausborgen kannst, oder sonst etwas?“, erwidert Katja.

„Vielleicht sollte Katja schneiden, und Holger nimmt sich als Erster ein Stück, dann ich und dann Katja“, sagt Gert.

„Aber dann bekommen wir doch nur die gleichen Misstrauensprobleme in grün“, antwortet Holger, „beziehungsweise in pink“ und schaut auf Katjas neuen Pullover. „Diese Idee aus der Kriegsgeschichte funktioniert wohl nicht bei drei Leuten.“

Die drei schweigen eine Weile und starren auf den Pochelsteiner Stollen. „Ich hab’s!“, ruft Katja auf einmal. „Wir bitten einfach Papa, dass er den Stollen möglichst fair in drei Stücke teilt.“

Holger rollt die Augen: „‚This is the dumbest thing I have ever heard!‘, würde da Bill Gates sagen. Papa würde doch wahrscheinlich zuerst für sich ein Stück abzweigen. Da kriegen wir dann alle weniger.“

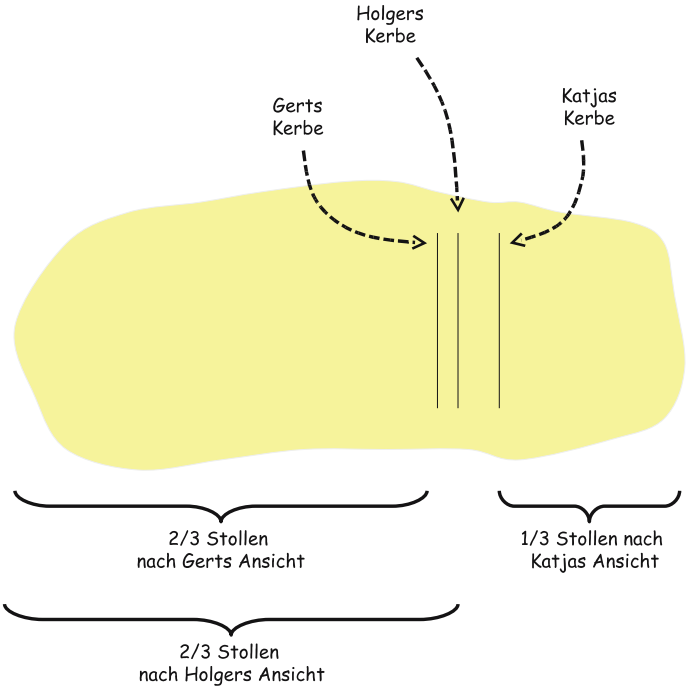
„Da hast du auch wieder recht“, sagt Katja kleinlaut.

Wieder ist es eine Zeit lang still. Dann sagt Holger: „Jetzt krieg ich aber wirklich Hunger.“ Er fuchtelt mit dem Messer herum. „Ich glaube, wenn ich hier abschneide, bekomme ich genau ein Drittel des Stollens.“ Er führt die Klinge zum Stollen, zieht sie noch einmal kurz zurück, setzt sie auf und möchte zu schneiden beginnen.

„HALT!!“, rufen Katja und Gert wie aus einem Munde. Holger hält inne und zieht das Messer zurück. Eine deutliche Kerbe ist jetzt am Stollenrücken zu erkennen.

„Das Stück wird doch zu groß“, sagt Katja. „Gib das Messer her! Ich glaube, ein Drittel des Stollens kriegst du genau hier!“, und ritzt den Kuchen fast einen Zentimeter näher am Ende.

„Eigenartig“, sagt Gert. „Ich finde, dass Holger das Drittelstück zu klein machen würde. Ich würde hier abschneiden.“ Er nimmt das Messer und ritzt den Stollenrücken ein paar Millimeter näher zur Mitte von Holgers Kerbe aus gesehen.




„Na toll!“, klagt Katja. „Drei Personen, drei Meinungen. So kommen wir nie auf einen grünen Zweig.“

„Ja, aber der Kuchen wird vielleicht noch grün vor Schimmel, bis wir uns da auf irgendetwas geeinigt haben“, ätzt Holger.

„Du bist ein Ekel“, erwidert seine Schwester.

„Ich hab’s! Ich hab’s, ich hab’s!“, ruft auf einmal Gert. Die Geschwister schauen ihn verdutzt an. „Wir schneiden den Stollen einfach irgendwo zwischen Katjas und Holgers Kerben.“, fährt Gert aufgeregt fort.

„Und was soll das bringen?“, fragen die Geschwister.

„Wir geben dieses Stück einfach Katja.“

„Aber dann bekommt sie doch mehr als ein Drittel des Stollens!“, ruft Holger empört.

„Aus ihrer Sicht schon“, antwortet Gert, „und das wird sie wohl fair finden, oder, Katja?“ Sie nickt. „Aus unserer Sicht bekommt sie aber weniger als ein Drittel.“, fährt Gert fort. „Das heißt, aus unserer Sicht bleiben mehr als zwei

Drittel des Stollens für uns beide übrig. Dieses große Stück können wir dann fair in zwei Hälften teilen, so wie es euer Maier-Nachbar im Krieg gemacht hat, und damit bekommen Holger und auch ich jeder ein Stück, das wir jeder für mehr als ein Drittel des Stollens halten. Das ist doch auch für uns fair, Holger, oder?“

Holger kratzt sich am Kopf und denkt nach. „Das ist wirklich raffiniert, Gert“, sagt er schließlich. „Jeder von uns macht eine Kerbe in den Kuchen, dort wo er glaubt, dass ein Drittel auf einer Seite der Kerbe liegt und zwei Drittel auf der anderen Seite. Bekommt er allein die Seite mit dem Drittel oder mehr, dann kann er happy sein, denn er hat mindestens so viel bekommen, wie er glaubt, dass ihm zusteht. Bekommt er mit einem anderen zusammen die Seite mit den zwei Dritteln oder mehr, dann kann er auch happy sein, denn zu zweit haben sie mindestens soviel bekommen, wie sie glauben, dass ihnen zusammen mindestens zusteht. Sie müssen nur noch diesen gemeinsamen Teil fair in zwei Hälften teilen.“

„Und der Trick ist, dass durch den Schnitt zwischen der ersten und zweiten Kerbe jedem von uns einer der beiden Fälle beschert wird. Am Schluss bekommt dann sogar jeder von uns drei mehr, als er glaubt, dass ihm zusteht. Das grenzt schon an Zauberei!“

Gert strahlt, stolz auf seine Idee. „Es könnte aber auch sein, dass wir alle drei genau die gleiche Kerbe machen“, wirft er ein. „Dann bekommt keiner echt mehr, als er glaubt, dass ihm zusteht.“

„Richtig!“, sagt Katja. „Kommt, lasst uns jetzt endlich den Stollen schneiden und essen!“

Holger nimmt das Messer. „Also wie war das? Ich muss zwischen der ersten und zweiten Kerbe irgendwo schneiden, vom Kuchenende gerechnet?“

„Ja“, antwortet Katja, „aber warte, ich glaube, es hat an der Tür geläutet.“

Sie springt auf und läuft aus dem Wohnzimmer. Kurz darauf kommt sie mit ihrer Cousine Sandra zurück.

„Hallo Jungs!“, tönt sie, „und nachträglich fröhliche Weihnachten!“

„Hallo Sandra“, grüßen sie zurück. „Long time, no see.“

„Euer Englisch ist wirklich umwerfend. Oh, der Kuchen sieht aber gut aus. Krieg ich ein Stück?“

Die drei schauen einander entgeistert an. „Ach Sandra“, sagt Katja schließlich, „jetzt haben wir uns ganz schlau überlegt, wie wir den Stollen ganz fair in drei Teile schneiden können, so dass keiner glaubt, dass er zu kurz kommt, und jetzt platzt du herein.“

Sandra schaut interessiert: „Ganz schlau und ganz fair, in drei Teile? Erzählt einmal, wie geht das!“

Katja legt los und erzählt ganz stolz die eben ausgeheckte faire Schnittmethode für drei. Sandra hört aufmerksam zu und ist beeindruckt. „Und es bekommt wirklich jeder sogar mehr, als er glaubt, dass ihm zusteht?“, fragt sie schließlich.

„Wenn wir drei verschiedene Kerben machen, dann sicher“, antwortet Katja. „Aber jetzt ist das Ganze ja wertlos, denn jetzt sind wir vier, und wie man fair in vier Teile schneidet, das wissen wir nicht.“

„Noch nicht! Noch nicht!“, sagt Sandra. „Aber ich glaube, ich weiß schon, wie man das machen könnte.“

„Erzähl!“, erwidern die drei erstaunt. Das ging ein wenig schnell.

„Also“, beginnt Sandra, „es ist ja eigentlich ganz einfach“.

„Das sagt unser Mathelehrer auch immer, und dann kommt meist etwas ganz Unverständliches“, bemerkt Gert.

„Ganz einfach unverständlich?“ ergänzt Holger.

„Genau! Ganz einfach unverständlich.“ blödelt Gert.

„Also hört mal zu“, fährt Sandra fort, „es ist wirklich ganz einfach. Jeder von uns Vieren macht eine Kerbe in den Stollen, dort wo er glaubt, dass man genau ein Viertel des Stollens abschneiden würde. Und dann zählen wir vom Stollenende weg die Kerben, und zwischen der ersten und zweiten Kerbe machen wir einen Schnitt. Dieses abgeschnittene Stück bekommt dann der, der die erste Kerbe gemacht hat. Denn der bekommt dann mehr, als was er für ein Viertel hält, und das ist aus dessen Sicht wohl fair.“

„Das heißt, man muss sich merken, wer welche Kerbe gemacht hat?“, wirft Katja ein.

„Natürlich!“, antwortet Sandra. „Und für die anderen drei, die das erste Viertel nicht bekommen haben, erscheint es auch fair, denn aus deren Sicht wurde weniger als ein Viertel des Stollens weggeschnitten, es bleibt also mehr als drei Viertel übrig.“

„Was machen die dann mit den drei Vierteln oder mehr?“, fragt Holger.

„Aber das ist doch jetzt klar“, sagt Gert. „Sie verwenden die faire Dreiteilungsmethode, die wir uns gerade ausgedacht haben, und teilen so die drei Viertel oder mehr, in dreimal jeweils ein Viertel oder mehr. Das ist ja wirklich einfach!“

„Am Schluss hat dann auch wieder jeder mehr als er glaubt, dass er eigentlich bekommen sollte. Das ist wirklich verrückt.“, ergänzt Katja.

„Also kommt, lasst uns endlich diesen Weihnachtsstollen teilen und essen, bevor ihn die Ameisen davontragen.“, sagt Holger ungeduldig.

Das tun sie dann auch. Es dauert zwar eine Weile, bis da alles richtig, so wie geplant geteilt wird (und es würde auch schneller gehen, wenn sich Holger seine Kerben merken könnte), aber schließlich sitzt jeder bei seinem Stück Pochelsteiner Weihnachtssstollen aus Tante Hedwigs Küche und genießt.

„Ich habe irgendwo gelesen“, sagt Holger mampfend, „dass sie in Erfurt oder so irgendwo einen ganz langen Weihnachtsstollen gebacken haben, zehn Meter waren es, glaube ich. Ob man den auch fair aufteilen kann? Auf, sagen wir, hundert Leute?“

„Du hast wieder Sorgen!“, sagt Katja.

„Nein! Ich finde die Frage interessant“, meint Sandra. „Und ich glaube, ich weiß auch, wie man das machen kann“, fährt sie fort, „zumindest theoretisch.“

„Ohje. ‚Theoretisch', wenn ich das schon höre!", stöhnt Katja. „Und ich weiß auch gar nicht, wie du uns das heute noch erklären könntest. Bis du da jeden Schnitt für jedes der hundert Stücke erläutert hast, ist es ja schon nach Mitternacht."

„Nein. Das kann ich schnell erklären.", erwidert Sandra. „Es ist ja auch wieder ganz einfach."

„Ohja.", sagt Gert aufgeregt. „Ich glaube, ich weiß auch, wie das gehen kann. Ich werd's erklären."

„Aber fall doch Sandra nicht ins Wort, Gert!", mahnt Katja.

„Entschuldigung!", murmelt der und schaut ein bisschen betroffen.

„Also", beginnt Sandra, „da haben wir den langen Stollen und hundert Leute. Jeder schaut sich den Stollen genau an und überlegt sich, wo er glaubt, dass man abschneiden müsste, um seiner Meinung nach genau ein Hundertstel des Stollens zu bekommen. Natürlich müssen die das alles vom gleichen Ende des Stollens aus tun. Dann macht jeder eine Kerbe in den Stollen und merkt sie sich."

„Na toll", spottet Holger. „Wie soll denn das gehen? Vor lauter Kerben wird der Stollen doch um die Stelle in Krümeln zerfallen, und keine Kerbe wird mehr sichtbar sein, ganz zu schweigen davon, wie man sich da seine Kerbe merken können soll."

„Ja, ich weiß.", sagt Sandra. „Deswegen habe ich auch ‚theoretisch' gesagt. Und außerdem könnten sie ja statt Kerben zu schneiden so Zahnstocherfähnchen an den entsprechenden Stellen in den Stollen stecken."

„Da können sie gleich ihre Namen auf die Fähnchen schreiben, und keiner muss sich mehr was merken", ergänzt Gert.

„Richtig!", fährt Sandra fort. „Jeder hat also sein Fähnchen in den Stollen gesteckt, dort wo er meint, dass der richtige Schnitt für ein Hundertstel des Stollens wäre. Dann zieht man vom Ende des Stollens los und macht zwischen dem ersten und dem zweiten Fähnchen einen Schnitt. Der, dessen Name auf dem ersten Fähnchen steht, bekommt dieses Stück."

„Vom welchen Ende des Stollens sollen denn die Fähnchen gezählt werden?", stänkert Holger. „Da könnte jemand ja ein Riesenstück bekommen, über neun Meter."

„Hör auf, Brüderchen!", sagt Katja. „Du weißt schon, was gemeint ist. Die müssen sich natürlich auf ein Stollenende geeinigt haben, wo sie ein Hundertstel abschneiden wollen. Und das ist dann auch das Ende, von dem das Fähnchenzählen beginnt."

„Genau!", fährt Sandra fort. „Der mit dem ersten Fähnchen hat also sein Stück bekommen, und das ist größer als, was er für ein Hundertstel des Stollens gehalten hatte. Für die 99 anderen ist der Stollen ihrer jeweiligen Meinung nach um weniger als ein Hundertstel kleiner geworden, es sind also ihrer jeweiligen Meinung nach noch mehr als 99 Hundertstel des Stollens übrig, genug um für jeden der 99 ein Hundertstel des ursprünglichen Stollens zu liefern."

„Also, der erste Schnitt war fair.", fällt nun Gert ein. „Keiner fühlt sich übervorteilt, und eine Person hat ein Hundertstel des Stollens bekommen,

beziehungsweise, was sie für ein Hundertstel oder mehr hält. Und was jetzt mit den 99 anderern passiert, ist auch klar."

„Ja", sagt wieder Sandra. „Das große Stollenstück, von dem jeder der 99 glaubt, dass es mehr als 99 Hundertstel des Riesenstollens ausmacht, muss jetzt fair in 99 Stücke geteilt werden."

„Und wie soll das wieder gehen?", fragt Katja ein bisschen abwesend.

„Rekursiv, Katja, Rekursiv mit der gleichen Methode. Ist doch klar!"

„Wie? *‚Rekursiv'*?", fragt Katja. „Das Wort habe ich schon ein paarmal im Informatikunterricht gehört. Und da hat das für mich nie Sinn gemacht."

„Schau, Katja!", sagt Sandra. „Wir wissen jetzt, wie man, wenn man 100 Personen und einen Stollen hat, vom Stollen ein Stück abschneiden kann, sodass eine Person mit ihrem Stück zufrieden ist und die 99 anderen mit dem großen Stück, das sie gemeinsam haben, zufrieden sind. Sie müssen dieses große Stück fair in 99 Teile schneiden. Und das machen sie nach der gleichen Methode: Jeder schätzt, wo er glaubt, dass man ein 99-stel des Stollens abschneiden würde, und steckt dort sein Fähnchen hinein. Dann wird wieder vom Ende gezählt und ein Schnitt zwischen dem ersten und zweiten Fähnchen gemacht. Das kurze Stück geht dann an den Besitzer des ersten Fähnchens, der damit mehr bekommt als er für ein 99-stel gehalten hatte. Für die übrigen 98 ging ihrer Meinung nach weniger als ein 99-stel verloren. Sie sind zufrieden, wenn das große Reststück fair in 98 Teile geteilt wird. Muss ich dir noch erklären, wie das geschieht, Katja?"

„Nein, danke, Sandra. Ich glaube, ich habe es kapiert.", sagt sie. „Das funktioniert wieder nach der gleichen Methode mit 98 Fähnchen, und einer bekommt ein 98-stel Stück, und die restlichen 97 müssen sich den Reststollen fair in 97 Stücke teilen. Das geht wieder mit 97 Fähnchen und so weiter, und so weiter. Und das ist, was Rekursion bedeutet?"

„Richtig", antwortet Sandra. „Beziehungsweise auch richtig. In diesem Fall wollen wir einen Stollen fair in $x$ Teile schneiden, also zum Beispiel in $x = 100$ Teile, aber $x = 72315$ wäre auch möglich, und wir tun das, indem wir es schaffen, dass einer seinen fairen Teil bekommt, und wir dazu noch das neue Problem bekommen, einen Stollen fair in $x - 1$, also 99, Teile zu schneiden. Und das geht wieder mit dem gleichen Schema und so fort."

Katja schaut ganz erstaunt. „Ich glaube, mir ist ein Licht aufgegangen, Sandra. Das mit der Rekursion ist wirklich nicht so schwierig. Zumindest in diesem Fall hier."

Inzwischen hat jeder sein Stollenstück aufgegessen. Holger sagt etwas schlapp: „Aber eines gefällt mir bei dieser Stollenaufteilmethode nicht. Wenn ich mir das so vorstelle, wie sie den 10 Meter langen Riesenstollen fair auf 100 Personen aufteilen wollen, dann braucht das irgendwie zu lange. Schon beim ersten Durchgang, bis sich da alle Hundert überlegt haben, wo sie ihr Fähnchen genau hinstecken wollen, das braucht schon einige Minuten. Und im nächsten Durchgang sind es noch immer 99 Personen."

„Lass mich das einmal überschlagen!", fällt ihm Gert ins Wort. „Sagen wir, jeder braucht nur eine Sekunde, um sich den Platz für sein Fähnchen zu suchen

und es in den Stollen zu stecken. Alles andere geht, sagen wir, blitzschnell und braucht keine Zeit. Beim ersten Durchgang müssen 100 Fähnchen gesteckt werden, braucht 100 Sekunden, beim zweiten Durchgang müssen 99 Fähnchen gesteckt werden, braucht 99 Sekunden, beim dritten Durchgang braucht man 98 Sekunden, und so fort. Insgesamt braucht man also 100+99+98+97+ . . . +3+2+1 Sekunden. Das sind 5050 Sekunden, also fast eineinhalb Stunden."

„Woher weißt du das mit den 5050 so schnell?", fragt Katja.

„Ich habe halt aufgepasst und mir die Formel gemerkt: die Summe der Zahlen von 1 bis $n$ ist $n(n+1)/2$. Da haben sie doch diese Story vom kleinen Klaus erzählt."

„Du meinst den kleinen Gauß.", fällt ihm Holger ein wenig verärgert ins Wort. „Fürs faire Teilen in 100 Stücke braucht man also schon eineinhalb Stunden. Wieviel das wohl bei 1000 Stücken wäre?"

„Ungefähr 500000 Sekunden!", erwidert Gert trotzig. „Das ist, glaube ich, fast eine ganze Woche. Da verhungere ich ja, bevor ich mein Stollenstück bekomme."

„Ja, oder der Stollen vergammelt.", meint Holger. „Das faire Teilen muss doch auch irgendwie schneller gehen."

Sie bleiben ruhig. Gert hat wieder begonnen, mit den Teppichfransen herumzuspielen.

Plötzlich sagt Sandra: „Ich glaube, ich habe eine Idee, wie man schneller fair in viele Teile teilen könnte. Im Informatikunterricht haben wir etwas von *‚Teile-und-Herrsche'* gehört. Man müsste den Stollen irgendwie fair in zwei große Teile schneiden, ein Teil für die einen 50 Personen, den anderen Teil für die anderen 50. Die beiden Gruppen könnten dann gleichzeitig und unabhängig voneinander weitermachen."

„Rekursiv.", wirft Katja ein.

„Ja, rekursiv, aber ein wenig anders als vorher."

„Und wie soll das faire Teilen in zwei große Teile für jeweils 50 Personen genau funktionieren mit diesem *‚Teile-und-Herrsche'*?", fragt Holger. „Eigentlich sollte es hier wohl *‚Teile-und-Esse'* heißen."

„Ich glaube", sagt Katja schnell, „bei dir, Brüderchen, sollte es wohl eher *‚Teile-nicht-und-Esse'* heißen."

Bevor Holger etwas sagen kann, geht die Tür auf. Die Mutter kommt herein und fragt: „Wo ist denn Tante Hedwigs Kuchen hin? Den wollte ich jetzt dem Herrn Maier hinüberbringen?"

## Fragen

- Wenn man so einen Stollen fair in zwei Teile schneiden möchte, gibt es da außer der „Nachbar-Maier"-Methode noch eine andere?
- Bei der *‚Teile-und-Herrsche'*-Idee, die Sandra vorschlägt, muss bei, sagen wir, 100 Personen Folgendes passieren: Die Leute müssen in zwei Gruppen

von je 50 geteilt werden, und der große Stollen muss in zwei Teile geschnitten werden, für jede Gruppe einen. Das soll so geschehen, dass jede der beiden Gruppen mit ihrem Stollenteil zufrieden ist. Wie kann man das erreichen?

- Könnte man bei der vorherigen Fragestellung auch so teilen, dass es eine Gruppe von 49 Personen gibt, und eine von 51, und beide Gruppen fühlten sich fair behandelt?
- Ist sogar jede beliebige Personenaufteilung fair behandelbar?
- Wenn Sandras *‚Teile-und-Herrsche‘*-Idee rekursiv für 100 Personen durchgeführt würde, wie lange würde es dauern, bis jede Person ihr Stollenstück bekommt? (Zähle dabei nur das Stecken von Fähnchen mit jeweils einer Sekunde.)

## Zum Weiterlesen

Das Problem des fairen und gerechten Teilens stellt sich in vielen Variationen: Soll zwischen zwei Parteien geteilt werden oder zwischen mehreren Parteien? Ist das zu teilende Gut beliebig zerlegbar wie der Stollen in dieser Geschichte, oder besteht es aus letztendlich unzerlegbaren Stücken, wie wenn z. B. acht Meerschweinchen auf drei Kinder aufgeteilt werden sollen. Was soll man tun, wenn das zu teilende Gut nicht überall gleich ist, wenn also beispielsweise der Stollen an einem Ende mehr Rosinen enthält? Eine große Übersicht über Probleme dieser Art gibt das englischsprachige Buch *Cake-Cutting Algorithms: Be Fair if You Can* von Jack Robertson und William Webb.[1]

Ein deutschsprachiger Artikel über eine besondere Variante, nämlich das sogenannte „Neidfreie Teilen“ findet sich in der Ausgabe der Zeitschrift *Spektrum der Wissenschaft* vom Dezember 1996.

In der deutschen Version von Wikipedia findet man einiges unter dem Stichwort „Scheidungsformel“, und in der englischen Version einiges unter „Cake cutting“.

Rekursion und auch der sogenannte „Teile-und-Herrsche“-Ansatz (englisch *divide-and-conquer*) sind grundlegende Methoden der Informatik. Sie tauchen auch in einigen anderen Kapiteln dieses Buches auf, wie z. B. in den Kap. 11 (Multiplikation langer Zahlen) und 3 (Schnelle Sortieralgorithmen).

[1] Jack Robertson und William Webb: *Cake-Cutting Algorithms: Be Fair if You Can*. A K Peters Ltd, 1998 (ISBN: 1-56881-076-8).

# Teil IV

# Optimieren

# Übersicht

Heribert Vollmer und Dorothea Wagner

Universität Hannover
Universität Karlsruhe

Wie kann man unter allen möglichen Verbindungen von einer Stadt zu einer anderen die kürzeste finden? In welcher Reihenfolge muss man eine Folge von Städten anfahren, damit die entstehende Rundreise durch die gewünschten Ziele möglichst kurz wird? Im abschließenden Teil dieses Buches geht es um Aufgabenstellungen, bei denen aus einer im Allgemeinen sehr großen Anzahl „möglicher Lösungen“ eine in gewissem Sinne „optimale Lösung“ gefunden werden soll – in der Informatik nennen wir solche Aufgabenstellungen *Optimierungsprobleme*.

Wir werden sehen, dass es für viele Optimierungsprobleme clevere Verfahren gibt, die sehr schnell („effizient“) die beste Lösung finden. Das oben genannte *Kürzeste-Wege-Problem* gehört dazu. In Kapitel 34 wird ein Lösungsalgorithmus für diese und verwandte Aufgaben vorgestellt. Für alle Optimierungsprobleme, die in den ersten sieben Kapiteln des letzten Teils dieses Buches behandeln werden, hat man effiziente Algorithmen gefunden: In Kapitel 35 sollen Inseln mit Brücken verbunden werden, damit man von jeder Insel zu jeder anderen gelangen kann. Insgesamt soll natürlich möglichst wenig Brückenstrecke verbaut werden. Kapitel 36 zeigt, wie der Autoverkehr so auf die unterschiedlich breiten Straßen einer Stadt verteilt werden kann, dass möglichst wenig Stau entsteht und der Verkehr fließt. Dies ist also ein Beispiel für so genannte „Flussprobleme“, die aus der heutigen Informatik nicht mehr wegzudenken sind. Die Aufgabe einer Partnerschaftsagentur, die heiratswillige Herren und Damen zueinander führt, wird in Kapitel 37 zumindest theoretisch gelöst. In Kapitel 38 soll für eine neue Feuerwehr ein Standort gefunden werden, der für die Häuser, für die die Feuerwehr zuständig ist, optimal gewählt ist.

Bei den letzten beiden Optimierungsproblemen, für die eine effiziente Lösung vorgestellt wird, handelt es sich um spezielle Problemvarianten, die *Online-Probleme*, bei denen sich die präzise Aufgabestellung erst im Laufe der Zeit nach und nach ergibt: In Kap. 39 geht es darum zu entscheiden, ob für einen Skiurlaub Skier gekauft oder geliehen werden sollen, obwohl nicht bekannt ist, ob und wie oft in den kommenden Jahren Skier wieder benötigt

werden. Ähnlich sollen in Kap. 40 bei einem Umzug möglichst wenige Umzugskartons gepackt werden, wobei sich erst im Laufe der Zeit ergibt, welche Gegenstände eingepackt werden müssen.

Für viele andere Optimierungsprobleme ist bis heute kein effizientes Lösungsverfahren bekannt. Alle bislang verwendeten Algorithmen durchsuchen – etwas vereinfacht dargestellt – den gesamten Lösungsraum nach der optimalen Lösung. Da hierbei also (fast) jede Lösung einmal konstruiert und angesehen werden muss, hängt die Effizienz von der Anzahl aller möglichen Lösungen ab. Dies ist bei den meisten Optimierungsproblemen eine Größe, die sehr schnell astronomische Ausmaße annimmt. In Kap. 41 wird diese Problematik anhand des *Rucksackproblems* erläutert – dabei geht es darum, wie die zur Verfügung stehende Kapazität eines Rucksacks optimal genutzt werden kann.

Das oben angesprochene Problem, eine kürzeste Rundreise zu bestimmen, das so genannte *Travelling Salesman Problem*, gehört auch zu diesen widerspenstigen Aufgabenstellungen, die sich bislang jeder effizienten Lösung versperren. Bei diesem Problem geht es darum, eine kürzeste Rundreise zu bestimmen, wie oben bereits angesprochen. In Kap. 42 wird ein effizienter *Approximationsalgorithmus* für dieses Problem vorgestellt: Dabei handelt es sich um ein Verfahren, das zwar nicht immer die beste Lösung findet, aber zumindest eine Rundreise bestimmt, deren Länge nur um einen garantierten Faktor länger ist als eine optimale Rundreise. Der in Kap. 42 vorgestellte Algorithmus ist speziell für das Travelling Salesman Problem entworfen worden. Bei dem Verfahren *Simulated Annealing* aus dem abschließenden Kap. 43 handelt es sich dagegen um eine Methode aus der Physik, die für verschiedene Optimierungsprobleme mit bestimmten Eigenschaften auf beinahe magische Weise eine Näherungslösung findet.

# 34

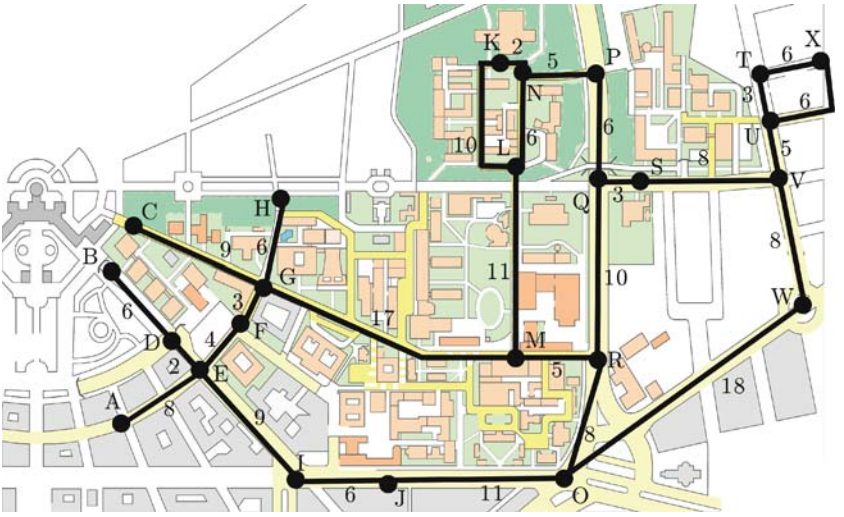
# Kürzeste Wege

Peter Sanders und Johannes Singler

Universität Karlsruhe

Ich bin gerade nach Karlsruhe in meine erste Studentenbude gezogen. So eine Großstadt ist schon ziemlich kompliziert! Einen Stadtplan habe ich zwar schon, aber wie finde ich heraus, wie ich am schnellsten von A nach B komme? Ich fahre zwar gerne Fahrrad, aber ich bin notorisch ungeduldig, deshalb brauche ich den wirklich kürzesten Weg – zur Uni, zu meiner Freundin ...

Ich gehe die Planung also systematisch an. So könnte es gehen: Ich breite den Stadtplan auf einem Tisch aus und lege dünne Fäden entlang der Straßen. Die Fäden werden an Kreuzungen und Abzweigungen miteinander verknotet. Der Vollständigkeit halber mache ich außerdem Knoten an allen möglichen Start- und Zielpunkten sowie an den Enden von Sackgassen.

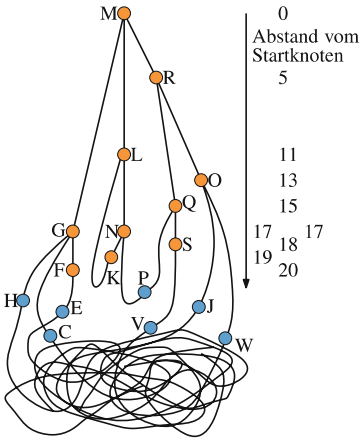



Copyright Universität Karlsruhe (TH), Institut für Photogrammetrie und Fernerkundung

Jetzt kommt der Trick: Ich fasse den Startknoten, und ziehe ihn langsam nach oben. Nach und nach löst sich das Gespinst von der Tischplatte. Ich habe die Knoten beschriftet, so dass ich immer noch weiß, welcher Knoten wohin gehört. Schließlich hängt alles senkrecht unter dem Startknoten.

Nun habe ich leichtes Spiel: Um einen kürzesten Weg von meinem Startknoten zu einem Ziel zu finden, muss ich nur den Zielknoten ausfindig machen und straff gespannte Fäden nach oben bis zum Startknoten zurückverfolgen. Der Abstand zwischen Start- und Zielknoten lässt sich außerdem einfach mit einen Maßband abmessen. Der so gefundene Weg muss der kürzeste sein, denn gäbe es einen noch kürzeren, hätten sich Start- und Zielknoten nie so weit voneinander entfernt (ohne Fäden zu zerreißen).

Nehmen wir z. B. an, ich möchte den kürzesten Weg von der Mensa (M) zum Rechenzentrum (F) herausfinden. Ich hebe das Netz also am Knoten M immer weiter an, wobei sich Knoten für Knoten vom Tisch löst. Die Abbildung unten zeigt die Situation in dem Moment, wo Knoten F das erste Mal in der Luft hängt. Zur Verbesserung der Übersichtlichkeit sind die Knoten da ein wenig horizontal auseinander gezogen. Die orangenen Knoten hängen in der Luft. Die Zahlen rechts geben den Abstand von M an, wenn man die Fadenlängen aus der ersten Abbildung verwendet.

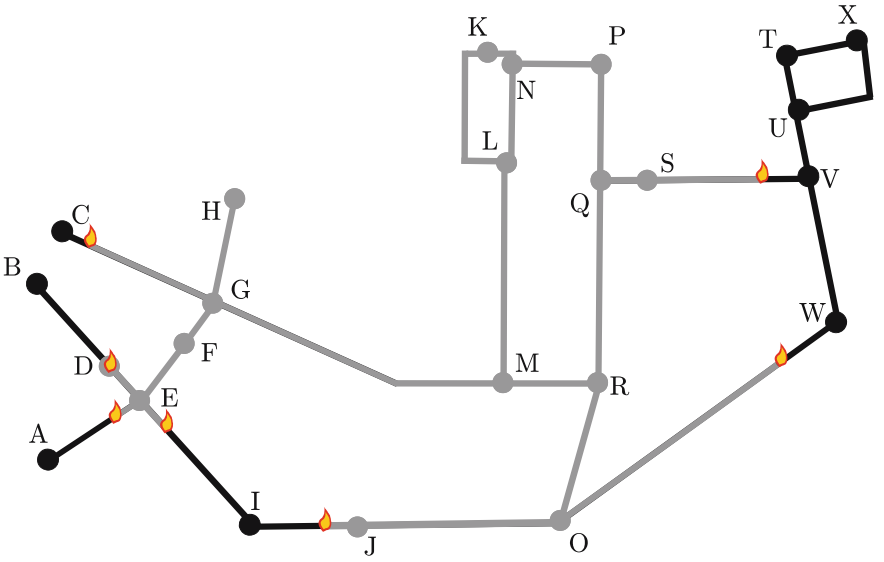



Man erkennt, dass der kürzeste Weg von M nach F über G führt. Zwischen L und K hat sich bereits eine durchhängende Schlaufe gebildet, die sich auch nicht mehr spannen wird. Das bedeutet, dass kein kürzester Weg von M aus über diese Verbindungen läuft.

Ich habe das Ganze also für die Umgebung meiner Uni erfolgreich getestet. Aber schon mein erster Versuch mit ganz Karlsruhe ist kläglich an verhedderten Bindfäden gescheitert. Ich habe die halbe Nacht gebraucht, um die Fäden zu entwirren und erneut auf dem Stadtplan auszulegen.

Am nächsten Tag ist mein kleiner Bruder zu Besuch. Kein Problem, meint er: „Ich habe die nötige Technologie, um das Problem zu lösen.“ Er packt sei-

nen Chemiebaukasten aus und tränkt die Fäden mit einer mysteriösen Flüssigkeit. O weh! Mein Bruder zündet das Gespinst am Startpunkt an. In den nächsten Sekunden verschwindet das Zimmer in einer Rauchwolke. Dieser Pyromane hat aus den Fäden Zündschnüre gemacht. Er erklärt mir voller Stolz, was das soll: Alle Fäden brennen gleich schnell ab. Also ist der Zeitpunkt, zu dem ein Knoten Feuer fängt, proportional zur Entfernung vom Startpunkt. Außerdem enthält die Richtung, aus der ein Knoten Feuer fängt, die gleiche Information wie die gespannten Fäden bei meinem Ansatz mit dem hängenden Gespinst. Toll! Leider hat er vergessen, eine Videoaufnahme des Infernos zu machen. Jetzt haben wir nur einen Haufen Asche. Selbst mit einem Video müßte ich für einen weiteren Versuch mit einem anderen Startpunkt von vorne anfangen. Unten sieht man eine Momentaufnahme, nachdem die Flammen von Startpunkt M aus schon einige Teilstrecken (grau) verbrannt haben.




Nachdem ich meinen Bruder hochkant aus meiner Bude geworfen habe, beginne ich nachzudenken. Ich muss meine Scheu vor Abstraktion überwinden und das Ganze meinem dämlichen Computer verklickern. Das hat aber auch Vorteile: Fäden, die es nicht gibt, können sich nicht verheddern oder abbrennen. Mein Prof hat mir gesagt, dass ein Herr Dijkstra einen Algorithmus, der das Kürzeste-Wege-Problem ganz ähnlich wie das Bindfadenverfahren löst, schon 1959 aufgeschrieben hat. Netterweise kann man den Algorithmus von Dijkstra in der Bindfadenterminologie beschreiben.

### Der Algorithmus von Dijkstra

Im Wesentlichen geht es darum, den Bindfadenalgorithmus zu simulieren. Eine Implementierung muss zu jedem Knoten die von ihm ausgehenden Fäden zu

benachbarten Knoten und deren Länge kennen. Es verwaltet außerdem eine Tabelle $d$, welche die Distanz vom Startknoten abschätzt. Die Entfernung $d[v]$ ist höchstens so groß wie die Entfernung vom Startknoten zum Knoten $v$. Genauer bezeichnet $d[v]$ die Länge der kürzesten Verbindung vom Startknoten zu Knoten $v$, die zwischendurch nur *hängende* Knoten verwendet. Solange es noch keine „hängende Verbindung“ gibt, ist $d[v]$ dagegen unendlich. Damit gilt am Anfang, wenn es noch keine hängenden Knoten gibt, $d[\text{Startknoten}] = 0$ und $d[v] =$ unendlich für alle anderen Knoten. Wir bezeichnen im Folgenden Knoten als *wartend*, wenn sie noch nicht hängen, aber ein endliches $d[v]$ haben.

Der hier angegebene Pseudocode beschreibt die Berechnung der Entfernungen aller Knoten vom Startknoten:

Algorithmus von Dijkstra in Bindfadenterminologie

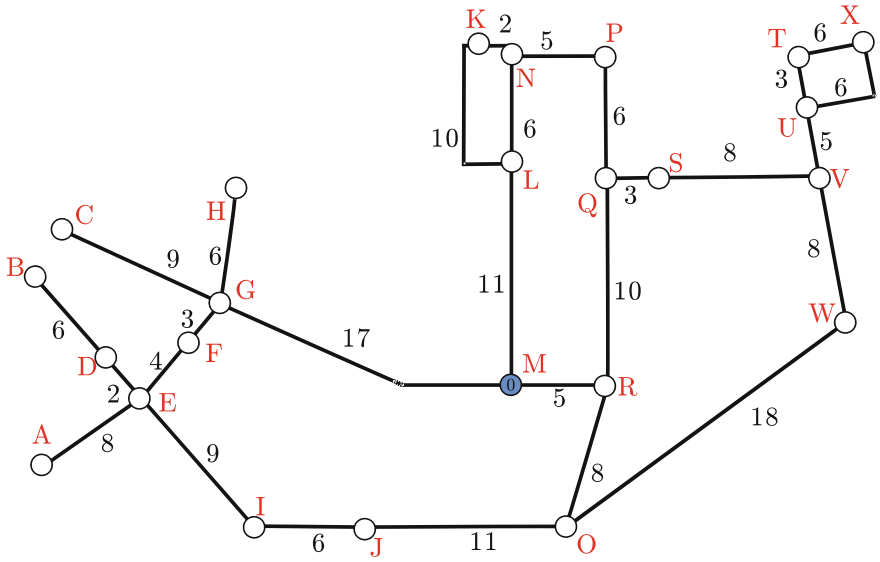
```
1  Alle Knoten warten, alle d[v] sind unendlich, nur d[Startknoten] = 0
2  while es wartende Knoten gibt do
3      v := der wartende Knoten mit kleinstem d[v]
4      Mache v hängend
5      for all Fäden von v zu einem Nachbarn u der Länge ℓ do
6          if d[v] + ℓ < d[u], then d[u] := d[v] + ℓ
               // kürzeren Weg hin zu u gefunden, führt über v
```

Was hat dieser Algorithmus mit dem Prozess des langsamen Hochhebens des Startknotens zu tun? Jede Iteration der while-Schleife entspricht dem Übergang eines Knotens $v$ von wartend nach hängend. Der jeweils nächste angehobene Knoten ist der wartende Knoten $v$ mit dem kleinsten Wert $d[v]$. Dieser Wert ist dann die Höhe, in die wir den Startknoten heben müssen, damit $v$ hängend wird. Da andere, später angehobene Fäden diese Höhe nicht mehr verkleinern können, entspricht $d[v]$ auch der entgültigen Entfernung vom Startknoten zu $v$.

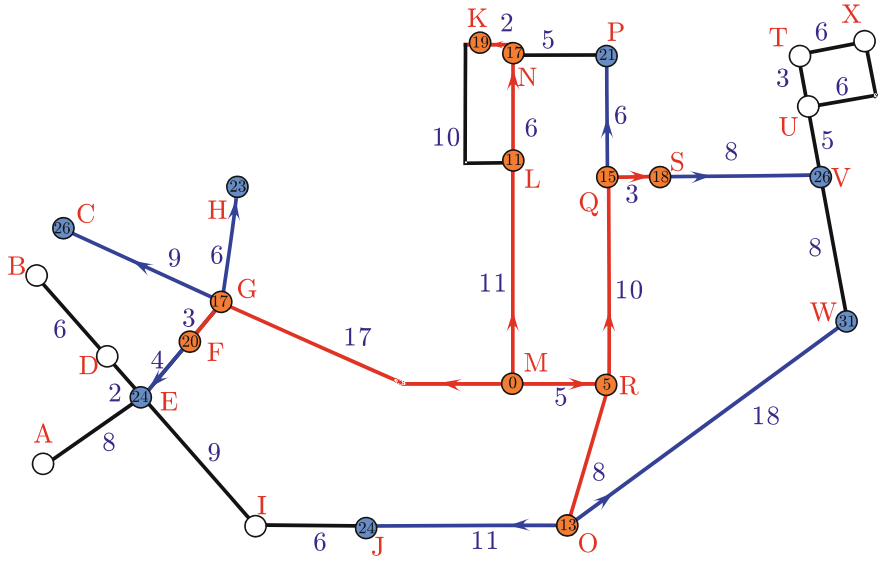
Ein großer Pluspunkt für Dijkstras Algorithmus ist, dass die $d[u]$-Werte der anderen Knoten sich sehr einfach anpassen lassen, wenn $v$ hängend wird: Lediglich die von $v$ ausgehenden Fäden müssen betrachtet werden. Dies erledigt die innere for-Schleife. Ein Faden zwischen $v$ und einem Nachbarknoten $u$ mit Länge $\ell$ ergibt eine Verbindung vom Startknoten über $v$ zu $u$ der Länge $d[v] + \ell$. Wenn dieser Wert kleiner ist als der bisherige Wert von $d[u]$, wird $d[u]$ entsprechend verringert. Am Ende hängen alle vom Startknoten aus erreichbaren Knoten, und die $d[v]$-Werte geben die kürzesten Weglängen an.

Im Folgenden sieht man einige Schritte im Ablaufe des Algorithmus. Hängende Knoten sind orange, blaue sind wartend und die verbleibenden, bisher unerreichten Knoten sind weiß. In den Knoten steht das aktuelle $d[v]$. Nach Beendigung des Algorithmus kann man vom Zielknoten aus die roten Fäden entlang rückwärts gehen, um den kürzesten Weg zu finden.

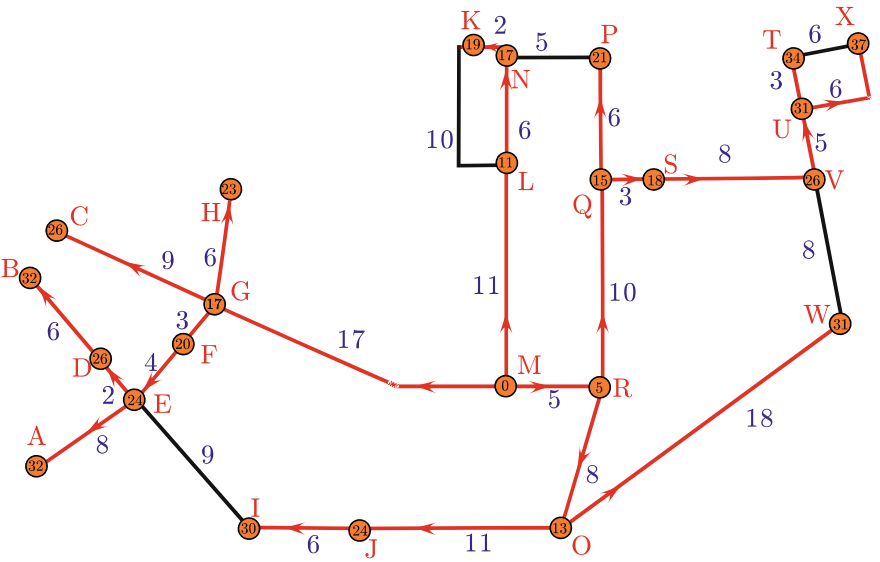
Der Algorithmus startet mit folgender Konfiguration. Alle Knoten liegen noch auf dem Tisch.




Die nächste Abbildung zeigt den Zustand des Algorithmus nach zehn Schritten, also die gleiche Situation wie beim hängenden Fadengewirr in der ersten Abbildung in diesem Kapitel.



Der Endzustand sieht so aus: Alle Knoten hängen in der Luft, die kürzesten Wege von M aus führen über die roten Fäden.




Jetzt kann ich mittels meiner Computerimplementierung nach Herzenslust Abstände zwischen Knoten ausrechnen. Dabei muss ich weder Asche wegfegen, noch Fäden entwirren. Langsam verraucht mein Zorn. Vielleicht bekommt mein Bruder doch kein permanentes Hausverbot. Für echte Routenplanung muss ich Dijkstras Algorithmus aber noch so erweitern, dass auch die kürzesten Wege selbst ablesbar sind: Immer wenn $d[u]$ neu gesetzt wird, merkt sich das Programm, welcher Knoten dafür verantwortlich war, nämlich der Knoten $v$, der gerade neu angehoben wurde. Am Ende wird die Route durch Zurückverfolgen dieser Vorgängerverweise rückwärts vom Ziel zum Start aufgerollt. Die Vorgängerverweise enthalten die gleiche Information wie die roten Fäden in den Beispielabbildungen.

## FAQ / Zum Weiterlesen

**Wo wird Dijkstras Algorithmus ausführlich beschrieben?** Es gibt viele gute Algorithmenlehrbücher, die alles über Dijkstras Algorithmus erklären. Zum Beispiel:
T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen.* Spektrum Akademischer Verlag.

**Wer war Dijkstra?** Edsger W. Dijkstra[1] lebte von 1930 bis 2002. Er erfand nicht nur den hier vorgestellten Algorithmus, sondern lieferte auch wichtige

[1] `http://de.wikipedia.org/wiki/Edsger_Wybe_Dijkstra`

Beiträge zur systematischen Programmierung und zur Modellierung gleichzeitiger Abläufe. Im Jahr 1972 erhielt er den Turing Award, die höchste Auszeichnung für einen Informatiker. Sein berühmter Artikel *"A note on two problems in connexion with graphs"* wurde 1959 in der Zeitschrift *Numerische Mathematik* veröffentlicht. Das ‚andere' Problem, von dem Herr Dijkstra spricht, ist die Berechnung minimaler spannender Bäume. Lässt man nämlich das „$d[v]+$" in Zeile 6 unseres Pseudocodes weg, erhält man den Jarník-Prim-Algorithmus aus Kap. 35 (Minimale aufspannende Bäume).

**Wie heißen Bindfäden etc. in Fachchinesisch?** *Knoten* heißen in der Informatik tatsächlich so, aber statt von Bindfäden spricht man von *Kanten*. Netzwerke aus Knoten und Kanten bilden *Graphen*.

**Hab ich so etwas Ähnliches nicht schon irgendwo hier gesehen?** In der Informatik ist die Suche nach Pfaden und das verwandte Problem des Suchens nach Kreisen sehr wichtig.

- *Tiefensuche* zählt systematisch bestimmte Pfade auf. Darauf bauen viele Algorithmen auf. Siehe z. B. die Kap. 7 (Tiefensuche) und 9 (Zyklensuche in Graphen).
- Die *Euler-Kreise* aus Kap. 29 benutzen jede Kante genau ein Mal.
- Beim in Kap. 42 (Das Travelling Salesman Problem) beschriebenen Handlungsreisenden-Problem wird eine möglichst kurze Rundreise zwischen Städten gesucht. Die Reisezeiten zwischen den Städten zu bestimmen, ist wiederum ein Problem kürzester Wege.

**Wie implementiere ich den Pseudocode effizient?** Wir benötigen eine Datenstruktur, die die folgenden Operationen unterstützt: Knoten einfügen, Knoten mit kleinstem Abstand entfernen und Abstand ändern. Da diese Kombination von Operationen häufig benötigt wird,hat man einen Namen dafür – *Prioritätswarteschlange*. Schnelle Prioritätswarteschlangen brauchen für keine der Operationen mehr Zeit als proportional zum *Logarithmus* der Anzahl an Knoten.

**Geht es noch schneller?** Müssen wir für den schnellsten Weg von Karlsruhe nach Barcelona tatsächlich das gesamte Straßennetz Westeuropas inspizieren? Bis hinunter zum letzten Feldweg? Der gesunde Menschenverstand sagt etwas anderes. Gegenwärtige kommerzielle Routenplaner betrachten „weit weg" von Start und Ziel nur Fernstraßen, können aber leider nicht garantieren, dass kein Schleichweg übersehen wurde. In den letzten Jahren wurden aber noch schnellere Verfahren entwickelt, die außerdem optimale Ergebnisse garantieren. Siehe z. B. Arbeiten unserer Arbeitsgruppe `http://algo2.iti.uni-karlsruhe.de/schultes/hwy/`.

**Nur ein Routenplaner für Straßennetze?** Das Problem ist viel allgemeiner, als es den Anschein hat. Zum Beispiel benötigt Dijkstras Algorithmus keine Kenntnis über die geografische Position eines Knotens. Man ist auch nicht auf (räumliche) Abstände beschränkt, sondern könnte z. B. Reisezeiten

als Fadenlängen verwenden. Das Ganze funktioniert sogar bei Einbahnstraßen oder unterschiedlichen Reisezeiten für Hin- und Rückrichtung, denn unser Algorithmus betrachtet die Länge eines Fadens immer nur in Richtung vom Start zum Ziel. Das Netzwerk kann auch viele andere Dinge modellieren. Zum Beispiel öffentliche Verkehrsmittel samt Abfahrtszeiten oder Kommunikationsleitungen im Internet. Sogar Probleme, die auf den ersten Blick nichts mit Pfaden zu tun haben, lassen sich oft passend umformulieren. Zum Beispiel läßt sich der Abstand zwischen zwei Zeichenketten (Gensequenzen) in Kap. 32 (Dynamische Programmierung) als Weglänge in einem Graphen auffassen. Knoten sind Paare von Buchstaben aus den zwei Eingaben, die „einander zugeordnet werden". Kanten entsprechen den Operationen *Streichen*, *Einfügen*, *Ersetzen* und *Übernehmen*.

**Können Straßen negative Länge haben?** So etwas kann tatsächlich nützlich sein. Damit kann ich z. B. ausdrücken, dass sich dort meine Lieblingseisdiele befindet, für die ich gern einen kleinen Umweg in Kauf nehme. Allerdings darf es keine Rundreise geben, die insgesamt eine negative Länge hat. Ansonsten könnte ich beliebig lange (Eis essend) im Kreis herumfahren, der Weg würde immer kürzer werden, es gäbe also keinen kürzesten mehr. Doch auch bei Abwesenheit negativer Kreise ist der Algorithmus von Dijkstra überfordert. Das Problem ist, dass ein Bindfaden, dem wir negative Länge zuschreiben, über einen bereits hängenden Knoten viele verbesserte Verbindungen zu anderen Knoten herstellen könnte. Dijkstras Algorithmus sieht diesen Fall aber nicht vor. Eine in dieser Beziehung bessere Alternative ist der Algorithmus von Bellman und Ford, der viel vorsichtiger vorgeht, dafür aber viel langsamer ist.

# 35

# Minimale aufspannende Bäume (Wenn das Naheliegende das Beste ist...)

Katharina Skutella und Martin Skutella

Technische Universität Berlin

Einst lebte in einem fernen Inselreich der Stamm der Algolaner. Die Stammesmitglieder hausten verstreut auf allen sieben Inseln des unten abgebildeten Inselreichs.

Zwischen den sieben Inseln und dem Festland verkehrten mehrere Fähren, die gegenseitige Besuche und Ausflüge auf das Festland ermöglichten. Die Fährverbindungen sind in der Karte (Abb. 35.1) gestrichelt eingezeichnet. Die Zahlen geben die Länge der Fährverbindungen in Metern an.

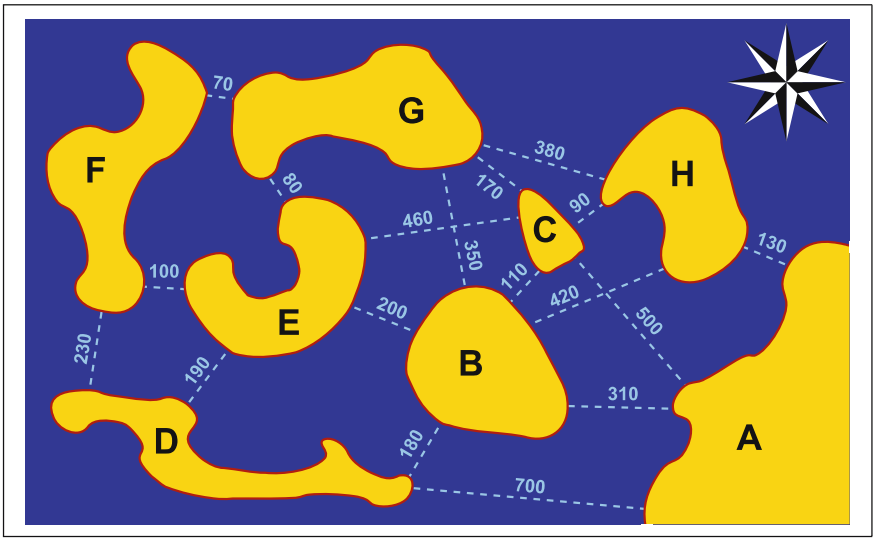



**Abb. 35.1.** Das Inselreich der Algolaner umfasste die sieben Inseln B,C,...,H. Die gestrichelten Linien stellen Fährverbindungen dar. Die Zahlen geben die Länge der Fährverbindungen in Metern an. Zum Beispiel verkehrte eine Fähre zwischen dem Festland A und Insel D, die bei jeder Überfahrt eine Strecke von 700 m zurücklegte

## Der Brückenbau der Algolaner

Bei stürmischem Wetter kam es regelmäßig vor, dass eine Fähre kenterte. Deshalb beschlossen die Algolaner, einige Fährverbindungen durch Brücken zu ersetzen.

Im ersten Jahr sollte durch den Bau einer Brücke eine der sieben Inseln an das Festland angeschlossen werden. Die Algolaner wählten die Brücke der Länge 130 m zwischen A und H, da die Verbindungen von A zu anderen Inseln länger sind.

Im zweiten Jahr wollte man eine weitere Insel an das Festland anschließen. In Frage kam also der Bau einer Brücke von A oder von H aus zu einer der anderen Inseln. Die Algolaner entschieden sich für die kürzestmögliche Brücke der Länge 90 m zwischen C und H.

Im dritten Jahr wollte man mit dem Bau einer weiteren Brücke (von A, C oder H aus) eine dritte Insel mit dem Festland verbinden. Die kürzestmögliche Variante war dieses Mal die Brücke der Länge 110 m zwischen B und C. Den Zwischenstand des laufenden Bauprojekts kannst du der Abb. 35.2 entnehmen. Wie du siehst, waren die Algolaner noch längst nicht fertig mit ihren Bauvorhaben.

In den folgenden Jahren wurden die Brücke der Länge 170 m zwischen C und G, dann die Brücke der Länge 70 m zwischen F und G, als nächstes die Brücke der Länge 80 m zwischen E und G und schließlich die Brücke der Länge 180 m zwischen B und D gebaut.

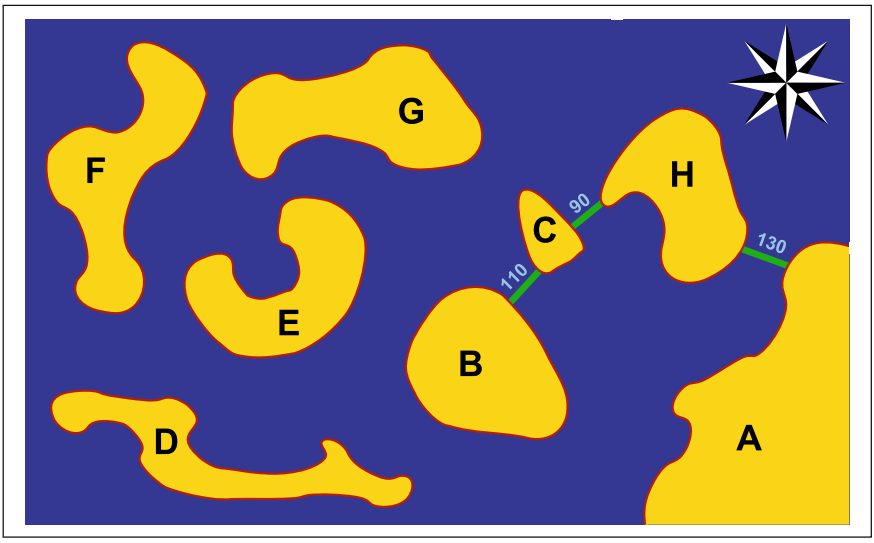



**Abb. 35.2.** Der Zwischenstand des Bauprojekts nach drei Jahren. Die Inseln B, C und H sind bereits an das Festland angebunden

Nach Ablauf des siebten Jahres waren also alle Inseln untereinander und mit dem Festland durch Brücken verbunden. Das Brückenprojekt war damit abgeschlossen. Das fertige Brückensystem der Algolaner ist in Abb. 35.3 dargestellt.

Die Algolaner waren sehr zufrieden, da der Aufwand für den Bau der Brücken zwar groß gewesen war, man aber andererseits überzeugt war, keine unnötig langen Brücken gebaut zu haben. Die Länge aller Brücken zusammen betrug, wie du leicht nachrechnen kannst, genau 830 m.

## Der Brückenbau nach dem Orkan

Kurz nach Fertigstellung der letzten Brücke fegte ein wütender Orkan über das Inselreich und zerstörte die mühsam errichteten Brücken. Nachdem sie sich von dem Schock erholt hatten, beschlossen die Algolaner, ein neues Brückensystem zu errichten. Die neuen Brücken sollten wieder alle Inseln untereinander und mit dem Festland verbinden.

In Folge des Orkans war das Baumaterial knapp geworden. Man einigte sich darauf, zunächst eine möglichst kurze Brücke zu bauen. Daher wurde im ersten Jahr die Brücke der Länge 70 m zwischen F und G erbaut.

Auch im zweiten Jahr war nur wenig Material vorhanden, so dass man die nächst längere Brücke der Länge 80 m zwischen E und G erbaut.

Im dritten Jahr wurde gemäß dieser Strategie die Brücke der Länge 90 m zwischen C und H gebaut. Nach dem Bau dieser drei Brücken waren die drei

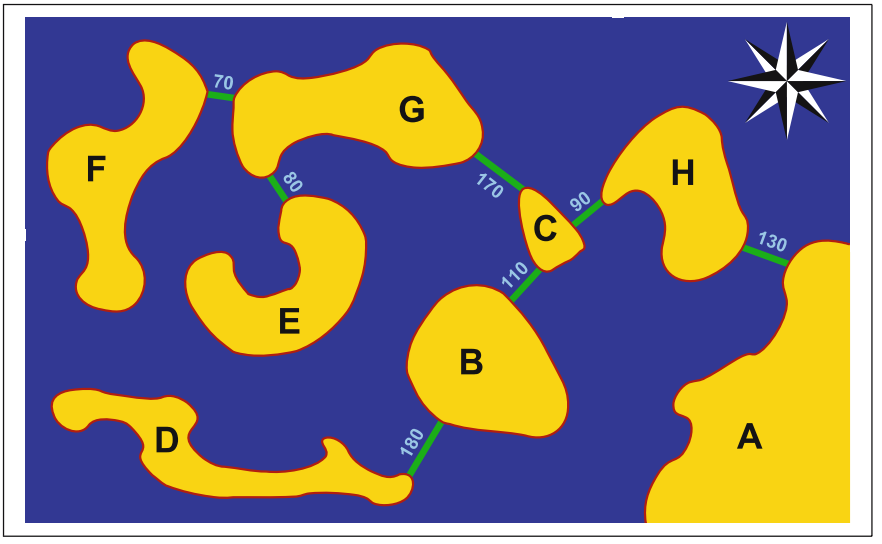



**Abb. 35.3.** Das fertige Brückensystem der Algolaner

Inseln E, F und G und die beiden Inseln C und H miteinander verbunden (siehe Abb. 35.4).

Die kürzeste noch nicht bestehende Verbindung war im vierten Jahr die der Länge 100 m zwischen E und F. Da diese beiden Inseln jedoch bereits über G miteinander verbunden waren, errichtete man statt dessen die Brücke der Länge 110 m zwischen B und C.

Im fünften Jahr kam die Brücke der Länge 130 m zwischen A und H hinzu, danach die Brücke der Länge 170 m zwischen C und G und schließlich im siebten Jahr die Brücke der Länge 180 m zwischen B und D. Das neue Brückensystem der Algolaner kannst du Abb. 35.5 entnehmen.

Mit Verblüffen stellten die Algolaner fest, dass trotz der veränderten Strategie bei der Auswahl der Brücken wieder das gleiche Brückensystem der Länge 830 m entstanden war (vergleiche Abb. 35.3). Dies bestärkte die Algolaner in ihrer Auffassung, das optimale Brückensystem für ihre Inseln gefunden zu haben. Und wenn kein zweiter Orkan sein Unwesen über dem Inselreich getrieben hat, dann spazieren die Algolaner noch heute glücklich und stolz über ihre Brücken...

## Die Algorithmen von Prim und Kruskal

Jetzt fragst du dich bestimmt, ob die Algolaner zurecht so stolz auf ihr Brückensystem waren. Vielleicht gibt es ja doch ein besseres, also kürzeres Brückensystem? Du kannst dich durch Ausprobieren davon überzeugen, dass jedes andere Brückensystem, das alle sieben Inseln und das Festland miteinander verbindet, länger als 830 m ist.

Ein Brückensystem minimaler Gesamtlänge, das einige Orte (hier das Festland A und die Inseln B bis H) miteinander verbindet, heißt „minimaler aufspannender Baum“. Das Problem, einen minimalen aufspannenden Baum zu finden, hat neben dem Brückenbau zahlreiche andere praktische Anwendungen. Es tritt beispielsweise auf, wenn Grundstücke eines Neubaugebietes möglichst kostengünstig an die Kanalisation angeschlossen werden sollen. Andere Anwendungen tauchen beim Entwurf von Computer-Chips und bei der Planung von Verkehrs- oder Kommunikationsnetzen (Telefon, Fernsehen, Internet etc.) auf.

Hinter den beiden Strategien der Algolaner stecken bekannte Algorithmen zur Lösung des Problems. Die erste Strategie ist unter dem Namen „Algorithmus von Prim“ bekannt. Dieser Algorithmus bindet die Orte nacheinander an das Festland an. Dabei wird in jedem Schritt die kürzestmögliche Brücke gebaut.

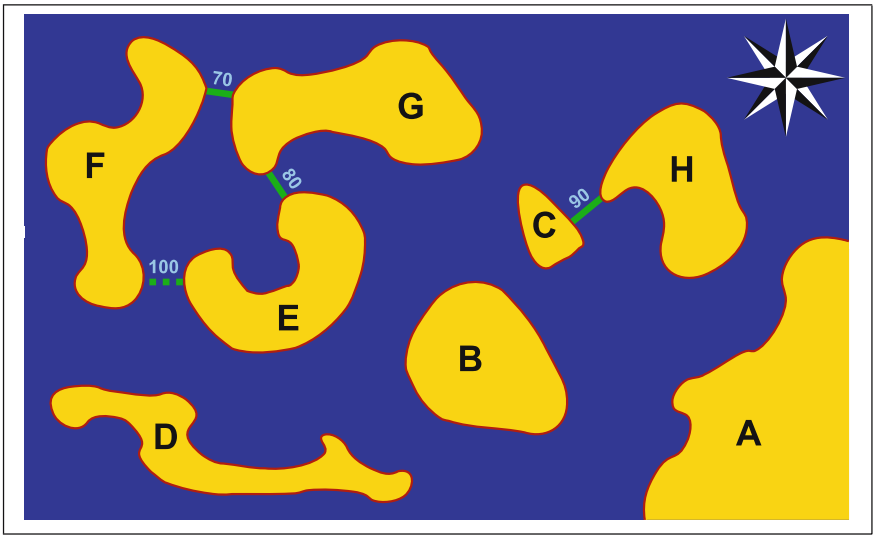



**Abb. 35.4.** Der Stand des zweiten Brückenbauprojekts nach drei Jahren. Das Brückensystem besteht aus drei Brücken der Längen 70 m, 80 m und 90 m. Die kürzeste, noch nicht bestehende Verbindung ist diejenige zwischen den Inseln E und F (in der Karte gestrichelt eingezeichnet). Die Algolaner entschieden sich im vierten Jahr jedoch gegen den Bau dieser Brücke, da diese beiden Inseln bereits über G miteinander verbunden sind

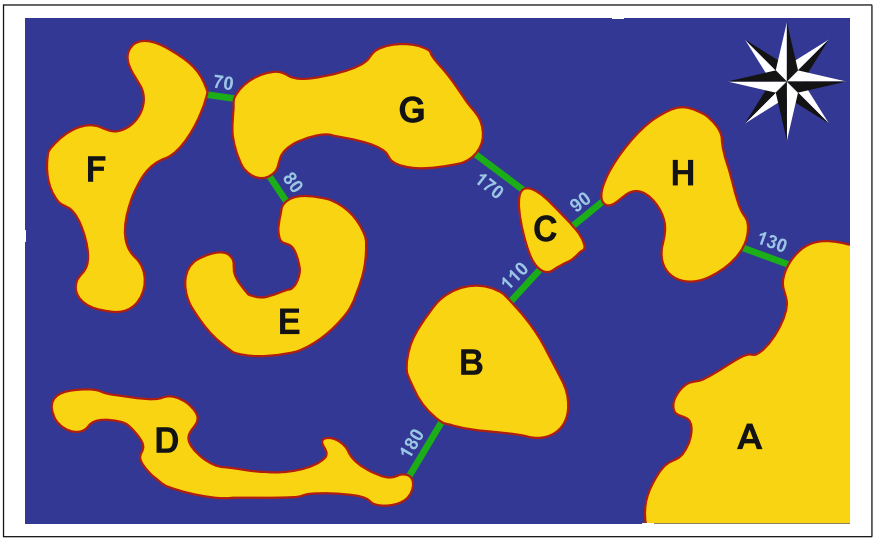



**Abb. 35.5.** Das zweite Brückensystem der Algolaner

| Algorithmus von Prim |
| --- |
| Wähle einen speziellen Ort aus (Festland) und nenne ihn erreichbar.<br>Alle anderen Orte sind zunächst nicht erreichbar.<br>Führe den folgenden Schritt so oft aus, bis alle Orte erreichbar sind:<br>Baue die kürzeste Brücke zwischen zwei Orten, von denen einer erreichbar und der andere nicht erreichbar ist, und nenne den bislang nicht erreichbaren Ort erreichbar. |

Die zweite oben beschriebene Strategie (nach dem Orkan) ist unter dem Namen „Algorithmus von Kruskal“ bekannt. Dieser Algorithmus baut in jedem Schritt die kürzeste Brücke, die zwei noch nicht miteinander verbundene Orte verbindet. Er unterscheidet sich von dem Algorithmus von Prim darin, dass er nicht nur Brücken in Betracht zieht, die eine Verbindung zum Festland herstellen, sondern den Bau beliebiger Brücken zwischen bislang unverbundenen Orten erlaubt.

| Algorithmus von Kruskal |
| --- |
| Führe den folgenden Schritt so oft aus, bis alle Orte untereinander durch Brücken verbunden sind:<br>Baue die kürzeste Brücke, die zwei Orte verbindet, die bislang nicht voneinander aus erreichbar sind. |

Die Algorithmen von Prim und Kruskal berechnen einen minimalen aufspannenden Baum und sie haben eine weitere Gemeinsamkeit. Führ dir noch einmal das Vorgehen der Algolaner vor Augen. Bei beiden Verfahren haben die Algolaner relativ kurzsichtig von Jahr zu Jahr geplant. Sie haben jedes Jahr die beste (d.h. kürzeste) Brücke gebaut, die zu dem Zeitpunkt in Frage kam. Dabei haben sie keine Rücksicht darauf genommen, welche Auswirkungen die getroffene Entscheidung für den weiteren Verlauf des Brückenbauprojekts hat. Wie du siehst, kann auch das „Naheliegende“ einmal zum Erfolg führen.

Algorithmen mit dieser Eigenschaft werden auch „greedy“ (englisch für „gierig“) genannt, weil sie in jedem Schritt die beste Wahl treffen, die momentan zur Verfügung steht. Solch ein „kurzsichtiger“ Ansatz führt bei anderen Problemen nicht immer zum Ziel. Stelle dir beispielsweise vor, Du sollst nur die Insel D durch ein Brückensystem minimaler Länge an das Festland A anbinden. In dem kurzsichtig entworfenen Brückensystem der Algolaner erreicht man D von A aus über die Inseln H, C und B. Für diesen Weg ergibt sich eine Gesamtlänge von 510 m. Bestimmt kannst du eine kürzere Verbindung zwischen A und D finden! Wenn du mehr darüber erfahren möchtest, wie man die kürzeste Verbindung vom Festland A zu der Insel D findet, dann schau doch mal in Kap. 34 nach.

Die beiden Algorithmen haben übrigens eine weitere interessante Eigenschaft. Sie berechnen immer eine Lösung, in der die Länge der längsten Brücke so klein wie möglich ist. Am Beispiel des Inselreichs kannst du das überprüfen.

## Zum Weiterlesen

1. Kapitel 34 (Kürzeste Wege)

   Nicht alle Algolaner waren zufrieden mit ihren Brücken. Zum Beispiel beschwerte sich Häuptling Hinkefuß von der Insel D, der regelmäßig den Medizinmann auf dem Festland konsultierte, dass der Weg von D über B, C und H nach A unnötig lang sei (510 Brückenmeter). Man hätte doch besser eine Brücke von A nach B bauen sollen, welche die Verbindung auf 490 Brückenmeter verkürzt hätte. Wie man die kürzeste Verbindung vom Festland zu allen Inseln findet, erfährst du in Kap. 34.

2. Kapitel 42 (Das Travelling Salesman Problem)

   Auch der Milchmann Müde-Molke, der täglich jede Insel mit einem Sack Kokosnüsse belieferte, machte seinem Unmut Luft. Seiner Meinung nach hätte man die Brücken so bauen sollen, dass sich ein möglichst kurzer Rundweg vom Festland aus über alle Inseln ergeben hätte. Wie dem Milchmann geholfen werden kann, erfährst du in Kap. 42.

3. Kapitel 3 (Schnelle Sortieralgorithmen)

   Möchte man den Algorithmus von Kruskal anwenden, ist es ratsam, zunächst die zur Diskussion stehenden Verbindungen der Länge nach zu sortieren, um sie dann in dieser Reihenfolge abarbeiten zu können. Wie man möglichst schnell sortiert, erfährst du in Kap. 3.

4. Kapitel 9 (Zyklensuche in Graphen)

   Im vierten Jahr nach dem Orkan bauten die Algolaner nicht etwa die kürzere Brücke von E nach F, sondern die Brücke von B nach C. Denn die Brücke von E nach F hätte zwei Inseln miteinander verbunden, die bereits über mehrere andere Brücken miteinander verbunden waren. Man kann das auch so ausdrücken: Der Bau der Brücke von E nach F hätte einen Zyklus (oder Kreis) von E über F nach G und zurück nach E ergeben. Mehr über Zyklen und wie man sie findet erfährst du in Kap. 9.

5. Lutz-Westphal, B.: *Günstig verbunden: Minimale aufspannende Bäume*. In: Hussmann, S., Lutz-Westphal, B. (Hrsg.): *Kombinatorische Optimierung erleben*. Vieweg Verlag, 1. Auflage, 2007, Kap. 2, S. 39–66

   Leitungsnetze planen, Straßen erneuern und Computer verkabeln: Dieses Buch behandelt weitere Anwendungsbeispiele für minimale aufspannende Bäume. Die Beispiele werden sorgfältig und ausführlich modelliert. Die Lösungsalgorithmen werden anschaulich und verständlich erklärt. In diesem Buch findest du u.a. den Beweis dafür, dass die Algorithmen von Prim und Kruskal immer einen optimalen Baum bestimmen. Das Buch richtet sich gezielt auch an Schülerinnen und Schüler.

6. Gritzmann, P., Brandenberg, B.: *Das Geheimnis des kürzesten Weges: Ein mathematisches Abenteuer*, Springer, 3. Auflage, 2005, S. 138–161

   Auf ihren mathematischen Abenteuern lüftet die 15-jährige Ruth auch das „Geheimnis“ des minimalen aufspannenden Baumes. Warum der Algorithmus von Prim tatsächlich immer einen optimalen Baum findet, erfährst du in diesem unterhaltsamen und spannenden Buch.

7. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: *Introduction to Algorithms (Second Edition)*, MIT Press, 2001.

Wie man die Algorithmen von Prim und Kruskal so auf dem Computer programmiert, dass sie möglichst schnell die gesuchte Lösung finden, und vieles mehr, erfährst du in diesem Buch, dass auch an vielen Universitäten in Anfängervorlesungen für angehende Informatikerinnen und Informatiker verwendet wird.

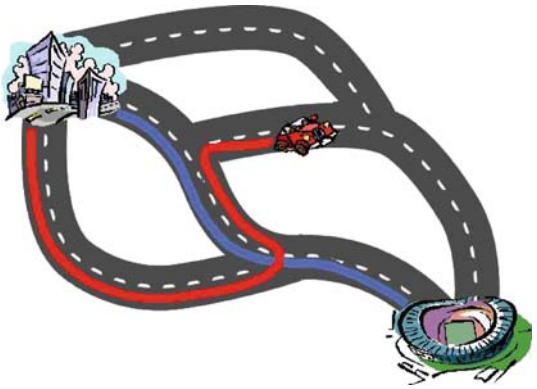
# 36

# Maximale Flüsse – Die ganze Stadt will zum Stadion

Robert Görke, Steffen Mecke und Dorothea Wagner

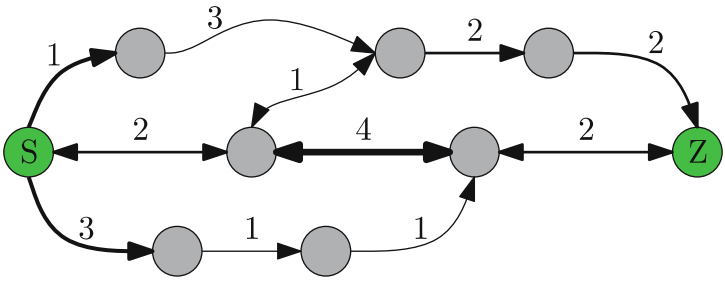
Universität Karlsruhe

„Was ist denn das für ein Mist, so kommen wir ja nie zum Fußballstadion!“ Jogi saß im Auto neben seiner Mutter und wurde langsam unruhig. „Ich kann doch auch nichts dafür, dass alle diese Straße zum Stadion benutzen. Jetzt ist eben Stau“, meinte sie nur. – „Dann dreh' doch einfach um und wir nehmen da hinten die Fordstraße, die Route benutzt keiner.“ Jogis Mutter glaubte zwar nicht so recht daran, fuhr aber um des lieben Friedens willen zurück und tatsächlich, auf der Fordstraße war weniger Verkehr. Jedenfalls bis zur nächsten Kreuzung. Dort mündete die Fordstraße auf die vielbefahrene, breite Bahnhofstraße und es gab wieder Stau. „Diese Idioten haben doch keine Ahnung! Mama, bieg links ab“ – „Aber zum Stadion geht es doch geradeaus“, meinte die. „Schon“, erwiderte Jogi, „aber da hinten können wir dann die Karlstraße nehmen, das ist zwar ein Umweg, aber dafür ist die Straße bestimmt frei.“ Jogis Mutter war immer noch skeptisch, aber einen Versuch war es wert. Und tatsächlich, den Umweg schien keiner nehmen zu wollen. Jogi versuchte sogar noch, einige der auf der Bahnhofstraße entgegenkommenden Fahrzeuge zum Umdrehen zu bewegen, aber ohne Erfolg. „Sind die blöd, jetzt stehen sie gleich im Stau, dabei könnten sie hier leicht durchkommen!“



Das Fußballspiel stellte sich als übles Herumgeschiebe heraus, so dass es Jogi schließlich so langweilig wurde, dass er nochmal über das heutige Verkehrsproblem nachdachte. „Wenn man die Autofahrer sich selbst überlässt, funktioniert es offensichtlich nicht sehr gut mit der Stauvermeidung. Man müsste an jeder Kreuzung Wegweiser aufstellen, nach denen sich die Autos dann richten müssten, damit möglichst viele Autos zum Ziel kommen und der Verkehr fließen kann. Aber wie finde ich die beste Lösung dieses Problems?“ Er kam zunächst zu keiner zufriedenstellenden Antwort. Erst ein paar Tage später sprach er noch mal mit seiner großen Schwester darüber. Die studierte schon Informatik, wusste aber auf Anhieb auch keine Lösung.

„Machen wir das Problem erst mal möglichst einfach: Nehmen wir an, alle Autofahrer starten vom gleichen Punkt“ – sie markierte diesen Punkt und malte ein großes „$S$“ für Start darauf – „und wollen zu einem anderen Punkt.“ Diesen markierte sie mit „$Z$“, für Ziel. „Dazwischen sind die Straßen, die an den Kreuzungen jeweils zusammentreffen.“ Sie malte einige Punkte und Linien zwischen Start- und Zielpunkt.

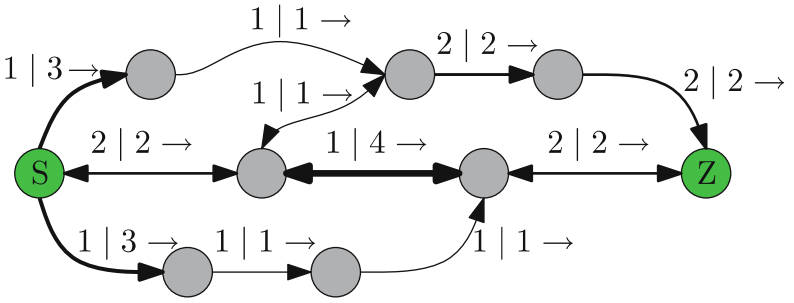



„Die Straßen können aber unterschiedlich breit sein, schreiben wir mal neben jede Straße die Anzahl der Spuren und machen sie in der Zeichnung entsprechend dicker oder dünner. Hmm, wir hatten es neulich in der Vorlesung von kürzesten Wegen, aber das hilft uns hier nicht richtig weiter. Natürlich kann man erst mal den kürzesten Weg von $S$ nach $Z$ suchen“. – „Du meinst diesen hier.“ Jogi zeichnete ihn ein. „Aber jetzt können wir doch weitere Wege suchen. Wir müssen uns nur merken, wie viele Autos jede Straße schon benutzen.“

Wir verlassen Jogi und seine Schwester an dieser Stelle mal und überlegen weiter: Wir haben also ein Straßennetz mit verschiedenen Straßenbreiten und wollen wissen, wie wir die Autos leiten müssen, damit der Verkehr am besten fließen kann. Damit es möglichst einfach bleibt, nehmen wir an, dass alle Autofahrer von $S$ nach $Z$ wollen. Normalerweise versucht man als Autofahrer, den kürzesten Weg zum Ziel zu nehmen. Wenn aber zu viele Autos gleichzeitig auf derselben Straße fahren, gibt es einen Stau. „Gleichzeitig“ heißt in unserem Fall „in der Stunde bevor das Fußballspiel losgeht“. Jede Straße kann nur eine bestimmte Zahl an durchfahrenden Autos verkraften (die *Kapazität* der Straße). Diese hängt weniger von ihrer Länge ab, mehr von ihrer Breite (Anzahl der Spuren). Eine Straße mit einer Spur können pro Stunde z. B. 1000 Autos

benutzen. Wir schreiben aber trotzdem eine 1, also die Zahl der Spuren, und keine 1000, da wir nicht mit so großen Zahlen hantieren möchten.

Kritische Punkte im Straßennetz sind auch die Kreuzungen. Wenn hier mehr Autos ankommen, als weiterfahren können, gibt es ebenfalls einen Stau. Wenn mindestens genauso viele Autos weiterfahren können, wie ankommen, ist aber alles in Ordnung. Wir fragen uns nun, wie viele Autos wir maximal durch das Straßennetz von $S$ nach $Z$ („gleichzeitig“, also pro Stunde) schleusen können. Eine Lösung des Problems für unser Beispiel wäre der folgende „Verkehrsfluss“:

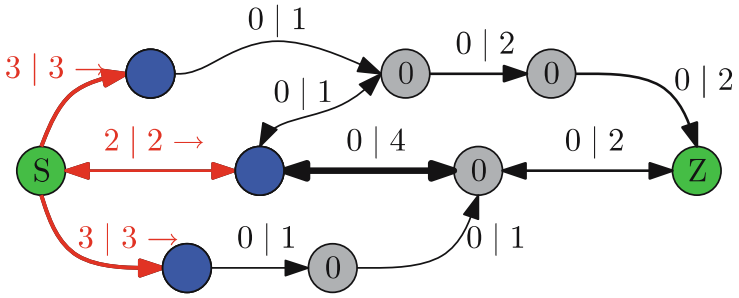



An jeder Straße steht nun zusätzlich die Anzahl der Autos, die diese Straße nehmen und ein Pfeil für die Richtung, in der sie fahren. Also heißt „$1 \mid 3 \rightarrow$“, dass eine Spur von drei vorhandenen benutzt wird und zwar nach rechts. Die erste Zahl darf niemals größer sein als die Zahl der Spuren der Straße (d. h., so etwas wie „$3 \mid 2$“ ist nicht erlaubt). Diese Regel ist so wichtig, dass wir ihr gleich einen Namen geben. Wir nennen sie die *Spurregel* (oder *Kapazitätsregel*). Die Bedingung, dass in jede Kreuzung genau so viele Autos hineinfahren wie herauskommen (sonst gibt es Stau) nennen wir die *Kreuzungsregel* (oder *Flusserhaltungsregel*, da sie dafür sorgt, dass der Verkehr an den Kreuzungen weiterfließen kann). Nur bei Start und Ziel gilt diese Regel nicht. Unter all den Verkehrsflüssen, die diese beiden Regeln erfüllen, suchen wir nun einen, bei dem möglichst viele Autos gleichzeitig am Start losfahren dürfen oder – was dann das Gleiche ist – am Ziel ankommen. Die Informatiker nennen so etwas dann einen *maximalen Fluss*.

Diese Art von Problemen gibt es nicht nur im Verkehrsbereich. Man kann sich so z. B. auch überlegen, wie man möglichst schnell Gebäude evakuieren oder Daten durch ein Computernetz leiten kann. Fallen dir noch andere Beispiele ein?

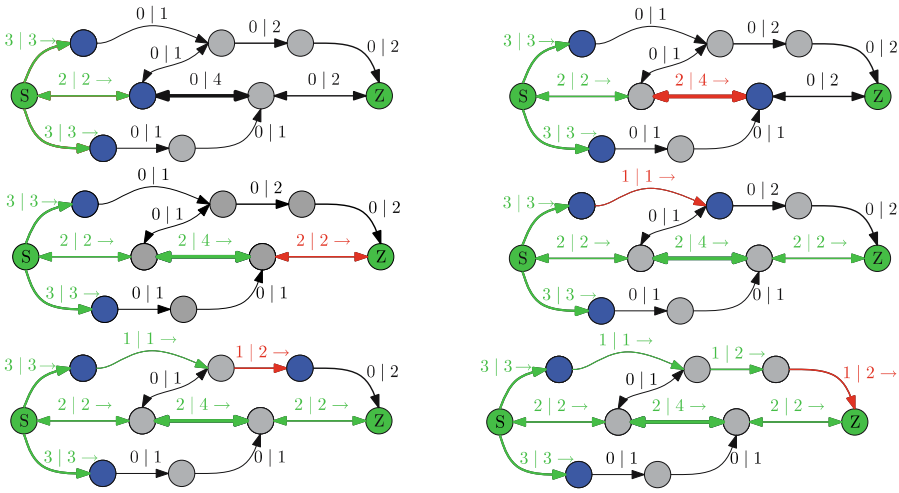
## Der Algorithmus

Wie finden wir nun einen maximalen Fluss? Versuchen wir es einfach mal: Am Anfang fahren noch gar keine Autos durch unser Straßennetz. Nun fangen wir damit an, erst einmal vom Startpunkt aus so viele Autos losfahren zu lassen (rot), wie überhaupt möglich ist, ohne die Spurregel zu verletzen.




Auf jeder Straße, die vom Startpunkt los geht, fahren (grün) nun also so viele Autos, wie freie Spuren vorhanden sind. Natürlich machen wir das alles erst einmal nicht mit echten Autos, sondern z. B. mit Modellautos. Oder einfach mit Bleistift und Papier. Erst wenn wir die beste Lösung für unser Problem gefunden haben, können wir damit anfangen, den Verkehr auf den echten Straßen mit echten Autos zu regeln.

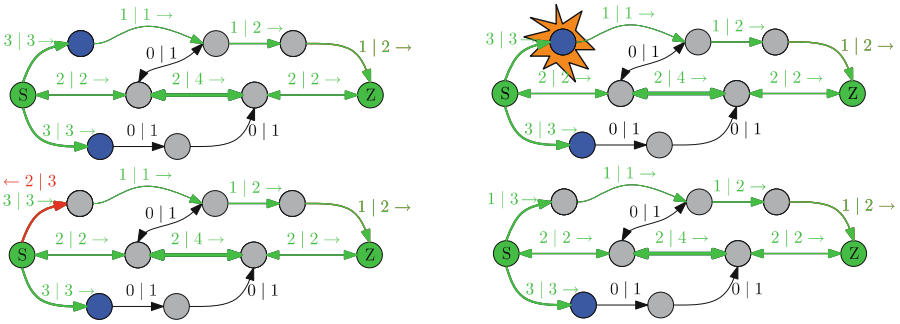
Jetzt haben natürlich die Kreuzungen an den Endpunkten dieser Straßen einen „Überschuss“ (blau) an Autos, die wir irgendwie weiterleiten müssen,



damit die Kreuzungsregel nicht verletzt wird. Um diesen Stau abzubauen, schieben wir die Autos einfach auf einer der nächsten Straßen weiter (wieder rot). Wir müssen aber wie immer die Spurregel beachten, dürfen also nicht mehr Autos weiterschieben, als Spuren vorhanden sind. Durch das Weiterschieben entsteht natürlich an der nächsten Kreuzung ein neuer Stau (blau), aber wenn die Autos bei $Z$ angekommen sind, müssen wir sie nicht mehr weiterschieben (sondern stellen sie dort einfach auf den Parkplatz).

Wir können also nicht immer so viele Autos weiterschieben, wie wir gerne möchten. Wichtig ist: Wir schieben immer so viele Autos wie möglich weiter, jedoch niemals mehr, als durch die Straße passen und natürlich auch nicht mehr, als die Kreuzung Überschuss hat. Und wir müssen natürlich auch auf

Einbahnstraßen achten. Falls wir irgendwo gar nicht mehr weiterkommen, weil mehr Autos an einer Kreuzung ankommen als über alle anderen Straßen weiterfahren können, müssen wir auch wieder Autos „zurückschieben“ können (wie an der Kreuzung oben links im nächsten Bild).




Dabei verringern wir also die Anzahl der Autos, die über eine Straße fahren. – Eigentlich darf man in Einbahnstraßen nicht rückwärts fahren, aber wir schieben hier ja keine echten Autos zurück. – Natürlich schieben wir höchstens so viele Autos zurück, wie nötig sind, damit der Überschuss an der Kreuzung verschwindet (danach ist die Kreuzung grau). Und selbstverständlich können wir nicht mehr Autos zurückschieben als ankommen.

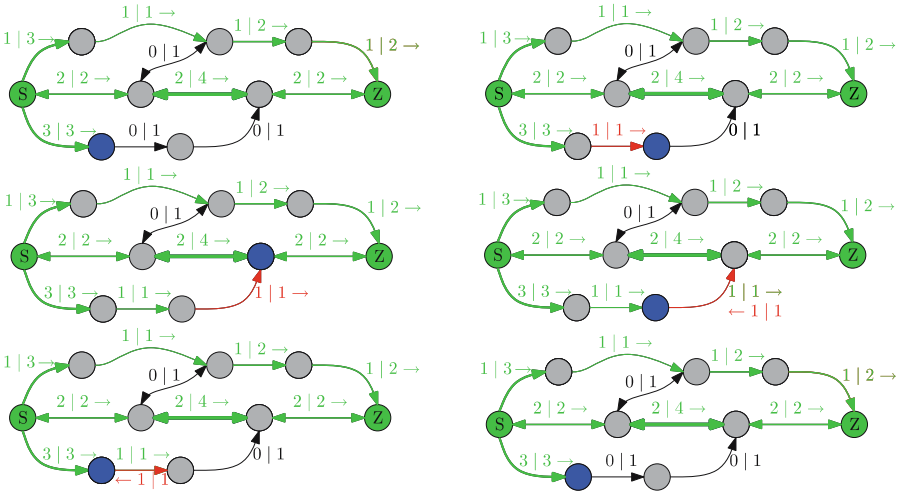
Insgesamt suchen wir uns in jedem Schritt eine Kreuzung mit Überschuss aus (also eine Kreuzung, an der im Moment mehr Autos ankommen, als weiterfahren) und schieben einen möglichst großen Teil davon weiter. Damit ergibt sich Folgendes:

Die Prozedur WEITERSCHIEBEN schiebt von einer Kreuzung aus, die Überschuss hat, entweder Autos weiter oder zurück.

```
1  procedure WEITERSCHIEBEN (K)
2  Voraussetzung K ist eine Kreuzung mit Überschuss
3  begin
4      Wähle eine der beiden folgenden Möglichkeiten:
5          Suche unter den Straßen, die von K wegführen, eine Straße aus,
           auf die noch Autos passen und schiebe so viele überschüssige
           Autos wie möglich darauf weiter.
               oder
6          Suche unter den Straßen, die zu K hinführen, eine Straße aus,
           auf der Autos zu dieser Kreuzung fahren, und schiebe so viele
           überschüssige Autos wie möglich über sie zurück.
7      Ende der Wahl
8  end
```

Leider führt dieses Vorgehen noch nicht zum Erfolg. Es kann dabei nämlich leicht passieren, dass zwei Kreuzungen (oder auch mehr als zwei) ihren Überschuss immer hin- und herschieben und wir nie zu einem Ende kommen, wie

bei den drei Kreuzungen in der folgenden Bildreihe. Die Informatiker sagen: „Der Algorithmus *terminiert* (endet) nicht.“




Wir brauchen also eine gute Idee, um die Suche nach dem besten Fluss etwas zielgerichteter zu machen. Führen wir doch zusätzlich die folgende Regel ein: Jede Kreuzung bekommt eine *Höhe*. Am Anfang haben alle Kreuzungen die Höhe 0. Später heben wir die Kreuzungen nach und nach an und zwar so: Wir vereinbaren, dass man Autos immer nur *von oben nach unten* schieben darf. Um also den Überschuss an einer Kreuzung weiterschieben zu dürfen, müssen wir sie erst einmal hochheben, sagen wir auf die Höhe 1. Dann dürfen wir zu allen Nachbarkreuzungen, die tiefer liegen, Überschuss weiterschieben (oder zurückschieben). Am Anfang heben wir also $S$ auf die Höhe 1 und schieben wie bisher so viele Autos wie möglich von $S$ weiter. Danach heben wir die nächste Kreuzung, die einen Überschuss hat, auf 1 und schieben weiter, dann die nächste und so weiter. Die Zielkreuzung $Z$ brauchen wir niemals anzuheben, denn wenn die Autos dort angekommen sind, müssen sie ja nicht mehr weiterfahren. Normalerweise können so schon ziemlich viele Autos bei $Z$ ankommen. Trotzdem kann es passieren, dass es nun noch Kreuzungen mit Überschuss gibt, die aber keine Nachbarkreuzungen mit Höhe 0 mehr haben, um ihren Überschuss loszuwerden. Dann dürfen wir sie weiter anheben. Wie weit heben wir sie an? Nun, mindestens um 1. Es kann sein, dass sich dadurch keine neue Möglichkeit zum Schieben ergibt. Dann können wir die Kreuzung noch um 1 weiter anheben. Aber nur so lange, bis sich gerade so wieder eine Möglichkeit zum Weiterschieben (oder zum Rückwärtsschieben!) ergibt. Auch das Anheben der Kreuzungen machen wir natürlich nur im Modell oder auf Papier. Wenn wir die Lösung unseres Problems gefunden haben, ist es nicht nötig, in der realen Welt die Bagger anrollen zu lassen, um irgendwelche realen Kreuzungen höherzulegen.

Die Prozedur ERHÖHEN hebt eine Kreuzung $K$ an, wenn sie einen Überschuss hat, diesen jedoch nicht weiterschieben kann.

```
1  procedure ERHÖHEN (K)
2  Voraussetzung Kein Weiterschieben des Überschusses von K möglich.
3  begin
4      Erhöhe K so lange, bis sich eine Möglichkeit zum Weiter- oder Zu-
       rückschieben zu einer niedrigeren Kreuzung ergibt.
5  end
```

In die Prozedur *Weiterschieben* fügen wir die Bedingung hinzu, dass Überschuss nur nach unten geschoben werden kann:

Diese Prozedur WEITERSCHIEBEN hat im Vergleich zur obigen Prozedur WEITERSCHIEBEN die Einschränkung, dass nur „bergab“ geschoben werden kann.

```
1  procedure WEITERSCHIEBEN (K)
2  begin
3      Führe einen der beiden folgenden Schritte durch:
4          Suche eine Straße, auf die noch Autos passen und die von K nach
           beispielsweise L führt. Falls K höher als L ist, schiebe überschüs-
           sige Autos auf dieser Straße weiter. Jedoch niemals mehr als auf
           die Straße passen und auch niemals mehr, als die Kreuzung noch
           Überschuss hat.
               oder
5          Suche eine Straße aus, auf der schon Autos von z.B. L zu K
           fahren. Falls K höher als L liegt, schiebe Autos zurück. Schiebe
           nie mehr Autos zurück, als die Kreuzung Überschuss hat.
6      Ende der Wahl      (Es ist nicht immer beides möglich!)
7  end
```

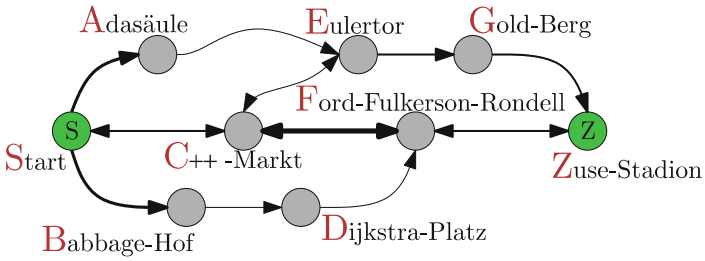
Wir können nun diese beiden Prozeduren (Erhöhen und Weiterschieben) so lange wiederholen, wie es eine Kreuzung mit Überschuss gibt, bzw. so lange, wie es von $S$ aus noch eine freie Straße gibt, auf der wir Autos losschicken können. Wir hören auf, $S$ zu erhöhen, wenn $S$ die Höhe $n$ hat, wobei $n$ die Anzahl der Kreuzungen in unserem Straßennetzwerk ist. Danach beseitigen wir nur noch den Überfluss an den restlichen Kreuzungen und sind fertig. Warum wir dann aufhören können, werden wir erst weiter unten, im Abschnitt „Warum funktioniert das?“ begründen. Ebensogut können wir $S$ auch gleich von Anfang an auf Höhe $n$ heben. In unserem Beispiel hat $S$ also die Höhe 9.

Der eigentliche Algorithmus sieht demnach so aus:

Der Algorithmus MAXIMALER FLUSS findet einen maximalen Fluss vom Startpunkt $S$ zum Ziel, indem wiederholt die Prozeduren ERHÖHEN und WEITERSCHIEBEN aufgerufen werden.

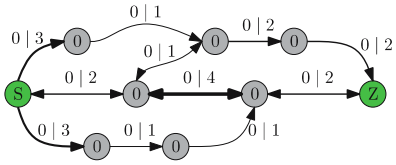
```
1   procedure MAXIMALER FLUSS (G, S, Z)
2   begin
3       Setze S auf Höhe n (n ist die Gesamtzahl der Kreuzungen)
4       Schiebe so viele Autos von S weg, wie jeweils auf die Straßen passen,
        die von S wegführen.
5       Setze alle anderen Kreuzungen (außer S) auf Höhe 0.
6       while Es exisitiert eine Kreuzung K mit Überschuss do
7           if Weiterschieben bei K möglich
8               Führe Prozedur WEITERSCHIEBEN (bei Kreuzung K) aus.
9           else
10              Führe Prozedur ERHÖHEN (bei Kreuzung K) aus.
11      endwhile
12  end
```

Um ein Beispiel für einen kompletten Durchlauf des Algorithmus beschreiben zu können, müssen wir nun erstmal allen Kreuzungen Namen geben:

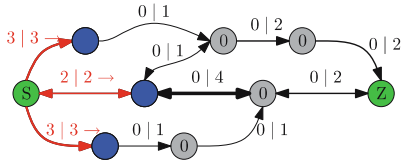



Die folgenden Bilder zeigen den Fortgang des Algorithmus. Wir haben in jede Kreuzung jeweils seine momentane Höhe hineingeschrieben.

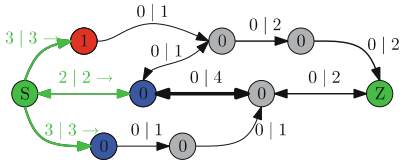
Anfangszustand:



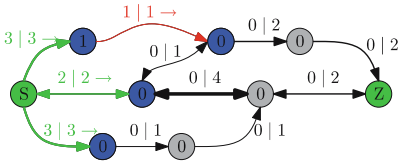
0. Schiebe alle Autos vom Start weg:



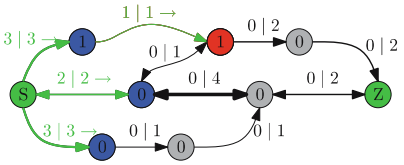
1. Erhöhen (Adasäule) auf 1:



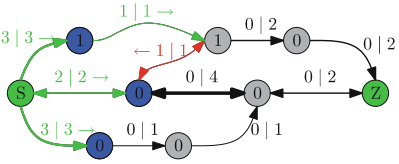
2. Weiterschieben (Adasäule):
Ein Auto zum Eulertor:



3. Erhöhen (Eulertor) auf 1

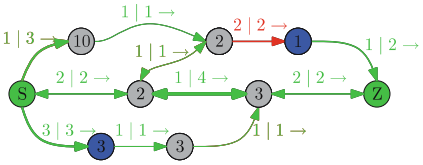



4. Weiterschieben ($E$): 1 nach $C$

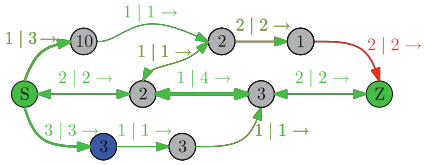



Die restlichen Schritte geben wir nur noch in Kurzform an:

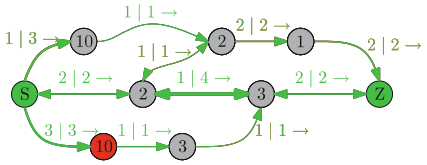
5. Erhöhen ($C$) auf 1
6. Weiterschieben ($C$): 3 nach $F$
7. Erhöhen ($F$) auf 1
8. Weiterschieben ($F$): 2 nach $Z$
9. Erhöhen ($F$) auf 2
10. Weiterschieben ($F$): 1 nach $C$
11. Erhöhen ($C$) auf 2
12. Weiterschieben ($C$): 1 nach $E$
13. Weiterschieben ($E$): 1 nach $G$
14. Erhöhen ($G$) auf 1
15. Weiterschieben ($G$): 1 nach $Z$
16. Erhöhen ($B$) auf 1
17. Weiterschieben ($B$): 1 nach $D$
18. Erhöhen ($D$) auf 2
19. Weiterschieben ($D$): 1 nach $B$
20. Erhöhen ($B$) auf 3
21. Weiterschieben ($B$): 1 nach $D$
22. Erhöhen ($D$) auf 3
23. Weiterschieben ($D$): 1 nach $F$
24. Erhöhen ($A$) auf 10
25. Weiterschieben ($A$): 2 nach $S$
26. Erhöhen ($F$) auf 3
27. Weiterschieben ($F$): 1 nach $C$
28. Weiterschieben ($C$): 1 nach $E$
29. Erhöhen ($E$) auf 2
30. Weiterschieben ($E$): 1 nach $G$



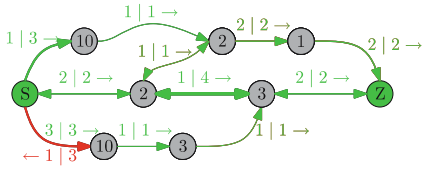

31. Weiterschieben ($G$): 1 nach $Z$



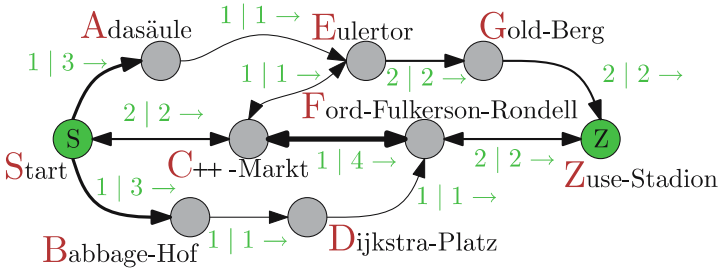

32. Erhöhen ($B$) auf 10




33. Weiterschieben ($B$): 2 nach $S$




Hier ist die Lösung unseres Problems:



Diese Lösung könnten wir nun verwenden, um in der realen Welt den Verkehr zu regeln, ohne dass es Stau gibt.

**Einige offene Fragen**

- Welche Kreuzung jeweils ausgewählt wird, ist im Algorithmus noch nicht genau festgelegt. Er funktioniert mit jeder beliebigen Reihenfolge, solange wir nur die Regeln für das Weiterschieben („immer nach unten“) und das Hochheben („nur, wenn kein Weiterschieben mehr möglich ist, und dann nur so weit, bis sich eine neue Möglichkeit ergibt“) beachten! Unser Beispiel braucht 33 Schritte: 15 mal Erhöhen und 18 mal Weiterschieben (das Erhöhen des Startpunktes ganz am Anfang und das Weiterschieben von dort aus haben wir nicht mitgezählt). Versuche doch mal, das obige Beispiel mit einer anderen Reihenfolge durchzuspielen. Schaffst du es, in weniger Schritten zum Ziel zu kommen?
- Hast du gemerkt, dass Überschuss erst dann Richtung $S$ zurückgeschoben werden kann, wenn die Höhe einer Kreuzung mehr als $n$ ist?

## Warum funktioniert das?

Der Algorithmus scheint auf magische Weise zu funktionieren. Wenn du Lust hast, kannst du einfach noch ein bisschen mit einem anderen Straßennetz herumexperimentieren. Falls dich aber interessiert, warum er funktioniert, dann lies einfach weiter:

Als erstes sollten wir uns klar machen, dass am Ende wirklich ein gültiger Verkehrsfluss herauskommt. Dazu müssen wir nur sehen, dass

- wir niemals mehr Autos über eine Straße geschickt haben, als sie Spuren hat (Spurregel) und
- am Ende keine Kreuzung mehr einen Überschuss hat (Kreuzungsregel).

Also kann der Verkehr ungehindert fließen.

Wenn man es an ein paar Beispielen ausprobiert hat, merkt man, dass es einen großen Unterschied gibt zwischen Kreuzungen, die höher als $n$ liegen und Kreuzungen, die niedriger sind. Von hohen Kreuzungen aus wird der Überschuss nämlich immer nur in Richtung $S$ zurückgeschoben. Dies sind die Kreuzungen, die keine Chance mehr haben, ihren Überschuss in Richtung $Z$ loszuwerden. Was aber passiert vorher?

Am Anfang wird immer nur Überschuss von Kreuzungen der Höhe 1 zu Kreuzungen der Höhe 0 geschoben. Dies sind sozusagen die einfachen Fälle. Erst wenn alle einfachen Möglichkeiten ausgeschöpft sind, werden die Kreuzungen nach und nach höher gehoben. Eine wichtige Beobachtung ist, dass Überschuss immer nur um 1 nach unten geschoben wird, also von einer Kreuzung $K$ der Höhe $h$ zu einer Nachbarkreuzung $L$ der Höhe $h-1$. Es kann nie

passieren, dass die Nachbarkreuzung $L$ beispielsweise Höhe $h-2$ hat. Schließlich hätten wir $K$ dann gar nicht so weit hochheben dürfen. Falls die Autos irgendwann bei $Z$ ankommen sollen, müssen sie also erst von $h$ auf $h-1$, dann von $h-1$ auf $h-2$ usw., langsam hinab bis zu $Z$, das immer Höhe 0 hat. Also geht es über mindestens $h$ verschiedene Stationen. Dadurch ist nun gewährleistet, dass Überschuss eben nicht ewig zwischen zwei Kreuzungen hin und her geschoben werden kann, denn insgesamt kann es ja höchstens $n-1$ verschiedene Stationen geben ($n-1$, nicht $n$, da wir nie bei $S$ vorbeikommen werden). Außerdem ist so gewährleistet, dass wirklich alle Möglichkeiten versucht werden, einen Überschuss noch weiterzuschieben, bevor wieder zu $S$ zurückgeschoben wird. Wenn es wirklich keine Möglichkeit mehr gibt, den Überschuss nach $Z$ zu leiten, funktioniert das Zurückschieben zu $S$ nach dem gleichen Prinzip. Am Schluss ist der ganze Überschuss entweder bei $Z$ oder bei $S$ angekommen und wir haben den besten Verkehrsfluss gefunden.

## Epilog

Jogis Schwester hat einige Zeit später gelernt, wie man wirklich beweisen kann, dass der Algorithmus den besten Verkehrsfluss findet. Das ist nämlich ziemlich schwierig. Es stellte sich aber heraus, dass es tatsächlich egal war, in welcher Reihenfolge man die Kreuzungen aussucht, von denen man den Überschuss weiterschiebt oder wann man sie anhebt. Sie hat auch erfahren, dass Andrew Goldberg und Robert Tarjan ihn im Jahre 1988 gefunden haben. Jogi steht auf dem Weg zum Stadion trotzdem noch oft im Stau.

## Lösung

Es gibt tatsächlich eine Möglichkeit, bei der nur 19 Schritte benötigt werden, um den besten Verkehrsfluss zu finden:

1. Erhöhen ($A$) auf 1
2. Weiterschieben ($A$): 1 nach $E$
3. Erhöhen ($C$) auf 1
4. Weiterschieben ($C$): 1 nach $E$
5. Weiterschieben ($C$): 1 nach $F$
8. Erhöhen ($B$) auf 1
7. Weiterschieben ($B$): 1 nach $D$
8. Erhöhen ($D$) auf 1
9. Weiterschieben ($D$): 1 nach $F$
10. Erhöhen ($F$) auf 1
11. Weiterschieben ($F$): 2 nach $Z$
12. Erhöhen ($E$) auf 1
13. Weiterschieben ($E$): 2 nach $G$
14. Erhöhen ($G$) auf 1
15. Weiterschieben ($G$): 2 nach $Z$
16. Erhöhen ($A$) auf 10
17. Weiterschieben ($A$): 2 nach $S$
18. Erhöhen ($B$) auf 10
19. Weiterschieben ($B$): 2 nach $S$

## Zum Weiterlesen

1. Kapitel 7 (Tiefensuche)

   Viele Flussalgorithmen sind angelehnt an eine so genannte Tiefensuche oder Breitensuche im Graphen, so auch der Algorithmus von Ford-Fulkerson. Wie eine Tiefensuche in einem Graphen grundsätzlich funktioniert und wie man sie nutzt, ist in diesem Kapitel nachzulesen.

2. Kapitel 9 (Zyklensuche in Graphen)

   In seltenen Fällen kann der Goldberg-Tarjan-Algorithmus irgendwo im Graphen Fluss in einem Kreis herumschicken, ohne dass dies zum effektiven Fluss vom Start zum Ziel beiträgt. Mit einer Zyklensuche kann man diese Kreise nachträglich entfernen, nachdem der maximale Fluss gefunden wurde. In diesem Kapitel steht, wie man so etwas macht.

3. Kapitel 34 (Kürzeste Wege)

   Ein verwandtes Problem zu den maximalen Flüssen ist die Suche nach einem kürzesten Weg. Hier will man nicht etwa einen ganzen Fluss vom Start zum Ziel senden, sondern für ein einzelnes Auto den schnellsten Weg vom Start zum Ziel finden. Wie man diese kürzesten Wege entdeckt, kann man in diesem Kapitel nachlesen.

4. Die 3D-Animation *Flow Commander* unter:
   `http://i11www.iti.uni-karlsruhe.de/adw/jaws/GTVisualizer3D.jnlp`
   (benötigt Java WebStart)

   Warum die Höhe nicht dreidimensional darstellen? Flieg' durch den Graph und schau dir das Schieben und das Anheben am Graphen in 3D an! Wenn auf deinem Rechner Java installiert ist (auf den meisten Rechnern ist das der Fall) kannst bei dieser URL *Flow Commander* installieren und starten.

5. Einer der ersten schnellen Flussalgorithmen: Lestor R. Ford, Jr. und D. R. Fulkerson (1962). *Flows in Networks*. Princeton University Press.

6. Der Originalveröffentlichung des hier gezeigten Algorithmus: Andrew V. Goldberg and Robert E. Tarjan: *A new approach to the maximum-flow problem*. Journal of the ACM 35:921–940, 1988.
   `http://dx.doi.org/10.1145/48014.61051`

7. Der Artikel in der englischen Wikipedia zum Algorithmus von Goldberg und Tarjan:
   `http://en.wikipedia.org/wiki/Push-relabel_algorithm`

8. Der Wikipedia-Artikel zu Flüssen:
   `http://de.wikipedia.org/wiki/Fluss_(Graphentheorie)`

   In diesem Artikel kann man unter anderem nachlesen, wie Flüsse und so genannte *Schnitte* zusammenhängen.

# 37

# Partnerschaftsvermittlung

Volker Claus, Volker Diekert und Holger Petersen

Universität Stuttgart

## Problemstellung

Eine Agentur für die Vermittlung von Partnerschaften versucht, aus einer Menge von Herren und Damen eine möglichst große Zahl von Paaren zusammenzustellen. Vorausgesetzt wird dabei, dass sich die beiden Personen, die ein Paar bilden, gegenseitig sympathisch sind. Weiterhin gehen wir von der klassischen Ehe aus: Jedes Paar besteht aus genau einem Herrn und genau einer Dame, und jede Person tritt in der Menge der Paare höchstens einmal auf.

Etwas abstrakter gesprochen gibt es eine Menge $\mathcal{H}$ von Herren, eine Menge $\mathcal{D}$ von Damen und eine Liste $\mathcal{L}$ der „sympathischen Paare“ der Form $HD$. Ein Eintrag der Form $HD$ bedeutet, dass sich der Herr $H$ und die Dame $D$ wechselseitig sympathisch sind. Das Ziel besteht nun darin, eine möglichst große Zahl von Paaren aus $\mathcal{L}$ zu bilden, wobei jede Person nur höchstens einmal vorkommen darf.

Abbildung 37.1(a) zeigt eine Menge $\mathcal{H} = \{A, B, C, D, E\}$ von 5 Herren und eine Menge $\mathcal{D} = \{P, Q, R, S, T\}$ von 5 Damen. Wenn sich ein Herr und eine Dame wechselseitig sympathisch sind, so verbinden wir sie durch einen Strich (eine so genannte „Kante“). Als Beispiel betrachten wir eine Menge $\mathcal{L}$ mit den Sympathie-Beziehungen $AP$, $AR$, $BP$, $BQ$, $BS$, $CQ$, $CR$, $CT$, $DR$, $DS$, $ES$ und $ET$. Dies ist in Abb. 37.1(b) grafisch angegeben.

Die Agentur könnte nun hieraus die folgenden Paare auswählen: $BP$, $CR$, $ES$. Dann muss sie allerdings aufhören, sie kann kein weiteres geeignetes Paar zusammenführen. Die Agentur hätte aber auch die Paare $AP$, $BQ$, $CR$, $DS$ und $ET$ bilden können, wodurch alle Personen einen Partner erhalten hätten.

Das Problem lautet nun also: Wie findet man eine möglichst große Teilmenge $M$ von $\mathcal{L}$, in der jede Person aus $\mathcal{H}$ und aus $\mathcal{D}$ höchstens einmal vorkommt? Eine solche Teilmenge $M$ nennt man eine größtmögliche Zuordnung (englisch: *maximum matching*) zu $\mathcal{H}$, $\mathcal{D}$ und $\mathcal{L}$.

Wir wollen nun zunächst die Idee und danach den Algorithmus vorstellen, mit dessen Hilfe eine solche größtmögliche Zuordnung konstruiert wird.

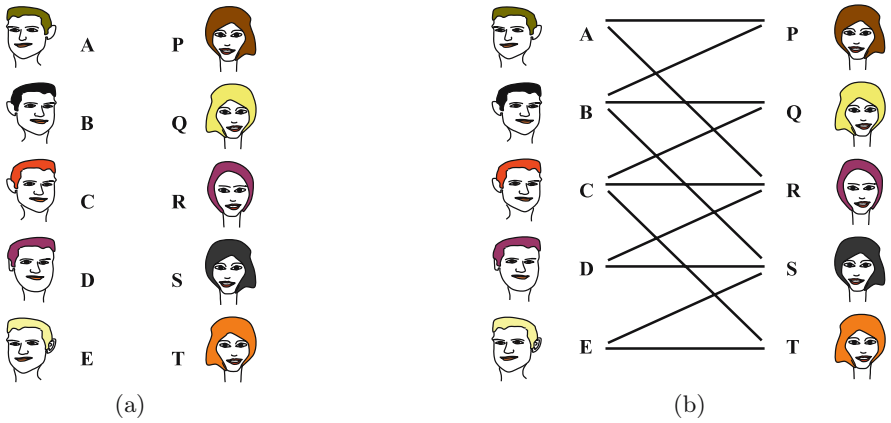



**Abb. 37.1.** Fünf Herren und fünf Damen (a) und ihre Sympathie-Beziehungen (b)

Hierbei ist es unwesentlich, ob der Algorithmus von einem Angestellten der Agentur (unter Verwendung von Karteikarten und anderen Hilfsmitteln) oder durch Anfragen der betroffenen Personen untereinander abgewickelt wird.

## Idee des Verfahrens

Anfangs kann man Personen, die sich sympathisch, aber nicht bereits zugeordnet sind, zu Paaren zusammenfassen. Sobald dies nicht mehr möglich ist, kann es eine partnerlose Person geben. Wir nehmen hier einen partnerlosen Herrn $H$ an, der alle ihm sympathischen Damen fragt, ob diese ihren Partner verlassen können. Diese Damen sind schon jeweils genau einem Herrn zugeordnet und geben daher die Frage an ihren Partner weiter. Im nächsten Schritt fragen nun alle diese Herren alle ihnen sympathischen Damen, ob jene ihren Partner verlassen können. Diese Damen fragen nun die ihnen zugeordneten Herren, ob sie eine neue Partnerin finden können usw. Es läuft also von dem partnerlosen Herrn $H$ eine Welle von Anfragen zu den jeweils sympathischen Damen und von diesen zu den jeweils zugeordneten Herren, die sich immer weiter über die Menge aller Personen ausbreitet, bis eine der folgenden Bedingungen erfüllt ist:

(i) Ein Herr befragt irgendwann eine bisher partnerlose Dame; in diesem Fall bricht die gesamte Anfragewelle sofort ab, man verfolgt von dieser neuen Partnerschaft rückwärts die Anfragekette bis zum Herrn $H$ zurück und tauscht auf genau diesem Weg die bisherigen Partnerschaften.
(ii) Man stellt fest, dass die Anfragewelle nur Damen erreicht, die bereits Partner haben; in diesem Fall streicht man den Herrn $H$ aus der Menge der Herren.

Im Folgenden orientiere man sich an diesem Bild einer Welle von Anfragen, die sich von $H$ über alle Personen ausbreitet, wobei abwechselnd eine Sympathiebeziehung und eine bereits getroffene Zuordnung durchlaufen werden. Sobald eine neue Partnerschaft entdeckt ist, bricht diese Welle schlagartig zusammen und auf dem Weg der Anfragen von dieser neuen Partnerschaft zu $H$ werden die Paare umgeordnet.

*Hinweis:* Diese Welle kann gleichzeitig oder in irgendeiner Reihenfolge abgearbeitet werden, am anschaulichsten ist aber die sich parallel gleichförmig ausbreitende Welle (in der Informatik spricht man von einem „Breitendurchlauf“ bzgl. $\mathcal{L}$). Hierbei fragt man keine bereits befragte Person noch einmal, sodass sich die Welle immer nur auf noch nicht betrachtete Personen ausdehnt.

## Die Konstruktion einer größtmöglichen Zuordnung

Man beginne mit einem beliebigen Paar aus der Liste $\mathcal{L}$. Dieses Paar sei $HD$ und man setze $M = \{HD\}$. (Ist die Liste $\mathcal{L}$ leer, so lassen sich natürlich keine Paare bilden und es gibt nichts zu tun.) Beachte, dass in der Menge $M$ im Folgenden jede Person höchstens einmal vorkommt.

Wir können ab jetzt annehmen, dass bereits eine Menge von Paaren $M = \{H_1D_1, H_2D_2, \ldots, H_rD_r\}$ konstruiert wurde. Kommen alle Herren in $M$ vor, so ist man fertig, denn Bigamie ist verboten. Interessant ist also die Situation, in der ein Herr $H$ ohne Partnerin ist, also noch nicht in $M$ vorkommt. Wie kann man ihm eine Partnerin $D$ zuordnen?

*(a)* Offensichtlich muss man zur Person $H$ jede Person $D$, die ihr sympathisch ist (d.h., $HD$ ist in der Liste $\mathcal{L}$ enthalten), befragen, ob sie ebenfalls noch keinen Partner hat. Gibt es eine solche Person $D$ ohne Partner, so füge man einfach $HD$ zur Menge $M$ hinzu und wende den Algorithmus erneut auf eine andere partnerlose Person an.
Dies haben wir ausgehend vom Paar $BP$ durch Hinzunahme der Paare $CR$ und $ES$ in Abb. 37.2 durchgeführt. Die Menge $M = \{BP, CR, ES\}$ ergibt sich aus den roten Kanten. Fall (a) trifft nun aber nicht mehr zu und wir können $M$ auf diese Weise nicht mehr vergrößern, obwohl es noch partnerlose Personen gibt.

*(b)* Was tut man, wenn es zu der partnerlosen Person $H$ keine partnerlose Person $D$ in $\mathcal{L}$ gibt? Dann ist also jeder Person $D$, die $H$ sympathisch findet, bereits ein Partner $H'$ zugeordnet. Jetzt wird man versuchen, dem Herrn $H'$ die Dame $D$ zugunsten von $H$ wegzunehmen; d.h., man strebt an, $H'D$ durch $HD$ zu ersetzen, wodurch nun $H'$ partnerlos wird.
Nun versucht man, eine der Damen $D'$ (aber nicht die Dame $D$), die $H'$ sympathisch sind, ihrem Partner $H''$ wegzunehmen, wodurch $H''$ partnerlos wird und einen anderen Partner erhalten muss usw. Hierbei werden nur solche Damen $D'$, $D''$, $D'''$ usw. betrachtet, die bisher noch nicht einbezogen wurden. Ausgehend von $H$ wird auf diese Weise eine abwechselnde Folge von

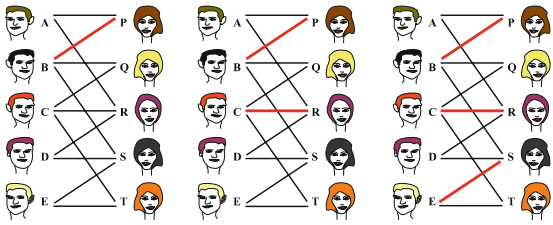
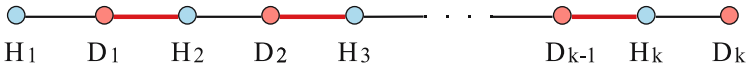



**Abb. 37.2.** Aufbau der Menge $M$ nur mit dem Kriterium (a) des Algorithmus

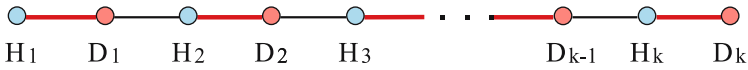
Sympathiebeziehungen $HD$ und bereits zugeordneten Beziehungen $H'D$ aus der Menge $M$ durchlaufen. Irgendwann tritt entweder Fall (a) ein oder die Iteration endet damit, dass alle hierbei erreichbaren Personen erfolglos durchprobiert worden sind.
Wenn Fall (a) eintritt, dann hat man eine neue Menge von Paaren erhalten, indem man genau die Ersetzungen „$H'D$ wurde durch $HD$ ersetzt“, die zum Fall (a) hinführen, tatsächlich vornimmt, während alle anderen Paare in $M$ unverändert bleiben. Präziser formuliert: Wenn Fall (a) erstmals eintritt, dann gibt es genau eine Folge (beginnend mit $H = H_1$)

$$H_1, D_1, H_2, D_2, \ldots H_k, D_k$$

mit $H_1$ ist partnerlos, $H_1D_1 \in \mathcal{L}, H_2D_1 \in M, H_2D_2 \in \mathcal{L}, \ldots, H_kD_{k-1} \in M$, $H_kD_k \in \mathcal{L}$ und $D_k$ ist partnerlos. Die Situation mit der partnerlosen Dame $D_k$




ermöglicht also, die roten Kanten in $M$ durch die schwarzen Kanten zu ersetzen („Umfärben“ der roten und schwarzen Kanten auf diesem Weg):




wobei $M$ um ein Paar wächst und alle Personen, die bisher bereits einen Partner besaßen, erneut einen Partner haben.

Mit dieser um 1 vergrößerten Menge $M$ startet man dann den Algorithmus erneut.

In unserem Beispiel führt dieses Vorgehen zum Erfolg (siehe Abb. 37.3 bis Abb. 37.7): Bisher wurden die Paare $BP$, $CR$ und $ES$ ausgewählt. Weitere Paare sind nicht mehr möglich. Also müssen jetzt Paare umgeordnet werden. Hierzu gehen wir in Abb. 37.3 von Herrn $A$ aus, der noch keine Partnerin hat. Wir ordnen ihm eine seiner möglichen Partnerinnen $P$ bzw. $R$ zu, wodurch

nun die Herren $B$ (bisheriger Partner von $P$) bzw. $C$ (bisheriger Partner von $R$) partnerlos werden; mit jedem von diesen wird nun erneut versucht, eine neue Partnerin zu finden (Abb. 37.4) usw. Dabei behandelt man jede Person höchstens einmal, d.h., wenn man bei dem Verfahren auf eine Person trifft, die bereits untersucht wurde, so betrachtet man diese nicht noch einmal. Tritt nun der Fall (a) ein, so erhält man eine neue Menge $M$, wie in Abb. 37.7 angegeben.

*(c)* Gelingt es auf diese Weise dagegen nicht, irgendwann auf den Fall (a) zu stoßen, so besagt ein mathematischer Satz, dass dann die Person $H$ aus dem weiteren Verfahren ausgeschlossen werden darf.

Dieser Sachverhalt ist im Prinzip recht gut zu verstehen. Denn was passiert, wenn die Menge $M$ nicht vergrößert wurde? Dann gibt es eine Anzahl von $d$ Damen, die ausgehend von $H$ insgesamt befragt wurden. Jede dieser $d$ Damen hat auch nach der Umordnung einen Partner, dies sind also $d$ Herren mit Partnerin. Nur der letzte der Herren, der seine Partnerin verlor, hat

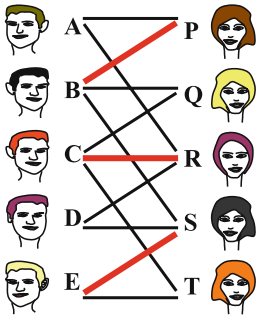



**Abb. 37.3.** $A$ ist partnerlos. $A$ kann mit $P$ oder mit $R$ zu einem Paar zusammengefasst werden. Wir beginnen mit $P$. Deren Partner ist $B$. Versuchsweise ersetzen wir in Abb. 37.4 nun $BP$ durch $AP$, angedeutet durch gestrichelte Kanten

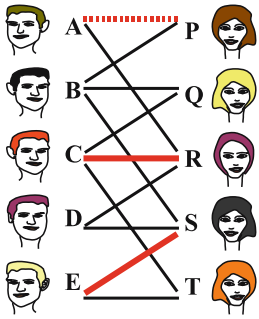



**Abb. 37.4.** Jetzt ist $B$ partnerlos. Da in $\mathcal{L}$ das Paar $BQ$ mit der partnerlosen Dame $Q$ liegt, tritt nun bereits Fall (a) ein

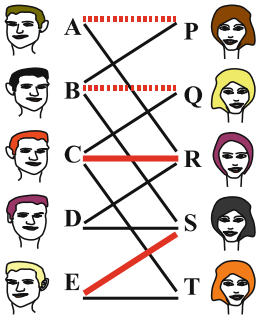



**Abb. 37.5.** Das „versuchsweise“ Umordnen wird daher endgültig übernommen und wir erhalten die Zuordnung $\{AP, BQ, CR, ES\}$, wobei $BP$ durch $AP$ und $BQ$ ersetzt wurde

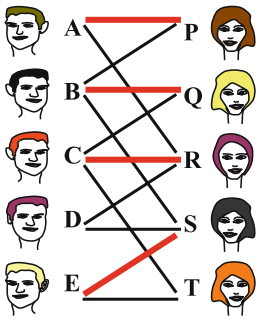



**Abb. 37.6.** Nun starten wir das Verfahren erneut mit der partnerlosen Person $D$ (beachte: In diesem Beispiel ist $D$ der Name eines Herrn). Versuchsweise nehmen wir $DR$ auf und entfernen hierfür $CR$. Die jetzt partnerlose Person $C$ kann mit der partnerlosen Person $T$ ein Paar bilden

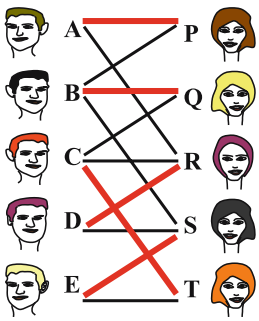



**Abb. 37.7.** Wir haben schließlich die Menge $M = \{AP, BQ, CT, DR, ES\}$ erhalten. Hier hat jede Person einen Partner und daher endet der Algorithmus. (Das versuchsweise Umordnen dauert maximal so viele Schritte, wie es Kanten gibt, da einmal ausprobierte Personen nicht erneut getestet werden.)

jetzt keine. Also gibt es $d+1$ Herren, die in dieser Reihe vorkamen. Und hier kommt die entscheidende Feststellung: Die Gesamtheit der möglichen Partnerinnen dieser $d+1$ Herren bildet genau die Menge dieser $d$ Damen. Folglich muss immer ein Herr partnerlos bleiben. Also können wir ihn von Anfang an streichen. Diese Argumentation wird weiter unten nochmals erläutert.

Wir halten fest: Wenn $M$ eine größtmögliche Zuordnung ist, so gibt es auch eine größtmögliche Zuordnung $M'$, die gleich viele Elemente wie $M$ besitzt, aber $H$ nicht enthält. Man kann daher auf $H$ verzichten. Man streiche also $H$ und führe den Algorithmus mit einer anderen partnerlosen Person erneut durch. (Übrigens hätten wir statt $H$ dann auch einen der letzten Herren aus einer versuchsweisen Umordnung streichen können. Dies wird man in der Praxis selten tun. Zunächst erhalten die aus Sicht der Agentur wichtigen Personen einen Partner. Bei dem hier beschriebenen Vorgehen ist garantiert, dass einmal vermittelte Personen auch am Ende nicht partnerlos sind.)

*(d)* Das Verfahren endet, wenn jede nicht gestrichene Person einen Partner hat, also in der Menge $M$ enthalten ist. Diese Menge $M$ ist dann eine größtmögliche Zuordnung.

Also: Wie erfolgt die schrittweise Erhöhung der Anzahl der Paare in der Zuordnungsmenge $M$ (vgl. Abb. 37.3 bis 37.7)? Die roten Kanten gehören zu den Paaren der Menge $M$. Die schwarzen Kanten bilden alle übrigen Sympathiebeziehungen ($\mathcal{L} - M$). Nun wird die Menge $M$ jedes Mal umgeordnet. Hierzu sucht man einen Weg aus abwechselnd schwarzen und roten Kanten, der mit einer schwarzen Kante startet und endet und an deren Anfang und Ende je eine partnerlose Person stehen. Liegt dieser Fall vor, so kann man in $M$ die roten Kanten durch die schwarzen Kanten genau dieses Weges ersetzen, wobei sich die Anzahl der Paare um 1 erhöht.

*Hinweis:* Solche Wege aus abwechselnd schwarzen und roten Kanten, deren Endknoten keine ausgehenden roten Kanten haben, heißen „erweiterbare Wege“ (englisch: *augmenting paths*).

## Der Algorithmus

Der folgende Algorithmus PARTNERVERMITTLUNG liefert eine maximal große Menge $M$ von Paaren, wenn die Mengen $\mathcal{H}$, $\mathcal{D}$ und $\mathcal{L}$ gegeben sind. Es ist klar, dass man von den beiden Mengen $\mathcal{H}$ und $\mathcal{D}$ nur eine Menge betrachten muss, um den nächsten partnerlosen Kandidaten auszuwählen. Wir beschränken uns hier auf die Menge $\mathcal{H}$ (siehe Zeile 2 im Algorithmus); in der Praxis wird man die kleinere der beiden Mengen $\mathcal{H}$ und $\mathcal{D}$ nehmen.

Gegeben seien die Mengen $\mathcal{H}$ und $\mathcal{D}$ und die Menge $\mathcal{L}$ als Menge von Paaren $HD$ für gewisse $H$ aus $\mathcal{H}$ und $D$ aus $\mathcal{D}$.

Der Algorithmus PARTNERVERMITTLUNG berechnet eine maximale Menge von Paaren $M$

```
1   wähle irgendein Paar HD aus ℒ aus; M := {HD};
2   while es gibt noch eine partnerlose Person H in ℋ do
3       verfolge von H ausgehend alle Wege, die abwechselnd aus einer Kante
        aus ℒ, die nicht in M liegt, und aus einer Kante aus M bestehen und
        die keine Person mehrfach enthält;
4       if man stößt hierbei auf eine partnerlose Person
                (diese liegt notwendigerweise in 𝒟)
5       then ersetze in M alle auf diesem Weg in M liegende Kanten durch
        die auf diesem Weg nicht in M liegenden Kanten;
6       else     (in diesem Fall gibt es keinen solchen Weg)
7           entferne H aus ℋ
8       end if
9   end while;
10  return M;
```

Das **Ergebnis** des Algorithmus ist eine größtmögliche Zuordnung M.

Im Inneren der while-Schleife ist eine systematische Suche im Teil „verfolge von $H$ ausgehend alle Wege, ...“ zu realisieren. Dieser Teil wird „rekursiv“ implementiert. Schon auf Seite 375, Teil (b), hatten wir geschrieben: „Nun versucht man eine der Damen $D'$ ...“. Hierin steckt genau die Rekursion, dass man das gleiche Verfahren mit $H'$ anstelle von $H$ durchzuführen hat, sofern $H'$ nicht bereits behandelt wurde. Hierbei sollte man ein Boolesches Feld mitführen, dessen Komponenten zu Beginn der while-Schleife (vor Zeile 3 im Programm) auf *false* gesetzt werden und in welchem man sich merkt, ob man eine Person in dieser Iteration bereits betrachtet hat oder nicht. Wie man das Gesamtverfahren nun implementiert, lässt sich in Büchern nachlesen. (Am Schluss dieses Beitrags finden sich Literaturhinweise.)

## Der Heiratssatz

Dass das Verfahren korrekt arbeitet, beruht auf dem *Heiratssatz* des englischen Mathematikers Philip Hall aus dem Jahr 1935.

Nach seinem Satz existiert eine Zuordnung aller Herren zu passenden Damen genau dann, wenn folgende Heiratsbedingung gilt: Für jede Teilmenge von Herren gibt es eine mindestens ebenso große Teilmenge von möglichen Partnerinnen.

Die Bedingung bedeutet, dass wenn z. B. 17 Herren aus der Menge $\mathcal{H}$ betrachtet werden, dann auch mindestens 17 potenzielle Partnerinnen für sie zur Verfügung stehen. Statt 17 dürfen wir auch jede andere Zahl einsetzen. Hierbei geht es zunächst nur um das richtige Verhältnis der Zahlen. Auf die Zuordnung kommt es noch nicht an.

Das Kriterium des Heiratssatzes eignet sich nicht direkt, um auch eine Lösung zu finden. Erstens würde die Überprüfung bei je 50 Damen und Herren mehr als eine Million Jahre dauern, selbst wenn pro Sekunde eine Milliarde Kombinationen von Herren geprüft werden könnten (denn man müsste ja alle $2^{50} = 1.125.899.906.842.624$ Teilmengen durchprobieren). Zweitens liefert der Satz nur die Information, ob eine Lösung existiert, aber nicht wie sie aussieht. Hierfür benötigen wir den oben beschriebenen Algorithmus.

## Wo braucht der Algorithmus den Heiratssatz?

Dies geschieht an der Stelle, an der man mindestens einen weiteren Herrn finden muss, dessen Dame man neu zuordnet und der die Möglichkeit haben muss, eine Ersatz-Dame finden zu können.

Wir erklären dies an einem Beispiel:

Betrachte die Abb. 37.8. Es sei $M = \{H_2D_1, H_4D_2, H_3D_3\}$. $H$ ist partnerlos. Wir starten mit $H$.

$\{H\}$ bildet eine Teilmenge $T_1$. Um weiter zu kommen, muss die zu $T_1$ gehörende Teilmenge $\{D_1, D_2\} = T_2$ (= alle mit $H$ verbundenen Personen) mindestens so groß wie $|T_1| = 1$ sein (Heiratssatz!), siehe Abb. 37.9.

Nun ersetzen wir probeweise $H_2D_1$ durch $HD_1$, wodurch $H_2$ partnerlos wird. Wir können aber auch $H_4D_2$ durch $HD_2$ ersetzen, wodurch $H_4$ partnerlos wird. Insgesamt müssen wir also Partnerinnen für $T_3 = \{H, H_2, H_4\}$ finden. Die zu $T_3$ gehörende Menge von Damen ist $T_4 = \{D_1, D_2, D_3, D_6\}$, siehe Abb. 37.10. Erneut gilt $|T_3| \leq |T_4|$, sodass es nach dem Heiratssatz hier eine Zuordnung geben muss. Diese spüren wir auf, indem wir die zu den Damen aus $T_4$ gehörenden Herren weiter untersuchen usw.

Wenn es also eine Lösung gibt, so sorgt der Heiratssatz dafür, dass der Algorithmus stets an irgendeiner Stelle mit einer noch nicht betrachteten Person fortfahren kann.

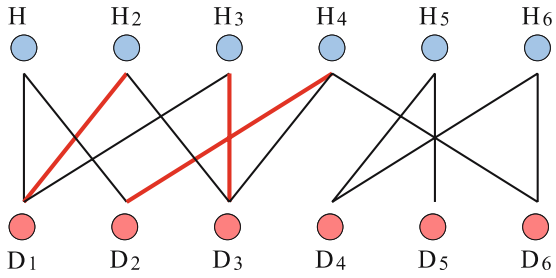



**Abb. 37.8.** $M = \{H_2D_1, H_3D_3, H_4D_2\}$, $H$ ist partnerlos

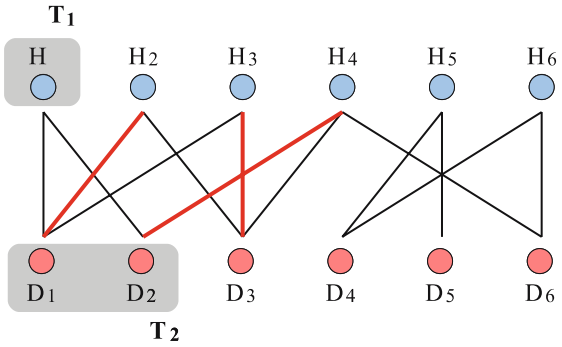



**Abb. 37.9.** $T_1$ und die zugehörige Teilmenge $T_2$

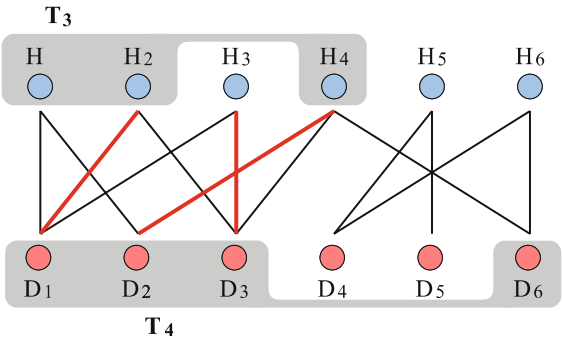



**Abb. 37.10.** $T_3$ und die zugehörige Teilmenge $T_4$

## Zeitanalyse

Die Laufzeit des Verfahrens lässt sich leicht nach oben abschätzen. Wir nehmen an, dass es $n$ Herren und $n$ Damen sowie $m$ Kanten (= Anzahl der Elemente in der Menge $\mathcal{L}$) gibt. Die while-Schleife im Algorithmus wird spätestens nach $m$ Schritten beendet sein, da jede Person höchstens einmal berücksichtigt wird und deshalb jede Kante nur höchstens einmal betrachtet werden muss. Weil nach jedem Schleifendurchlauf entweder ein Herr ausscheidet oder ein weiteres Paar hinzukommt, endet das gesamte Verfahren nach spätestens $(n-1)$-maliger Ausführung der while-Schleife. Als Laufzeitabschätzung erhalten wir also $n \cdot m$ als obere Schranke für die Zahl der Schritte, die vom Algorithmus ausgeführt werden.

Gibt es schnellere Methoden, um eine maximale Menge von Paaren zu berechnen? Ein Indiz für Zeitverschwendung in unserem Algorithmus ist, dass zu Beginn eines Schleifendurchlaufs keinerlei Information vorliegt, obwohl frühere Durchläufe Informationen über den Verlauf von erweiterbaren Wegen gesammelt hatten. Nutzt man dies geschickt aus, so lässt sich eine Laufzeit

proportional zu $m \cdot \sqrt{n}$ erreichen, man kann also den Faktor $\sqrt{n}$ einsparen. Dieses beschleunigte Verfahren wurde schon 1971 von den amerikanischen Forschern John E. Hopcroft und Richard M. Karp entwickelt. Die beiden Forscher haben für diese und viele andere Leistungen 1986 und 1985 jeweils den Turing-Award erhalten, eine Art Nobelpreis der Informatik.

## Zum Weiterlesen

1. Kapitel 36 (Maximale Flüsse)

   In diesem Kapitel wird das Problem des maximalen Flusses behandelt. Eine Lösung dieses Problems kann verwendet werden, um eine maximale Menge von Paaren zu bestimmen. Weiterhin lässt sich das hier vorgestellte Verfahren auf beliebige Graphen verallgemeinern (selbst überlegen oder z. B. in dem folgenden Buch nachlesen).

2. Thomas Ottmann, Peter Widmayer: *Algorithmen und Datenstrukturen.* Spektrum Akademischer Verlag, 2002.

   Ein auf Deutsch geschriebenes Standardlehrbuch. In dem Kapitel *Zuordnungsprobleme* werden größtmögliche Zuordnungen mittels der Berechnung von maximalen Flüssen gefunden.

3. Reinhard Diestel: *Graphentheorie.* Springer, 2000.

   Kapitel 1 behandelt sehr ausführlich Paarungen in bipartiten und allgemeinen Graphen. Dort ist auch ein Beweis des Heiratssatzes zu finden.

4. Dexter C. Kozen: *The Design and Analysis of Algorithms.* Springer, 1992.

   Dieses Buch enthält 40 Kapitel zu Themen der Algorithmentheorie, die jeweils einer Vorlesung entsprechen. In den Kapiteln 19 und 20 wird das effizientere Verfahren von Hopcroft und Karp mit den zugehörigen Korrektheitsbeweisen vorgestellt.

5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2001. Section 26.3: Maximum bipartite matching, pp. 664–669.

   Dies ist ein sehr ausführliches und sehr erfolgreiches Buch zur Algorithmentheorie. Eine deutsche Übersetzung ist unter dem Titel *Algorithmen – Eine Einführung* im Oldenbourg Verlag erschienen.

6. In der freien Enzyklopädie Wikipedia finden sich mehrere Artikel zum *Matching-Problem* und dem *Heiratssatz.*

## Danksagung

Wir danken Botond Draskoczy und Sascha Riexinger für die technische Unterstützung.

# 38

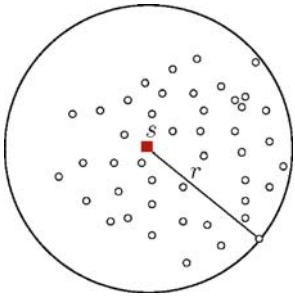
# Kleinster umschließender Kreis (Ein Demokratiebeitrag aus der Schweiz?)

Emo Welzl

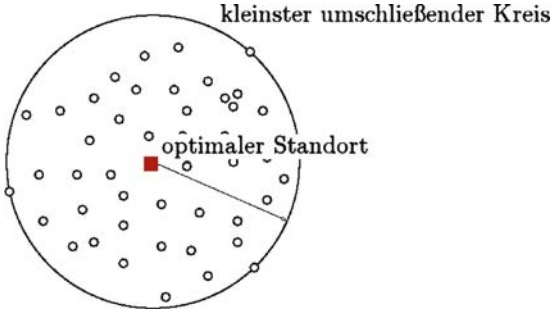
ETH Zürich, Schweiz

Für die Feuerwehr soll ein neuer Standort gefunden werden, der für die Häuser, für die sie zuständig ist, optimal gewählt ist. Die Qualität des neuen Standorts misst sich an der größten Distanz, die zu einem der relevanten Häuser auftritt, wobei diese größte Distanz natürlich möglichst klein sein sollte. Wir idealisieren die Häuser und den neuen Standort der Feuerwehr zu Punkten in der Ebene und wählen für die Distanz zwischen zwei Punkten deren Abstand. Die Eingabe unseres Problems ist also eine Menge $P$ von Punkten in der Ebene.

Betrachten wir einen Punkt $s$ als potenziellen Standort. Den Abstand des von $s$ am weitesten entfernten Punktes aus $P$ bezeichnen wir mit $r$. Dann umschließt der Kreis um $s$ mit Radius $r$ alle Punkte in $P$.




Es wird ersichtlich, dass der beste Standort der Mittelpunkt des Kreises mit kleinstem Radius ist, der alle Punkte in $P$ umschließt. (Von dort kann die Feuerwehr selbst das entlegenste Haus so schnell wie möglich erreichen.) Ein solcher kleinster Kreis existiert und er ist eindeutig – nehmen wir das als gegeben hin.




Der Häuser gibt es viele und man überlegt sich, wie man den besten Standort bestimmen soll.

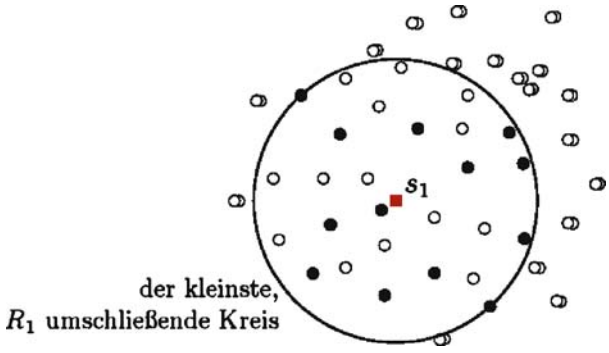
Jemand hat die Idee, Vertreter einer kleinen zufälligen Auswahl $R_1$ von, sagen wir, 13 Häusern zusammenkommen zu lassen, um den für sie besten Standort zu ermitteln – ohne jegliche Rücksicht auf die anderen Häuser. Eine Literaturrecherche bringt nämlich eine Methode zum Vorschein, die das Problem zwar für 13 Häuser schnell löst, das Verfahren erweist sich aber leider für alle Häuser als viel zu langsam.

Der Vorschlag wird aufgegriffen, das Ergebnis ist ein Standort $s_1$ und ein Radius $r_1$, sodass der Kreis mit Mittelpunkt $s_1$ und Radius $r_1$ alle ausgewählten Häuser umschließt – der kleinste, $R_1$ umschließende Kreis.

*Ein erster Standort ist gefunden.*

Obwohl die Menge $R_1$ zufällig gewählt wurde, ist der Protest gegen diesen neuen Standort groß, vor allem unter den Besitzern der Häuser, die außerhalb des ermittelten Kreises liegen.

Dem Protest wird stattgegeben, man entschließt sich zu einer zweiten Runde, aber nach wie vor hat man keine Ahnung, wie für die riesige Menge an Punkten der kleinste Kreis berechnet werden soll – für 13 war es schon schwierig genug. Auch der Vorschlag, alle einzuladen, die außerhalb des ersten Kreises liegen, erscheint nicht realistisch. Es kommt zu folgendem Kompromiss: Bei der Verlosung der 13 Vertreter haben alle, deren Häuser außerhalb jenes ominösen Kreises lagen, zwei Zettel im Lostopf.



Es werden 13 Vertreter gezogen, die Menge $R_2$, sie treffen sich, bestimmen ihren besten Standort $s_2$ und den entsprechenden Radius $r_2$ des kleinsten Kreises, der $R_2$ umschließt.

Wenig verwunderlich, auch dies fordert Widerstand heraus, wieder gibt es viele Häuser, die außerhalb des Kreises der für $R_2$ ermittelten Lösung liegen.

Man muss nun wissen, dass die Gemeindeleitung die Finanzierung des neuen Gebäudes für die Feuerwehr noch nicht gesichert hat. Sie findet daher Gefallen an dem Entscheidungsprozess – d.h. eigentlich Gefallen daran, dass der Prozess wohl nicht so schnell zu einem alle befriedigenden Ergebnis führen kann, wenn überhaupt.

Die Lösung der zweiten Runde wird also wieder verworfen. Für die nächste Runde werden für alle, deren Haus in der zweiten Runde außerhalb des Kreises lagen, die Zettel verdoppelt. Liegt ein Haus in beiden bislang ermittelten Lösungen außerhalb der jeweiligen Kreise, liegen dafür immerhin schon 4 Zettel im Topf!

Runde drei verläuft wie gehabt.

*Und so weiter und so weiter.*

Es kehrt Routine ein. Die Kreisfindungstreffen entpuppen sich als beliebte Unterhaltung, auch weil die Gemeinde Speis und Trank zur Verfügung stellt. Außerhalb des jeweils veröffentlichten Kreises zu liegen wird gar nicht mehr als Enttäuschung empfunden, weil der Vorschlag sowieso nicht realisiert wird und sich so die Chancen für eine Teilnahme am nächsten Treffen vergrößern. Der Lostopf schwillt an, aber der Gemeindesekretär hat schnell ein elektronisches Losverfahren eingerichtet (animiert durch das Kap. 25 über Zufallszahlen).

Doch dann, nachdem sich wieder einmal 13 Vertreter getroffen haben und die für sich, und nur für sich, beste Lösung vorschlagen, geschieht das eigentlich Unerwartete. Kein Haus liegt außerhalb des berechneten Kreises. Schnell macht die Kunde die Runde, ein eilig bestelltes Gutachten bestätigt, was alle vermutet haben: Das muss jetzt der kleinste, *alle Punkte* umschließende Kreis sein. Schließlich kann der kleinste Kreis, der alle umschließt, nicht kleiner sein, als der, der nur 13 der Punkte umschließt.

*Der optimale Standort ist gefunden!*

War es Glück, dass das gewählte Verfahren zum Erfolg führte, oder war es zu erwarten? Letzteres: Wir haben einen von Kenneth Clarkson entwickelten randomisierten (d.h. zufallsbasierten) Algorithmus kennengelernt, der für $n$ Punkte den kleinsten umschließenden Kreis berechnet. Man kann zeigen, dass das Verfahren mit Wahrscheinlichkeit 1 diesen Kreis berechnet, die erwartete Anzahl von Runden ist sogar nur logarithmisch in $n$. Dabei ist es wichtig, dass man die Größe der jeweils zufällig gezogenen Teilmenge, in unserer Geschichte 13, nicht zu klein wählt – aber 13 genügt, unabhängig davon, wie groß $n$ ist. Auch kann man genauso kleinste Kugeln um Punkte im 3-dimensionalen

Raum oder auch in höheren Dimensionen berechnen (nur die zufällige Auswahl muss man abhängig von der Dimension etwas größer wählen).

## Warum es funktioniert

Wer verwegen genug ist, dem sei noch in groben Zügen verraten, warum das wirklich so klappt. Dazu müssen wir erst die Struktur des Problems etwas besser verstehen. Der kleinste, die Punktemenge $P$ umschließende Kreis – höchste Zeit, dass er einen Namen bekommt: $K(P)$ – ist durch höchstens 3 der Punkte in $P$ bestimmt. Genauer gesagt, es gibt eine Menge $B$ von höchstens 3 Punkten in $P$, für die $K(B) = K(P)$ gilt. Beachte: Wenn für eine Teilmenge $R$ von $P$ nicht schon $K(R) = K(P)$ gilt, dann muss einer der Punkte in $B$ außerhalb von $K(R)$ liegen. In unserem Verfahren heißt das, dass in jeder Runde mindestens einer der Punkte in $B$ seine Zettel im Lostopf verdoppelt, und folglich hat nach $k$ Runden mindestens einer von ihnen mindestens $2^{k/3} \approx 1.26^k$ Zettel im Topf. Das wächst also ganz ordentlich an.

Andererseits kann man zeigen, dass durch die zufällige Auswahl im Durchschnitt nicht zu viele neue Zettel in den Topf kommen. Es ist ja so, dass genau die Zettel von den Häusern außerhalb des aktuellen Kreises verdoppelt werden. Diese Anzahl ist im Durchschnitt etwa $3z/13$, wenn gerade $z$ Zettel im Topf sind (den Beweis für diese Behauptung bleiben wir schuldig). Das heißt, wir erwarten in der nächsten Runde etwa $(1 + 3/13)z \approx 1.23z$ Zettel im Topf. (Richtig geraten: Die „3“ kommt von der Größe von $B$ und die „13“ von der Auswahlgröße, für die wir uns entschieden hatten.)

Einerseits vermehrt sich also die Anzahl der Zettel pro Runde etwa um den Faktor 1.23, d.h., es gibt etwa $n \cdot 1.23^k$ Zettel nach $k$ Runden ($n$ ist die Anzahl der Punkte). Andererseits gibt es einen Punkt, der nach $k$ Runden mindestens $1.26^k$ Zettel im Topf hat. Egal wie groß $n$ ist, das ist wegen $1.26 > 1.23$ früher oder später mehr als es insgesamt eigentlich geben dürfte. Es gibt eine einzige Möglichkeit, dieses Paradoxon aufzulösen: Das Verfahren ist vorher zu einem Ende gekommen.

Das ist vielleicht verwirrend. Aber so sind sie nun einmal, die randomisierten Algorithmen: an sich einfach – und zugleich verblüffend, dass es klappt.

## Zum Weiterlesen

1. Kenneth L. Clarkson: *A Las Vegas Algorithm for linear and integer programming when the dimension is small.* Journal of the ACM 42(2): 488–499, 1995.

   Dies ist die Originalarbeit, in der das hier beschriebene Verfahren entwickelt und analysiert wurde.

2. Kapitel 25 (Zufallszahlen)

   Hier erfährt man, wie man auf einem Rechner die für das Verfahren notwendigen Zufallszahlen generiert.

# 39

# Online-Algorithmen: Was ist es wert, die Zukunft zu kennen?

Susanne Albers und Swen Schmelzer

Albert-Ludwigs-Universität Freiburg

## Das Ski-Problem

In diesem Jahr möchte ich nach längerer Pause wieder in den Skiurlaub fahren. Leider sind mir meine alten Skier zu klein, und so stehe ich vor der Frage, ob ich mir Skier leihen oder doch lieber kaufen soll. Wenn ich sie mir leihe, so muss ich pro Tag 10 Euro Leihgebühr zahlen. Für die 7 Tage meines Urlaubs würde ich also 70 Euro zahlen. Wenn ich mir Skier kaufe, so fallen Kosten von 140 Euro an. Vielleicht fahre ich aber in den nächsten Jahren öfter in den Skiurlaub. Da wäre es ja besser, wenn ich mir ein Paar kaufte. Wenn ich aber keinen Spaß mehr daran habe, wäre es besser, nur die geringeren Leihkosten zu zahlen. Abbildung 39.1 zeigt die Gesamtkosten für das einmalige Kaufen und das tägliche Leihen, wenn man an genau $x$ Tagen Skier benötigt.

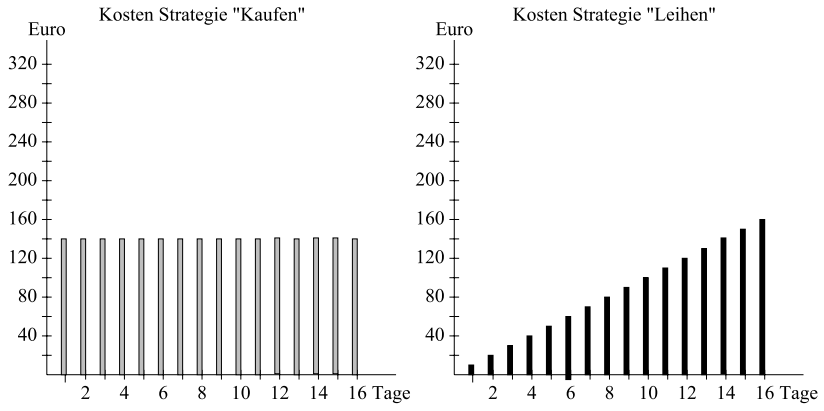



**Abb. 39.1.** Kosten der Strategien „Kaufen“ bzw. „Leihen“

Ohne zu wissen, wie oft ich Ski fahren gehe, kann ich wohl nicht verhindern, etwas mehr zu bezahlen. Hinterher ist man immer schlauer. Aber wie kann ich verhindern, dass ich hinterher sage: „Ich hätte ja für weniger als die Hälfte Ski fahren können?“ Probleme dieser Art bezeichnet man in der Informatik als *Online-Probleme*. Bei diesen muss man Entscheidungen treffen, ohne die Zukunft zu kennen. In unserem Fall wissen wir nicht, wie oft wir in der Zukunft Ski fahren werden, müssen aber entscheiden, ob wir Skier leihen oder kaufen. Würden wir die Zukunft kennen, so könnten wir leicht eine Entscheidung treffen. Fahren wir an weniger als 14 Tagen Ski, so ist es preiswerter, Skier zu leihen. Bei genau 14 Tagen bezahlt man in beiden Fällen gleich viel. Bei mehr als 14 Tagen sollten wir die Skier kaufen. Hat man vollständige Kenntnis über die Zukunft, so spricht man auch von einem *Offline-Problem*. In diesem Fall können wir, wie oben beschrieben, leicht die minimalen Kosten bestimmen. Wir können die minimalen Kosten mit einer Funktion $f$ beschreiben. Dabei gibt $x$ die Anzahl der Tage an, an denen wir Skier benötigen.

$$f(x) = \begin{cases} 10x, \text{ falls } x < 14 \\ 140, \text{ falls } x \geq 14 \end{cases}$$

In einem Online-Algorithmus muss man zu jedem Zeitpunkt mit dem bisherigen Kenntnisstand eine Entscheidung treffen. Wenn wir uns am ersten Tag entscheiden, Skier zu leihen, so stehen wir am zweiten Tag wieder vor der gleichen Entscheidung. Wenn man mit einer Online-Strategie nie mehr als das Doppelte dessen bezahlt, was man bezahlt hätte, wenn man die Zukunft gekannt hätte, nennt man das 2-kompetitiv (wettbewerbsfähig). Allgemein heißt eine Online-Strategie *k-kompetitiv*, wenn man nie mehr als das $k$-fache dessen bezahlt, was man bezahlt hätte, wenn man die Zukunft gekannt hätte. Es wird $k$ auch der kompetitive Faktor genannt. Wir wollen für unser Problem eine 2-kompetitive Online-Strategie entwickeln.

**Online-Strategie für das Ski-Problem:** Ich leihe mir am Anfang Skier aus. Würden durch das erneute Leihen von Skiern die Gesamtleihkosten den Kaufkosten entsprechen, so kaufte ich die Skier. In unserem Beispiel würden wir die ersten 13 Tage Skier leihen und am 14. Tag kaufen.

In der Abb. 39.2 ist links der optimale Wert (die Funktion $f$) in grau dargestellt. In schwarz sind im rechten Koordinatensystem die Kosten unseres Online-Algorithmus aufgetragen. An der Grafik ist leicht zu erkennen, dass wir nie mehr als doppelt so viel wie das Optimum zahlen. Geht man an weniger als 14 Tagen fahren, zahlt man so viel, wie man auch bezahlt hätte, wenn man vorher gewusst hätte, wie oft man fahren wird. Geht man öfter, so bezahlt man nie mehr als das Doppelte.

Unser Problem haben wir damit zufriedenstellend gelöst. Was aber passiert bei anderen Leih- und Kaufkosten? Müssen wir unsere Strategie dann ändern? Um zu erkennen, dass die vorgeschlagene Strategie für *beliebige Werte* von Leih- und Kaufkosten 2-kompetitiv ist, normieren wir die Kosten so,

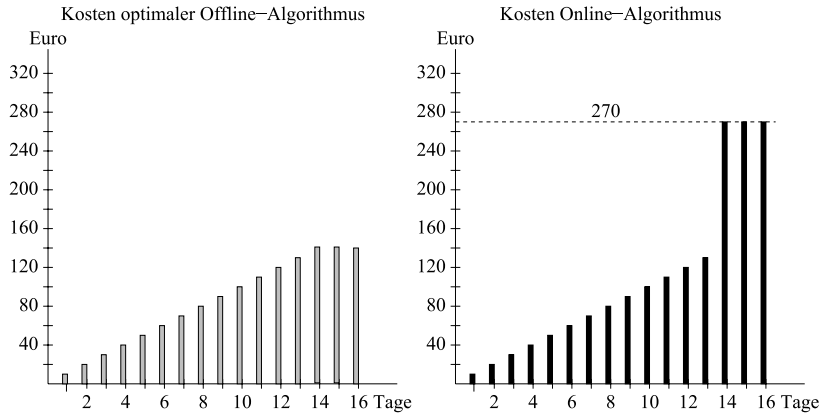



**Abb. 39.2.** Illustration des kompetitiven Faktors von 2 – konkretes Beispiel

dass das Leihen von Skiern 1 Euro und das Kaufen $n$ Euro kostet. Unsere Online-Strategie kauft am $n$-ten Tag, wenn die gesamten Leihkosten gerade $n-1$ betragen. Die Abb. 39.3 veranschaulicht, dass bei $x < n$ Tagen die Online-Strategie optimale Kosten erzeugt. Bei $x \geq n$ sind die erzeugten Kosten weniger als doppelt so hoch wie das Optimum. Wir sind also 2-kompetitiv.

Das ist ja schon ganz nett, aber eigentlich würden wir gern näher am Optimum sein. Leider ist das nicht möglich. Kaufen wir zu einem anderen Zeitpunkt, so gibt es immer mindestens einen Fall, in dem wir mindestens doppelt so viel wie das Optimum zahlen müssten. Kauft man früher, z. B. bereits am 11. Tag, so bezahlt man 240 Euro statt der notwendigen 110 Euro. Kauft man später, beispielsweise erst am 17. Tag, so bezahlt man 300 Euro

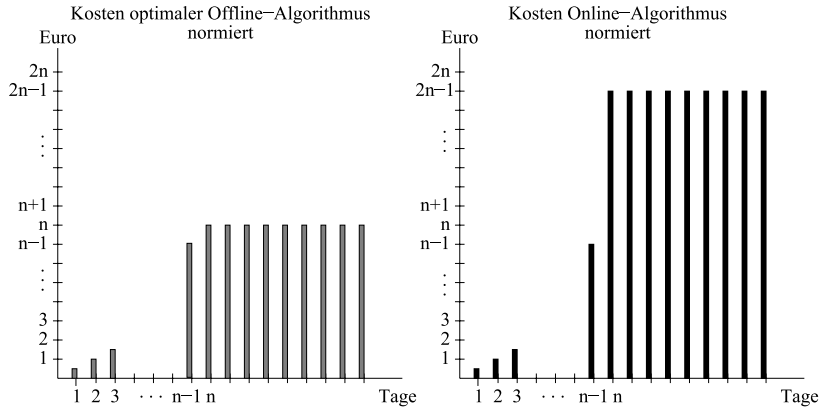



**Abb. 39.3.** Illustration des kompetitiven Faktors von 2 – allgemeiner Fall

statt 140 Euro. In beiden Fällen liegt man damit über dem Doppelten der notwendigen Kosten. Kein Online-Algorithmus ist also besser als 2-kompetitiv.

## Das Seitenwechselproblem

Ein wichtiges Online-Problem der Informatik, das intern in einem Computer ständig auftritt, ist das *Seitenwechselproblem*. Hier muss stets entschieden werden, welche Speicherseiten im Hauptspeicher eines Computers und welche auf Festplatte gehalten werden, vgl. Abb. 39.4. Der Prozessor eines Rechners kann auf den Hauptspeicher sehr schnell zugreifen, dieser verfügt jedoch über eine relativ kleine Speicherkapazität. Viel mehr Platz ist auf der Festplatte vorhanden. Dort benötigen Zugriffe jedoch deutlich mehr Zeit. Die Zugriffszeiten stehen dabei in einem Verhältnis von ca. $1 : 10^6$. Würde ein Hauptspeicherzugriff 1 Sekunde dauern, so müsste man auf die gleichen Daten von der Festplatte etwa 11,5 Tage warten. Aus diesem Grund benötigen wir einen Algorithmus, der Seiten derart in den Hauptspeicher lädt und aus ihm entfernt, dass wir möglichst selten auf die Festplatte zugreifen müssen. In dem folgenden Beispiel sind gerade die Speicherseiten $A$, $B$, $C$, $D$, $E$ und $G$ im Hauptspeicher. Der Prozessor erzeugt zunächst Datenanfragen auf die Seiten $D$, $B$, $A$, $C$, $D$, $E$, $G$ und ist in der glücklichen Lage, alle Seiten im schnellen Hauptspeicher vorzufinden.

Bei der nächsten Anfrage haben wir jedoch Pech. Die benötigte Seite $F$ liegt auf der Festplatte! Dieses Ereignis nennt man einen *Seitenfehler*. Jetzt

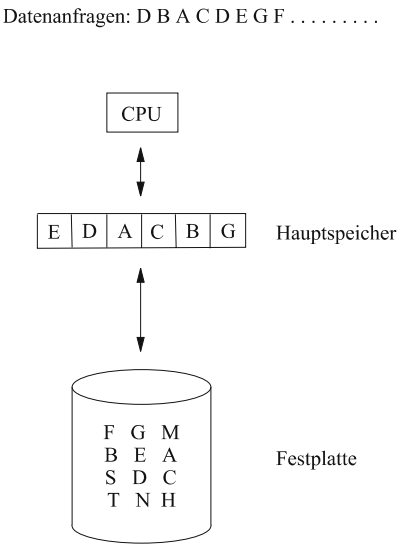



**Abb. 39.4.** Die Speicherhierarchie beim Seitenwechselproblem

muss die fehlende Seite von der Platte in den Hauptspeicher geladen werden. Leider ist dieser voll, und wir sind gezwungen, eine Seite zu entfernen, um die fehlende Seite in den Hauptspeicher zu bringen und den Datenzugriff durchzuführen. Aber welche Seite soll entfernt werden? Hier entsteht ein Online-Problem: Bei einem Seitenfehler muss ein Seitenwechsel-Algorithmus entscheiden, welche Seite aus dem Hauptspeicher auszulagern ist, ohne zukünftige Datenanfragen zu kennen. Wüsste man, welche Seite im Hauptspeicher lange nicht benötigt wird, so könnte man sich von dieser trennen. Dies ist aber nicht bekannt. Die folgende Online-Strategie hat sich in der Praxis bewährt:

> **Online-Strategie Least-Recently-Used (LRU):** Tritt bei vollem Hauptspeicher ein Seitenfehler auf, so entferne die Seite, die am längsten in der Vergangenheit nicht benötigt wurde. Dann lade die fehlende Seite.

In unserem Beispiel würde LRU die Seite $B$ entfernen, da auf diese die längste Zeit nicht zugegriffen wurde. Die Intuition von LRU ist, dass lange nicht benötigte Seiten für den Prozessor offenbar nicht mehr so interessant sind und somit hoffentlich auch in der nahen Zukunft nicht gebraucht werden. Man kann zeigen, dass LRU $k$-kompetitiv ist, wobei $k$ die Anzahl der Seiten ist, die gleichzeitig im Hauptspeicher gehalten werden kann. Dies ist durchaus eine hohe Kompetitivität, denn in der Praxis ist $k$ groß. Andererseits zeigt das Ergebnis, dass LRU eine beweisbare Güte besitzt. Dies trifft auf viele andere Seitenwechsel-Strategien nicht zu.

Neben LRU gibt es noch andere Strategien, die versuchen das Seitenwechselproblem möglichst gut zu lösen:

> **Online-Strategie First-In First-Out (FIFO):** Tritt bei vollem Hauptspeicher ein Seitenfehler auf, so entferne die Seite, die zuerst in den Hauptspeicher geladen wurde. Dann lade die fehlende Seite.

> **Online-Strategie Most-Recently-Used (MRU):** Tritt bei vollem Hauptspeicher ein Seitenfehler auf, so entferne die Seite, die zuletzt angefragt wurde. Dann lade die fehlende Seite.

Man kann beweisen, dass FIFO ebenfalls $k$-kompetitiv ist. Jedoch arbeitet FIFO in der Praxis bei weitem nicht so gut wie LRU. Für MRU kann man eine Sequenz von Anfragen konstruieren, die bei vollem Hauptspeicher in jedem Schritt einen Seitenfehler erzeugt. Dazu fragt man bei vollem Hauptspeicher zwei Seiten $A$ und $B$, die nicht im Hauptspeicher sind, immer wieder wechselseitig an. Bei der ersten Anfrage an $A$ wird die Seite $A$ in den Hauptspeicher geladen, nachdem gemäß MRU eine Seite entfernt wurde. Wird nun $B$ angefragt, so wird $A$ aus dem Hauptspeicher entfernt, da diese Seite zuletzt angefragt wurde. Jetzt wird wieder die Seite $A$ angefragt. Da wir $A$ gerade entfernt haben, müssen wir sie erneut von der Festplatte in den Hauptspeicher laden und $B$ gemäß MRU wieder entfernen. Dann wird wieder $B$ angefragt usw. Wie wir sehen, erzeugt unsere Online-Strategie MRU in jedem Schritt

einen Seitenfehler. Der optimale Algorithmus lädt die beiden Seiten $A$ und $B$ bei der jeweils ersten Anfrage in den Hauptspeicher und behält sie dann dort. Er erzeugt also nur zwei Seitenfehler. Unsere Online-Strategie MRU ist für diese Anfragesequenz also sehr schlecht. Dass zwei oder wenige Seiten zyklisch angefragt werden, ist in der Praxis häufig anzutreffen, da in Programmen oft Schleifen auftreten, die auf wenige Speicherseiten zugreifen. Die Strategie MRU sollte daher in der Praxis nicht angewandt werden.

## Zum Weiterlesen

In der Informatik treten in sehr vielen Bereichen Online-Probleme auf. Beispiele sind die Datenstrukturierung, das Prozessorscheduling oder die Robotik, um nur einige Gebiete zu nennen.

1. Kapitel 40 (Bin Packing)

   Hier wird das Bin-Packing-Problem studiert, das ebenfalls ein klassisches Online-Problem darstellt.

2. `http://de.wikipedia.org/wiki/Paging`

   Ein leicht zugänglicher Artikel zum Paging (Seitenwechselproblem).

3. D.D. Sleator and R.E. Tarjan: *Amortized efficiency of list update and paging rules*. In: *Communication of the ACM*, 28:202–208, 1985.

   Ein grundlegender Fachartikel über Online-Algorithmen, der sich mit dem Seitenwechselproblem und auch mit Online-Algorithmen für Datenstrukturen befasst. Der Artikel wird als Ursprungsreferenz der kompetitiven Analyse betrachtet.

4. S. Irani and A.R. Karlin: *Online computation*. In *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, 521–564, 1995.

   Ein Übersichtsichtsartikel zu Online-Algorithmen, der auch das Ski- und das Seitenwechselproblem behandelt,

5. A. Borodin and Ran El-Yaniv: *Online Computation and Competitive Analysis*. Cambridge Unviversity Press, 1998.

   Ein umfangreiches Lehrbuch über Online-Algorithmen.

# 40

# Bin Packing oder „Wie bekomme ich die Klamotten in die Kisten?“

Joachim Gehweiler und Friedhelm Meyer auf der Heide

Universität Paderborn

Ich habe diesen Sommer mein Abi gemacht und möchte zum Herbst mit dem Studium beginnen – Informatik natürlich! Da es in meinem kleinen Ort keine Uni gibt, werde ich deshalb in Kürze umziehen müssen. Dann heißt es, all die tausend Sachen in meinen Schränken und Regalen in Umzugskartons zu verpacken. Um den Umzug möglichst kostengünstig zu gestalten, werde ich mich dabei bemühen, die Gegenstände in möglichst wenige Kartons zu verpacken.


Wenn ich nun alle Gegenstände einfach so der Reihe nach aus den Regalen nehme und in einen Umzugskarton nach dem anderen verpacke, verschwende ich dabei eine ganze Menge Platz in den Kartons, da die Gegenstände verschieden groß und unterschiedlich geformt sind und sich dadurch eine Menge Lücken in den Umzugskartons ergeben.

Die optimale Lösung für dieses Problem, d.h. die kleinstmögliche Anzahl benötigter Umzugskartons, würde ich sicherlich finden, wenn ich alle Möglichkeiten, die Gegenstände in Umzugskartons zu verpacken, der Reihe nach ausprobiere. Doch das würde bei so vielen Gegenständen ein halbe Ewigkeit dauern und obendrein ein riesiges Chaos in der Wohnung verursachen.

Deshalb würde ich die Gegenstände schon am liebsten in der Reihenfolge in Umzugskartons verpacken, in der sie mir beim Ausräumen der Schränke und Regale zufällig gerade in die Hände fallen. Die entscheidende Frage ist daher, wie viele Umzugskartons ich bei dieser Vorgehensweise mehr benötige

als bei der optimalen Lösung. Um dies herauszufinden, werde ich das Problem nun analysieren.

## Das Online-Problem „billig umziehen“

Da ich die Gegenstände der Reihe nach aus den Schränken bzw. Regalen nehmen und verpacken möchte, habe ich es mit einem Online-Problem (vergleiche Einführung in Online Algorithmen, Kap. 39) zu tun:

- Die relevanten Daten (hier: die Größe der einzelnen Gegenstände) treffen erst nach und nach im Laufe der Zeit ein. Die Liste dieser Größen, in der Reihenfolge ihres Auftretens, bezeichnen wir im Folgenden mit $G$.
- Es liegen keine Informationen über die zukünftigen Daten (hier: die Größen der noch nicht betrachteten Gegenstände) vor.
- Die Anzahl der zu bearbeitenden Daten (hier: der zu verpackenden Gegenstände) ist nicht im Voraus bekannt.
- Die aktuelle Anfrage muss sofort bearbeitet werden (hier: Gegenstände werden nicht vorübergehend zur Seite gelegt).

In der Realität liegen für einige Aspekte zwar Schätzwerte vor, da ich in der Wohnung schließlich jahrelang gelebt habe und somit eine gewisse Vorstellung davon habe, was sich alles in den Regalen und Schränken befindet; dies werde ich hier jedoch außer Acht lassen und das Problem somit idealisiert betrachten.

In Fachkreisen wird dieses Problem allgemein als das *Bin Packing* Problem bezeichnet (englisch bin = Kiste, Behälter; (to) pack = packen, verpacken).

Meine Verpack-Strategie sieht nun als Online-Algorithmus folgendermaßen aus: Die Eingabe besteht aus einer Daten-Sequenz $G = (G_1, G_2, \ldots)$ von Größen $G_i$ der zu verpackenden Gegenstände. Die Umzugskartons werden mit $K_j$ bezeichnet, und die Ausgabe besteht aus der Anzahl $n$ der benötigten Umzugskartons.

Algorithmus NextFit

```
1  Setze n := 1.
2  Für jedes G_i aus G tue Folgendes:
3      Falls G_i in Karton K_n keinen Platz mehr hat,
4          schließe Karton K_n und
5          setze n := n + 1.
6      Packe G_i in Karton K_n.
```

Mit etwas mehr Aufwand kann ich auch folgendermaßen vorgehen: Ich könnte die Umzugskartons erst am Ende verschließen und beim Verpacken eines jeden Gegenstands alle angefangenen Umzugskartons darauf überprüfen, ob vielleicht noch genügend Platz übrig ist. Dies hätte dann einen Vorteil, wenn ich für einen besonders großen Gegenstand einen neuen Umzugskarton

nehmen muss und danach wieder eine Reihe kleinerer Gegenstände zu verpacken sind, denn diese kleineren Gegenstände können eventuell noch in einem früheren Umzugskarton Platz finden.
Als Online-Algorithmus sieht diese zweite Strategie folgendermaßen aus:

Algorithmus FirstFit

```
1  Setze n := 1.
2  Für jedes G_i aus G tue Folgendes:
3      Für j := 1, ..., n tue Folgendes:
4          Falls G_i in Karton K_j noch Platz hat,
5              packe G_i in Karton K_j und
6              fahre mit dem nächsten Gegenstand fort (gehe zu Schritt 2).
7      Setze n := n + 1 und
8      packe G_i in Karton K_n.
```

## Analyse der Algorithmen

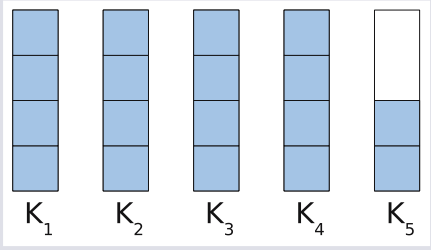
Um die Analyse, wie gut oder schlecht meine Strategien funktionieren, etwas zu vereinfachen, treffe ich folgende Annahme: Ein Gegenstand passt genau dann in einen Umzugskarton, wenn das noch unbenutzte Volumen des Umzugskartons größer oder gleich dem Volumen des zu verpackenden Gegenstands ist (ich vernachlässige also den aufgrund von „Verschnitt“ nicht nutzbaren Platz in den Umzugskartons). Weiterhin wähle ich die Volumeneinheit derart, dass die Kapazität der Umzugskartons genau 1 ist (und die Größen der zu verpackenden Gegenstände somit kleiner oder gleich 1 sind).

Um ein Gefühl für die Güte des Ergebnisses meiner Online-Algorithmen zu bekommen, schaue ich mir ein paar Beispiele an. Sind alle Gegenstände gleich groß – wie in Beispiel 1 – so liefern NextFit und FirstFit beide das optimale Ergebnis:

Beispiel 1

$G = \left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right)$

NextFit = FirstFit: $n = 5$

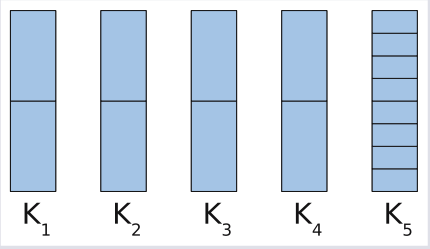


In Beispiel 2 liefern NextFit und FirstFit ebenfalls noch beide das optimale Ergebnis:

**Beispiel 2**

$G = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}\right)$

NextFit = FirstFit: $n = 5$

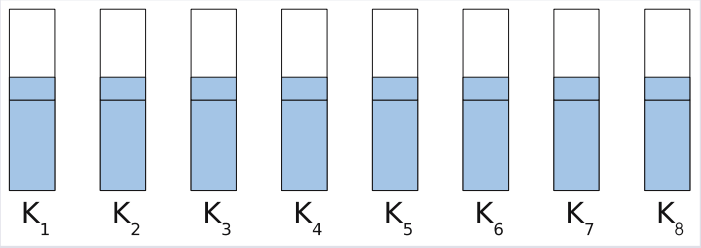



Beispiel 3 jedoch zeigt, dass die Reihenfolge der Gegenstände das Ergebnis beeinflussen kann – hier schneidet NextFit deutlich schlechter ab:

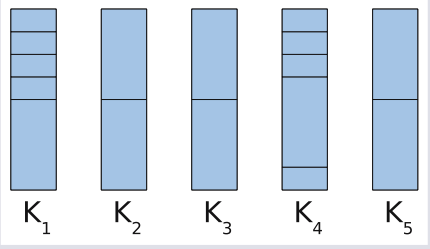
**Beispiel 3**

$G = \left(\frac{1}{2}, \frac{1}{8}, \frac{1}{2}, \frac{1}{8}, \frac{1}{2}, \frac{1}{8}, \frac{1}{2}, \frac{1}{8}, \frac{1}{2}, \frac{1}{8}, \frac{1}{2}, \frac{1}{8}, \frac{1}{2}, \frac{1}{8}, \frac{1}{2}, \frac{1}{8}\right)$

NextFit: $n = 8$




FirstFit: $n = 5$



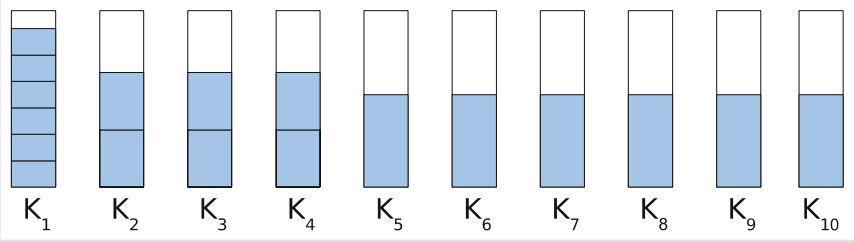

Wählt man in Beispiel 3 anstatt der Größe $\frac{1}{8}$ eine viel kleinere Zahl, so steigt der ungenutzte Platz pro Umzugskarton in NextFit sogar auf fast die

Hälfte. Die Strategie NextFit verbraucht im Extremfall also fast doppelt so viele Umzugskartons wie bei optimaler Packung nötig wären.
Die Strategie FirstFit hat in Beispiel 3 zwar noch optimal abgeschnitten, doch auch für FirstFit gibt es ungünstige Eingaben, wie Beispiel 4 zeigt:

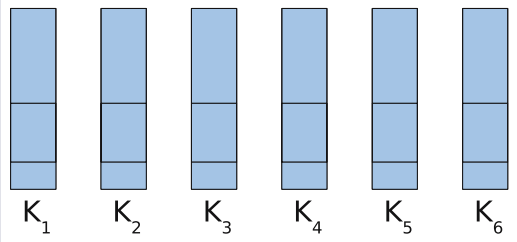
**Beispiel 4**

$G = (0.15, 0.15, 0.15, 0.15, 0.15, 0.15,$
$0.34, 0.34, 0.34, 0.34, 0.34, 0.34,$
$0.51, 0.51, 0.51, 0.51, 0.51, 0.51)$

FirstFit: $n = 10$




Optimal: $n = 6$




Hier ergibt sich also ein Verhältnis von 10:6 für FirstFit gegenüber der optimalen Packung, d.h., FirstFit benötigt 1,67-mal so viele Umzugskartons.

Es stellt sich nun die Frage, ob die Negativbeispiele 3 und 4 bereits die schlechtestmöglichen Eingabefolgen für NextFit bzw. FirstFit darstellen, oder ob es noch schlimmer kommen kann.

In Kap. 39 haben wir gelernt, dass man eine Online-Strategie $\alpha$-kompetitiv nennt, wenn man für keine einzige Eingabe mehr als die $\alpha$-fachen Kosten erzeugt im Vergleich zur optimalen Lösung, d.h. den Kosten, die man verursacht hätte, wenn man die Zukunft gekannt hätte. Um nun den kompetitiven Faktor $\alpha$ von NextFit zu berechnen, bezeichnen wir mit $k$ die Anzahl der Gegenstände und mit $n$ die Anzahl der von NextFit benötigten Umzugskartons. Weiterhin beschreiben wir mit $v(G_i)$ das Volumen des Gegenstands $G_i$ und mit $v(K_j)$ das belegte Volumen im Umzugskarton $K_j$. Betrachtet man nun zwei nacheinander befüllte Umzugskartons $K_j$ und $K_{j+1}$, $1 \leq j < n$, so gilt:

$$v(K_j) + v(K_{j+1}) > 1$$

Wäre diese Bedingung nicht erfüllt, hätten sämtliche Gegenstände in $K_{j+1}$ noch in $K_j$ Platz gehabt und wären somit nach Definition von NextFit nicht in $K_{j+1}$ gelegt worden. Addiert man nun das belegte Volumen von $K_1$ und $K_2$, von $K_3$ und $K_4$, von $K_5$ und $K_6$ usw., so ist dies jeweils echt größer als 1 und man erhält somit für die Summe über alle Umzugskartons:

$$\sum_{j=1}^{n} v(K_j) > \left\lfloor \frac{n}{2} \right\rfloor$$

Mittels der Gaußklammer $\lfloor\ \rfloor$ runden wir den Bruch auf der rechten Seite der Ungleichung ab für den Fall, dass $n$ ungerade ist. Wir erhalten somit eine untere Schranke für die Anzahl der benötigten Kartons, d.h. einen Wert, der selbst bei der optimalen Packung nicht unterboten werden kann. Nun gilt, dass das Gesamtvolumen aller Gegenstände gleich dem insgesamt in allen Umzugskartons belegten Volumen ist, d.h.

$$\sum_{i=1}^{k} v(G_i) = \sum_{j=1}^{n} v(K_j)$$

Also benötigt aufgrund des Gesamtvolumens der Gegenstände selbst die bestmögliche Packung mindestens

$$\left\lceil \sum_{i=1}^{k} v(G_i) \right\rceil \geq \left\lceil \frac{n}{2} \right\rceil$$

Umzugskartons. Das Gesamtvolumen der Gegenstände muss hierbei aufgerundet werden, da die Anzahl der Umzugskartons ganzzahlig sein muss. Für das Verhältnis der von NextFit benötigten Umzugskartons zur Anzahl der bei der optimalen Packung benötigten Umzugskartons, $\frac{\text{Lösung NextFit}}{\text{optimale Lösung}}$, folgt somit:

$$n / \left\lceil \frac{n}{2} \right\rceil \leq 2$$

Der Online-Algorithmus NextFit ist also 2-kompetitiv (vergleiche Einführung in Online Algorithmen, Kap. 39). Da dieser Beweis auf FirstFit übertragbar ist, ist somit auch FirstFit 2-kompetitiv. Mit einem deutlich komplizierteren Beweis (den wir hier nicht vorführen, vergleiche weiterführende Materialien) kann man sogar zeigen, dass FirstFit 1,7-kompetitiv ist.

## Wie gut kann ein Online-Algorithmus für Bin Packing sein?

Wir kennen nun eine Schranke für die Approximationsgüte von NextFit bzw. FirstFit und wissen, dass es Eingabefolgen gibt, die diese Schranke beinahe erreichen, d.h. für die das Ergebnis fast einen Faktor 2 bzw. 1,7 vom

theoretischen Optimum entfernt ist. Auf der einen Seite ist dies ein gutes Ergebnis, da wir wissen, dass der verschwendete Platz in den Umzugskartons niemals einen bestimmten Faktor überschreitet, andererseits ist das Ergebnis aber auch etwas unbefriedigend, denn es macht schon einen schmerzlichen Unterschied, ob der Umzug nun 1000 EUR oder 2000 EUR (bzw. 1700 EUR) kostet.

Um nun abschließend zu beurteilen, wie gut oder schlecht die Verpackungsstrategien tatsächlich sind, gilt es noch zu untersuchen, wie gut ein Online-Algorithmus für dieses Problem überhaupt sein kann. Denn da die Eingabefolge nicht im Voraus bekannt ist, dürfte es wohl unmöglich sein, einen Online-Algorithmus zu entwerfen, der immer ein optimales Ergebnis liefert. Dies werden wir nun beweisen.

Angenommen, wir haben eine Eingabefolge, die $2 \cdot x$ Gegenstände der Größe $\frac{1}{2} - \varepsilon$ enthält, wobei $x$ eine positive, ganze Zahl und $\varepsilon$ eine beliebig kleine, positive Zahl ist. Die optimale Packung für diese Eingabefolge sind offensichtlich $x$ Umzugskartons, die jeweils mit 2 Gegenständen gefüllt sind. Wir betrachten nun einen beliebigen Online-Algorithmus und bezeichnen diesen mit BinPac. BinPac wird die $2 \cdot x$ Gegenstände der Eingabefolge – je nach Strategie – so auf die Umzugskartons verteilen, dass in jedem Umzugskarton entweder ein oder zwei Gegenstände landen. Mit $b_1$ bezeichnen wir die Anzahl an Umzugskartons, in denen ein Gegenstand liegt, und mit $b_2$ die Anzahl der Umzugskartons, in denen zwei Gegenstände liegen. Mit $b = b_1 + b_2$ bezeichnen wir die Anzahl der insgesamt von BinPac benötigten Umzugskartons. Man stelle nun fest, dass folgender Zusammenhang gilt:

$$b_1 + 2 \cdot b_2 = 2 \cdot x$$

$$\Rightarrow b_1 = 2 \cdot x - 2 \cdot b_2$$

Setzt man dies in $b = b_1 + b_2$ ein, so erhält man:

$$b = (2 \cdot x - 2 \cdot b_2) + b_2 = 2 \cdot x - b_2 \tag{40.1}$$

Auf dieses Zwischenergebnis werden wir später wieder zurückkommen. Nun überlegen wir uns, was passiert, wenn unsere Eingabefolge $4 \cdot x$ Gegenstände enthält, wobei die ersten $2 \cdot x$ Gegenstände wieder die Größe $\frac{1}{2} - \varepsilon$ haben und die restlichen $2 \cdot x$ Gegenstände die Größe $\frac{1}{2} + \varepsilon$. Da Online-Algorithmen nicht in die Zukunft schauen können, wird sich BinPac für die ersten $2 \cdot x$ kleineren Gegenstände genauso verhalten wie vorhin, als keine weiteren Gegenstände folgten. Die Gegenstände der Größe $\frac{1}{2} + \varepsilon$ können nun zunächst auf die $b_1$ Umzugskartons, in denen noch Platz ist, verteilt werden, und für die restlichen $2 \cdot x - b_1$ Gegenstände muss jeweils ein neuer Umzugskartons angefangen werden. Insgesamt benötigt BinPac bei dieser Eingabefolge also mindestens

$$b + (2 \cdot x - b_1) = (b_1 + b_2) + (2 \cdot x - b_1) = 2 \cdot x + b_2 \tag{40.2}$$

Umzugskartons. Die optimale Lösung wäre aber gewesen, in jeden Umzugskarton jeweils einen der kleineren und einen der größeren Gegenstände zu legen, so dass man insgesamt nur $2 \cdot x$ Umzugskartons benötigt hätte.

Mit diesen Vorüberlegungen können wir nun beweisen, dass kein Online-Algorithmus besser als $\frac{4}{3}$-kompetitiv sein kann. Wir führen diesen Beweis, indem wir zunächst annehmen, dass es sehr wohl einen besseren Online-Algorithmus gäbe, und führen diese Annahme dann zum Widerspruch.

Angenommen, BinPac wäre besser als $\frac{4}{3}$-kompetitiv. Dann müsste die Anzahl der benötigten Umzugskartons bei der ersten betrachteten Eingabefolge echt kleiner sein als $\frac{4}{3}$ der bei einer optimalen Lösung benötigten Anzahl von Umzugskartons. Formal bedeutet das:

$$b < \frac{4}{3} \cdot x$$

Wendet man diese Bedingung auf Gleichung (40.1) an, so erhält man:

$$2 \cdot x - b_2 < \frac{4}{3} \cdot x$$

$$\Rightarrow b_2 > \frac{2}{3} \cdot x \tag{40.3}$$

Analog müsste für die zweite Eingabefolge gelten, dass die Anzahl der benötigten Umzugskartons (vergleiche Gleichung (40.2)) echt kleiner ist als $\frac{4}{3}$ der optimalen Lösung $(2 \cdot x)$. Formal bedeutet das:

$$2 \cdot x + b_2 < \frac{4}{3}(2 \cdot x)$$

$$\Rightarrow b_2 < \frac{2}{3} \cdot x \tag{40.4}$$

Damit haben wir einen Widerspruch, denn laut Gleichungen (40.3) und (40.4) müsste $b_2$ sowohl echt kleiner als auch echt größer als $\frac{2}{3} \cdot x$ sein, was unmöglich ist. Folglich muss die Annahme falsch gewesen sein, und wir haben bewiesen:

**Satz 1**

Es gibt keinen $\alpha$-kompetitiven Online-Algorithmus für das Bin Packing Problem mit $\alpha < \frac{4}{3}$.

Unter dem Aspekt, dass selbst die bestmögliche Online-Strategie für das Umzugsproblem nicht besser als $\frac{4}{3}$-kompetitiv sein kann, erscheint nun die 1,7-kompetitive Strategie FirstFit in einem ganz anderen Licht. Na, dann kann's mit dem Packen ja losgehen!

Eine weitere Einsatzmöglichkeit des Bin Packing ist beispielsweise, Dateien auf CDs zu verteilen (etwa im Rahmen einer Datensicherung). In diesem Fall lassen sich die oben beschriebenen Strategien sogar direkt anwenden, d. h., man muss keine vereinfachende Annahmen bezüglich des „Verschnitts" treffen, da dieses Problem bei Datenströmen nicht auftritt.

## Zum Weiterlesen

1. `http://www-cg-hci-f.informatik.uni-oldenburg.de/~da/iva/baer/start/bin1.html`

   Hier wird die Funktionsweise des FirstFit-Algorithmus anhand eines Java-Applets interaktiv veranschaulicht.

2. D. Johnson: *Fast algorithms for bin packing*. Journal of Computer and System Sciences 8 (1974), S. 272–314.

   Dies ist die erste Veröffentlichung zum Thema Online Bin Packing.

3. S. Seiden: *On the online bin packing problem*. In: *Proceedings of the 28th International Colloquium on Automata, Languages and Programming* (Jul 2001) S. 237–249.

   In diesem Artikel wird der bislang beste bekannte Algorithmus für Online Bin Packing (Harmonic++) vorgestellt, der 1,58889-kompetitiv ist.

4. A. van Vliet: *An improved lower bound for online bin packing algorithms*. Information Processing Letters 43, 5, (Oct 1992), S. 277–284.

   In diesem Artikel wird der Wert für $\alpha$ aus Satz 1 von $\frac{4}{3}$ auf 1,5401 verbessert.

# 41

# Das Rucksackproblem

Rene Beier und Berthold Vöcking

Max-Planck-Institut für Informatik, Saarbrücken
RWTH Aachen

In zwei Monaten startet die nächste Rakete zur Raumstation. Die Weltraumagentur ist etwas knapp bei Kasse und bietet deshalb kommerziellen Forschungsinstituten an, wissenschaftliche Experimente an der Raumstation durchführen zu lassen. Die Rakete kann neben den obligatorischen Verpflegungsrationen noch maximal 645 kg zusätzliche Last mitnehmen, die für die Experimente benutzt werden soll. Die Weltraumagentur erhält von den Forschungsinstituten verschiedene Angebote, in denen steht, wie viel Geld sie für den Transport und die Durchführung des Experiments zu zahlen bereit sind und wie schwer die Geräte für ihr Experiment sind. Welche Experimente soll die Weltraumagentur auswählen, wenn sie den Profit maximieren will?

Hinter diesem Szenario verbirgt sich ein klassisches Problem der Optimierung, das so genannte Rucksackproblem: Wir stellen uns vor, dass wir einen Rucksack haben, der höchstens ein bestimmtes vorgegebenes Gewicht $T$ trägt. $T$ wird als *Gewichtsschranke* bezeichnet. Neben dem Rucksack gibt es eine Menge von $n$ Objekten (Gegenständen), die jeweils ein Gewicht und einen Profit haben. Wir suchen eine Teilmenge der Objekte, die wir in den Rucksack packen können, ohne die Gewichtsschranke $T$ zu verletzen, so dass der Gesamtprofit möglichst groß ist, also die Summe der Profite der eingepackten Objekte maximiert wird.

Im obigen Beispiel steht der Rucksack symbolisch für die Rakete und er hat eine Gewichtsschranke von $T = 645$. Die Objekte repräsentieren die Experimente. Wir konkretisieren dieses Beispiel und stellen uns vor, dass $n = 8$ Objekte (Experimente) mit den folgenden Gewichten und Profiten vorliegen.

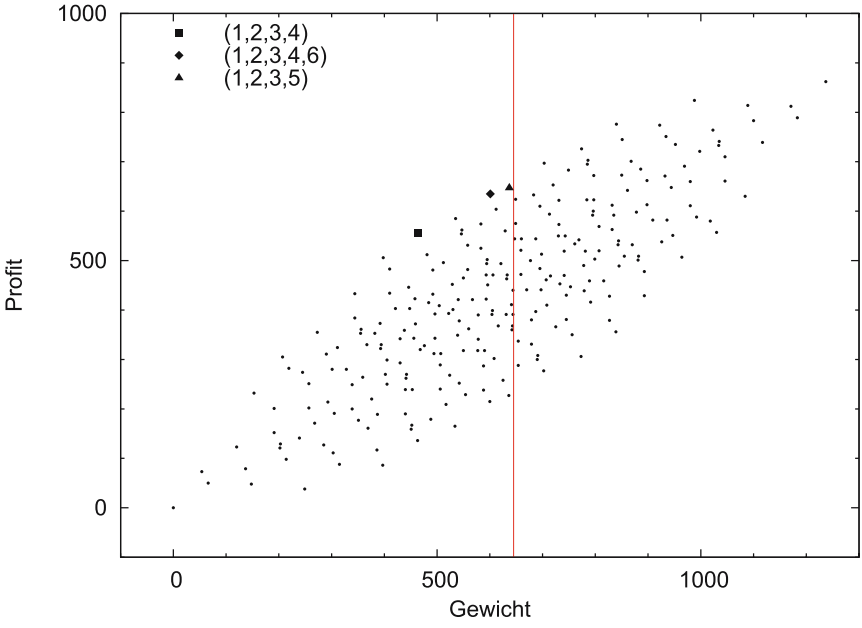
| Objekt-Nr. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Gewicht in kg | 153 | 54 | 191 | 66 | 239 | 137 | 148 | 249 |
| Profit in 1000 Euro | 232 | 73 | 201 | 50 | 141 | 79 | 48 | 38 |
| Profitdichte | 1.52 | 1.35 | 1.05 | 0.76 | 0.59 | 0.58 | 0.32 | 0.15 |

Intuitiv erscheint es sinnvoll, diejenigen Objekte zuerst auszuwählen, die den größten Profit pro Gewichtseinheit erzielen. Dieses Verhältnis zwischen

Profit und Gewicht eines Objektes bezeichnet man auch als *Profitdichte*. Wir haben in der Tabelle die Profitdichten berechnet und die Objekte bereits so angeordnet, dass die Profitdichte von links nach rechts immer kleiner wird.

In unserem ersten Algorithmus werden zuerst die Objekte sortiert, so dass die Profitdichte immer kleiner wird. Dazu können wir beispielsweise die in Kap. 2 und 3 vorgestellten Sortieralgorithmen benutzen. Nun startet man mit dem leeren Rucksack und fügt die Objekte nacheinander in der berechneten Reihenfolge ein. Wir stoppen, sobald das nächste Objekt nicht mehr in den Rucksack passt. In unserem Beispiel würden wir also nacheinander die Objekte $1, 2, 3$ und $4$ einpacken. Diese haben zusammen ein zulässiges Gewicht von 464. Nummer 5 können wir nicht mehr einpacken, da dann das Gesamtgewicht mit $464 + 239 = 703$ die Kapazität des Rucksacks von 645 überschreitet. Die vier Objekte im Rucksack erzielen zusammen einen Profit von 556. Ist das die beste Möglichkeit, den Rucksack zu packen? – Offensichtlich nicht, denn wir könnten zusätzlich noch Nummer 6 einpacken. Dann enthält unser Rucksack ein Gewicht von 601 und erzielt einen Profit von 635. Ist dies nun die beste Lösung? – Leider nein!

Um garantiert die beste Lösung zu finden, kann man einfach alle möglichen Kombinationen, den Rucksack zu bepacken, ausprobieren. Leider gibt es sehr viele Kombinationsmöglichkeiten. Wir können all diese Kombinationen graphisch in einem Gewicht-Profit-Diagramm darstellen, in dem wir für jede Teilmenge entsprechend ihres Gewichts und ihres Profits einen Punkt zeichnen, z. B. für die Teilmenge $\{1, 2, 3, 4, 6\}$ den Punkt $(601, 647)$.



Jeder Punkt repräsentiert eine Teilmenge der Objekte. Wie viele Punkte sind im Bild zu sehen? Für jedes Objekt ist zu entscheiden, ob es in den Rucksack gepackt wird oder nicht. Pro Objekt sind das zwei Möglichkeiten. Bei $n$ Objekten ergeben sich insgesamt $2^n$ Kombinationsmöglichkeiten. In unserem Beispiel müssten wir also $2^8 = 256$ Möglichkeiten überprüfen. Allerdings sind diejenigen Teilmengen mit Gewicht größer als 645 zu schwer für den Rucksack. Das sind gerade die Punkte, die rechts von der vertikalen Linie liegen. Punkte, die links oder auf der Linie liegen, sind zulässig. Von all den zulässigen Punkten möchten wir denjenigen mit dem größten Profit auswählen. Dieser Punkt entspricht der Teilmenge mit den Objekten $1, 2, 3$ und $5$, die zusammen Gewicht 637 und Profit 647 haben. Diese ist die so genannte *optimale Lösung*.

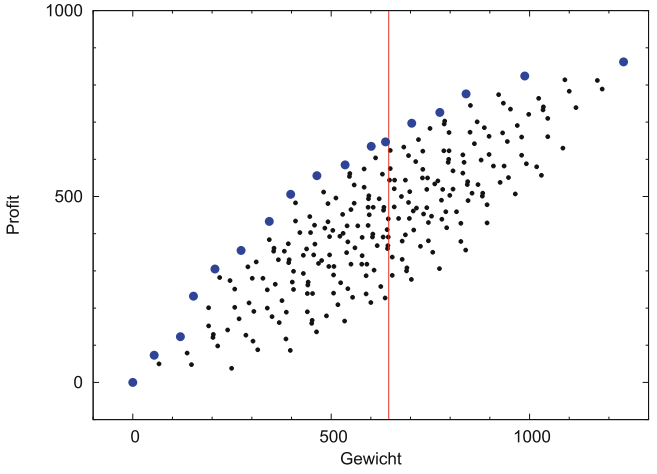
Das Ausprobieren aller möglichen Teilmengen hat allerdings einen großen Nachteil. Steigt die Anzahl der Objekte nur leicht an, so „explodiert“ die Anzahl der Teilmengen, die man ausprobieren muss. Erhält die Weltraumagentur in unserem Beispiel 60 Angebote für Experimente, so gibt es bereits

$$2^{60} = 1.152.921.504.606.846.976,$$

also mehr als eine Trillion verschiedene Möglichkeiten, eine Auswahl zu treffen. Wenn man optimistisch annimmt, dass ein Computer eine Milliarde Teilmengen pro Sekunde testen kann, so benötigt er trotzdem noch über 36 Jahre, um alle Teilmengen durchzuprobieren. So lange wollen wir den Start der Rakete aber nicht verzögern.

### Pareto-optimale Lösungen

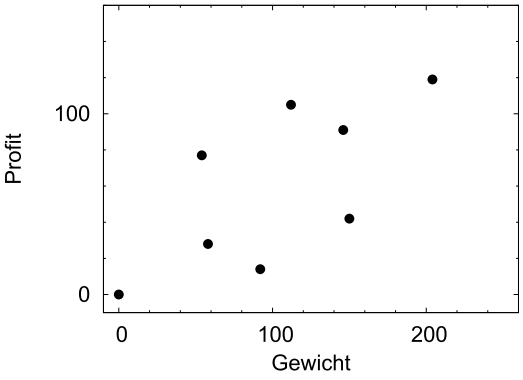
Wie kann man die optimale Lösung schneller finden? Die Grundidee für einen effizienteren Algorithmus basiert auf folgender Beobachtung: Eine Teilmenge von Objekten kann keine optimale Lösung sein, falls es eine andere Teilmenge gibt, die gleichzeitig weniger Gewicht und mehr Profit aufweist. Schauen wir noch einmal auf unser Beispiel:



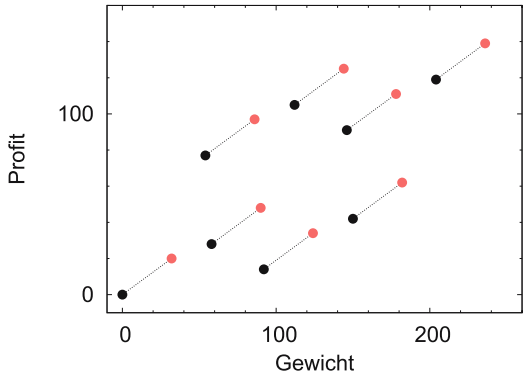
Keiner der schwarzen Punkte kann optimal sein, da wir für jeden schwarzen Punkt einen noch besseren Punkt finden können, nämlich einen solchen, der mit weniger Gewicht mehr Profit erreicht, also weiter oben und weiter links im Diagramm ist. Wir sagen, dass dieser bessere Punkt den schwarzen Punkt *dominiert*. Ein blauer Punkt im Diagramm hat hingegen die Eigenschaft, dass er von keinem anderen Punkt dominiert wird. Derartige Lösungen heißen *Pareto-optimal*. Eine Teilmenge ist somit Pareto-optimal, wenn es keine andere Teilmenge gibt, die mit weniger Gewicht einen höheren Profit erreicht. In unserem Beispiel sind nur 17 der 256 Punkte Pareto-optimal und somit haben wir nur 17 Kandidaten für die optimale Lösung. Diese Liste von Kandidaten ist übrigens unabhängig von der Gewichtsschranke des Rucksacks, d.h., für jede beliebige Gewichtsschranke $T$ kann man unter den Pareto-optimalen Teilmengen die optimale Lösung finden.

Ein kleines Detail, das wir bei der obigen Diskussion ignoriert haben, ist die Tatsache, dass zwei Teilmengen gleiches Gewicht oder auch gleichen Profit haben können. Deshalb definiert man, dass ein Punkt $A$ einen Punkt $B$ dominiert, genau dann, wenn $A$ nicht schwerer als $B$ und $A$ nicht weniger Profit als $B$ hat, jedoch nicht gleichzeitig sowohl die Profite als auch die Gewichte gleich sind.

Aber wie kann man eine Liste aller Pareto-optimalen Teilmengen berechnen, ohne explizit alle möglichen Teilmengen auszuprobieren? Betrachten wir das folgende kleine Beispiel mit zunächst nur drei Objekten. Die $2^3 = 8$ verschiedenen Teilmengen sind wieder in ein Gewicht-Profit-Diagramm eingezeichnet.




Was passiert, wenn man ein zusätzliches viertes Objekt zur Auswahl hat? Jede der 8 Teilmengen im Diagramm kann man erweitern, indem das vierte Objekt hinzufügt wird. Somit erhalten wir 8 neue Teilmengen. Jeder schwarze Punkt im Diagramm erzeugt also einen neuen roten Punkt, der jeweils um denselben Betrag nach rechts oben verschoben ist. Die Verschiebung nach rechts entspricht dem Gewicht, die nach oben dem Profit des zusätzlichen vierten Objektes. Die rote Punktmenge ist also nur eine verschobene Kopie der schwarzen.




Wie verändert sich die Pareto-Optimalität der Punkte durch das Hinzufügen des vierten Objektes? Betrachten wir als Erstes einen schwarzen Punkt, der nicht Pareto-optimal ist, der also von einem anderen schwarzen Punkt dominiert wird. Dieser bleibt natürlich dominiert, auch wenn noch zusätzlich die roten Punkte hinzukommen. Auch rote Punkte, die aus dominierten schwarzen Punkten erzeugt wurden, können nicht Pareto-optimal sein, da diese dann von einem roten Punkt dominiert werden. Man braucht somit nur die Pareto-optimalen schwarzen Punkte und die aus ihnen erzeugten roten Punkte zu berücksichtigen. Jetzt kann es noch passieren, dass ein solcher schwarzer Punkt von einem roten dominiert wird oder umgekehrt. Das müssen wir in unserem Algorithmus noch beachten.

Zum Berechnen aller Pareto-optimalen Lösungen für 4 Objekte reicht es also, dass man alle Pareto-optimalen Lösungen für 3 Objekte kennt. Oder, allgemein formuliert, wenn man alle Pareto-optimalen Lösungen für $i$ Objekte kennt, kann man daraus leicht alle Pareto-optimalen Lösungen für $i+1$ Objekte konstruieren: Als Erstes erzeugt man dazu alle roten Punkte, die aus den schwarzen Pareto-optimalen Punkten hervorgehen. Nun muss man diejenigen schwarzen Punkte löschen, die von einem roten Punkt dominiert werden und somit nicht mehr Pareto-optimal sind. Umgekehrt muss man nun noch diejenigen roten Punkte löschen, die von einem schwarzen Punkt dominiert werden. Bleibt die Frage, mit welcher Liste man anfängt. Für $i = 0$, wenn also gar kein Objekt zur Auswahl steht, gibt es nur den Punkt $(0, 0)$, der dem leeren Rucksack entspricht. Dieser ist offensichtlich Pareto-optimal und somit haben wir eine Liste der Länge eins, mit der wir beginnen können. Dieser Algorithmus wurde schon 1969 von Nemhauser und Ullmann erfunden. Wir schauen uns eine mögliche Umsetzung genauer an.

### Der Algorithmus von Nemhauser und Ullmann

Für eine effiziente Implementierung werden die Pareto-optimalen Punkte in einer Liste verwaltet, wobei die Punkte in der Liste nach aufsteigendem Gewicht sortiert sind. Zunächst wird eine Liste $L_0$ erzeugt, die nur den Punkt

$(0,0)$ enthält. Nun berechnet man nacheinander die Listen $L_1, L_2, \ldots, L_n$, wobei die $i$-te Liste $L_i$ alle Pareto-optimalen Punkte beinhaltet, die man erhält, wenn man nur die Objekte 1 bis $i$ berücksichtigt, also die Objekte $i+1$ bis $n$ außer Acht lässt.

Um die Liste $L_i$ aus der Liste $L_{i-1}$ zu berechnen, erzeugen wir zunächst eine weitere Liste $L'_{i-1}$, die die verschobene rote Punktmenge enthält. Jeder Punkt muss dazu nur kopiert und um das Gewicht des $i$-ten Objektes nach rechts und um den Profit des $i$-ten Objektes nach oben verschoben werden. Jetzt müssen die beiden Listen $L_{i-1}$ und $L'_{i-1}$ verschmolzen werden, so dass nur Pareto-optimale Punkte übrig bleiben. Da die Punkte in den Listen nach Gewicht (und somit auch nach Profit – Warum?) sortiert sind, können wir durch einmaliges paralleles Scannen beider Listen die dominierten Punkte herausfiltern. Somit ist die Zeit, die zum Verschmelzen benötigt wird, proportional zur Länge der Listen.

Der Algorithmus MERGE verschmilzt zwei sortierte Listen $L$ und $L'$ zu einer Ergebnisliste $E$.

```
1  procedure MERGE (L, L')
2  BEGIN
3     PMAX = −1; E = {}
4     REPEAT
5        Scanne L bis zu einem Punkt (w,p) mit Profit p > PMAX
6        Scanne L' bis zu einem Punkt (w',p') mit Profit p' > PMAX
7        Falls in Zeile 5 kein Punkt gefunden wurde (L fertig gescannt)
8           Füge die restlichen Punkte aus L' in E ein; RETURN(E)
9        Falls in Zeile 6 kein Punkt gefunden wurde (L' fertig gescannt)
10          Füge die restlichen Punkte aus L in E ein; RETURN(E)
11       IF (w < w') OR (w = w' AND p > p')
12          THEN Füge (w,p) in E ein und setze PMAX := p
13          ELSE Füge (w',p') in E ein und setze PMAX := p'
14 END
```

Die zuletzt erzeugte Liste $L_n$ enthält die Pareto-optimalen Punkte bezüglich aller $n$ Objekte. Zum Schluss wird aus den Punkten in $L_n$ derjenige ausgewählt, der den höchsten Profit erzielt und dabei ein Gewicht von höchstens $T$ aufweist. Die zu diesem Punkt gehörende Teilmenge von Objekten ist die gesuchte optimale Lösung.

Ist dieser Algorithmus jetzt besser als das Ausprobieren aller Teilmengen? – Nicht in jedem Fall. Es kann nämlich passieren, dass alle Teilmengen Pareto-optimal sind. Dies passiert beispielsweise, wenn alle Objekte dieselbe Profitdichte und gleichzeitig alle Teilmengen unterschiedliche Gewichtswerte haben. In typischen Anwendungen ist dies aber nicht der Fall. Häufig ist die Anzahl der Pareto-optimalen Teilmengen wesentlich geringer als $2^n$. In unserem Beispiel haben wir die Gewichte und Profite der acht Objekte zufällig erzeugt. Nur 17 der 256 Teilmengen sind Pareto-optimal. Um die Anzahl der Pareto-optimalen Teilmengen abzuschätzen, gibt es einige experimentelle

und mathematische Analysen. Aus diesen Analysen folgt, dass typischerweise nur ein äußerst kleiner Bruchteil der Teilmengen Pareto-optimal ist. Deshalb lassen sich mit dem beschriebenen Algorithmus ohne weiteres Probleme mit mehreren tausend Objekten lösen.

## Zum Weiterlesen

1. H. Kellerer, U. Pferschy und D. Pisinger: *Knapsack Problems*. Springer, 2004.

   In diesem englischsprachigen Lehrbuch werden auf umfassende Weise das Rucksack-Problem sowie verschiedene verwandte Probleme besprochen. Es gibt eine gute Übersicht über den aktuellen Stand der Forschung, über verschiedene Algorithmen und Anwendungen.

2. S. Martello und P. Toth: *Knapsack Problems: Algorithms and Implementations*. Willey, Chichster, 1990.

   Ein älteres, nicht mehr ganz aktuelles Lehrbuch.

3. In der deutschsprachigen Ausgabe der Wikipedia wird neben der bildlichen Beschreibung des Problems ein anderer Algorithmus vorgestellt, der auf dem Prinzip der dynamischen Programmierung (siehe Kap. 32) beruht. Dieser Algorithmus gilt als der Standardalgorithmus zum Lösen des Rucksackproblems, ist allerdings in vielen Fällen sehr viel langsamer und speicherintensiver als der hier vorgestellte.
   `http://de.wikipedia.org/wiki/Rucksackproblem`

4. Das Rucksackproblem kann man als Optimierungsproblem mit zwei Zielen auffassen. Zum einen möchte man das Gewicht der ausgewählten Objekte minimieren, gleichzeitig jedoch deren Profit maximieren. Auch bei anderen Problemen gibt es häufig mehrere konkurrierende Ziele. Beispielsweise ist beim Kürzeste-Wege-Problem (siehe Kap. 34) der kürzeste Weg nicht unbedingt der schnellste, da man beispielsweise auf Autobahnen viel schneller fahren kann als in der Stadt. Hier könnte man jedem Weg von A nach B zwei Zahlen zuordnen, die Länge und die Zeit, die man für diesen Weg benötigt. Entsprechend ist dann ein Weg Pareto-optimal, wenn man keinen anderen Weg finden kann, der kürzer ist und für den man auch noch weniger Zeit benötigt. In der Wikipedia findet man einen Artikel über Pareto-Optimierung:
   `http://de.wikipedia.org/wiki/Pareto-Optimierung`

# 42

# Das Travelling Salesman Problem

Stefan Näher

Universität Trier

## Das Problem des Handlungsreisenden

Das Travelling Salesman Problem (TSP) ist eines der bekanntesten und meist untersuchten Optimierungsprobleme. Es stellt sich wie folgt: Ein *Handlungsreisender* soll in einer *Rundreise* (auch Tour genannt) $n$ vorgegebene Städte besuchen. Er startet dazu in einer dieser Städte, besucht nacheinander die restlichen Städte und kehrt schließlich zu seinem Ausgangspunkt zurück.


Das eigentliche Optimierungsproblem besteht nun darin, die Rundreise so zu planen, dass ihre Gesamtlänge minimiert wird. Die Entfernungen zwischen allen Paaren von Städten sind hierzu (z. B. in Form einer Tabelle) gegeben. Neben den tatsächlichen geometrischen Abständen kann es sich dabei auch

um Reisezeiten oder um andere Kosten (z. B. für Treibstoff) handeln. Ziel ist also, die Rundreise so zu planen, dass der Gesamtreiseweg bzw. die Reisezeit oder aber die insgesamt anfallenden Kosten minimiert werden.

TSP gehört zu einer Klasse von sehr schwierigen Problemen, den so genannten NP-vollständigen Problemen. Auch das in Kap. 41 behandelte Rucksackproblem gehört dazu. Für diese Klasse von Problemen ist im Allgemeinen kein effizienter Algorithmus bekannt. Es wird vielmehr angenommen, dass jeder Algorithmus zur exakten Lösung dieser Probleme im schlechtesten Fall exponentiell viele Rechenschritte ausführen muss. Bis heute existiert jedoch kein mathematischer Beweis für diese Vermutung. Für spezielle Varianten von TSP und zur Berechnung von Näherungslösungen, die auch Rundreisen erlauben, die etwas teurer sind als die optimale Tour, sind allerdings sehr schnelle Algorithmen bekannt.

Das TSP-Problem tritt in der Praxis in vielen Anwendungen als Teilproblem auf. Hierzu gehören z. B. Optimierungsprobleme in der Verkehrsplanung und Logistik oder beim Entwurf integrierter Schaltungen und Mikrochips. Man kann auch andere Probleme aus der Klasse der NP-vollständigen Probleme auf TSP zurückführen.

Wir werden zunächst eine sehr einfache Strategie zur exakten Lösung von TSP untersuchen, die so genannte *brute-force* Technik (wir nennen sie auch Holzhammermethode), und zeigen, wie katastrophal langsam ein Algorithmus mit exponentieller Laufzeit sein kann.

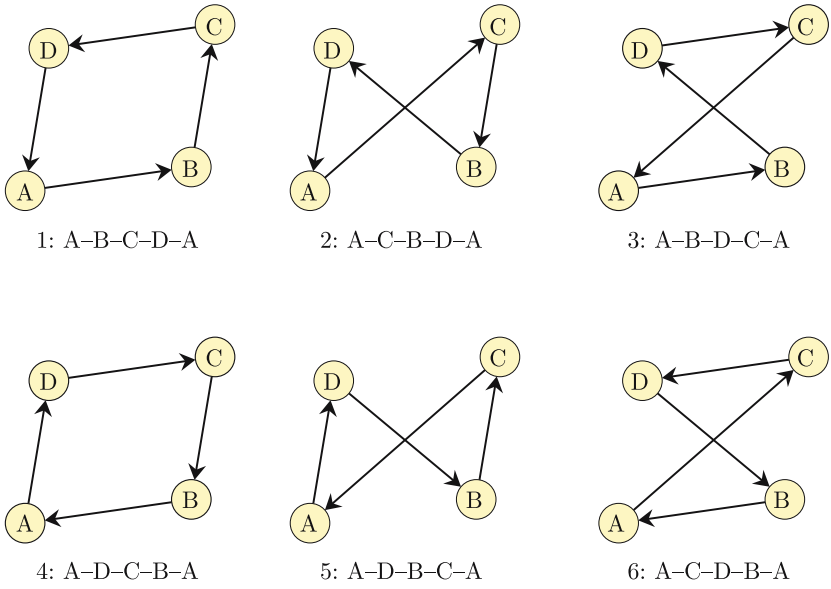



**Abb. 42.1.** Alle möglichen Rundreisen für 4 Städte

## Die Holzhammer-Methode

Die „Holzhammer-Methode“, die auch *brute-force* oder naive Methode genannt wird, ist der einfachste Algorithmus zur exakten Lösung von TSP. Er betrachtet nacheinander alle möglichen Rundreisen, berechnet für jede die Gesamtlänge und ermittelt durch Vergleich der so erhaltenen Werte die kürzeste Tour. Leider wächst aber die Zahl aller möglichen Rundreisen mit steigender Zahl $n$ der gegebenen Städte sehr schnell. Man kann sich leicht überlegen, dass es insgesamt $(n-1)! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (n-1)$ verschiedene Möglichkeiten gibt, $n$ Orte mit einer Rundreise zu besuchen: Jede Rundreise muss in einer Stadt (z. B. der ersten) beginnen, die restlichen $n-1$ Städte in irgendeiner Reihenfolge besuchen und dann wieder zur ersten Stadt zurückkehren. Es gibt aber genau $(n-1)!$ Möglichkeiten, die restlichen $(n-1)$ Städte nacheinander aufzuzählen (man spricht auch von der Zahl der Permutationen). Abbildung 42.1 zeigt die $6 = (4-1)!$ möglichen Rundreisen für 4 Städte A, B, C, D.

Bei den unteren drei Touren handelt es sich, wie man leicht sieht, um Umkehrungen der oberen drei Touren, so dass man eigentlich nur für die Hälfte der Rundreisen die Gesamtlängen ausrechnen muss. Bei $n$ Städten muss der Algorithmus also $\frac{1}{2} \cdot (n-1)!$ Rundreisen betrachten. Es gibt übrigens auch Varianten von TSP ohne diese Symmetrie-Eigenschaft. Dann muss man tatsächlich alle Touren untersuchen.

**Tabelle 42.1.** Anzahl aller möglichen Rundreisen und Laufzeit der Holzhammermethode

| Städte | mögliche Rundreisen | Laufzeit |
|---|---|---|
| 3 | 1 | 1 msec |
| 4 | 3 | 3 msec |
| 5 | 12 | 12 msec |
| 6 | 60 | 60 msec |
| 7 | 360 | 360 msec |
| 8 | 2.520 | 2,5 sec |
| 9 | 20.160 | 20 sec |
| 10 | 181.440 | 3 min |
| 11 | 1.814.400 | 0,5 Stunden |
| 12 | 19.958.400 | 5,5 Stunden |
| 13 | 239.500.800 | 2,8 Tage |
| 14 | 3.113.510.400 | 36 Tage |
| 15 | 43.589.145.600 | 1,3 Jahre |
| 16 | 653.837.184.000 | **20 Jahre** |

Tabelle 42.1 zeigt, wie gewaltig schnell die Zahl der Rundreisen mit steigendem $n$ wächst. Die letzte Spalte gibt eine Abschätzung für die Laufzeit eines Programms an, das TSP mithilfe der Holzhammermethode zu lösen versucht. Dabei wird angenommen, dass die Bearbeitung einer einzigen Rundreise etwa eine Millisekunde Zeit benötigt. Die letzte Zeile der Tabelle zeigt, dass ein solches Programm für die Lösung eines TSP-Problems mit nur 16 Städten mehr als eine halbe Billionen Rundreisen betrachten muss und so über 20 Jahre Rechenzeit benötigt. Laufzeiten im Minutenbereich sind nur für Probleme mit maximal 10 Städten möglich. Im Kap. 41 dieses Buches tritt ein ähnliches Problem mit einer extrem schnell wachsenden Anzahl von Möglichkeiten auf.

## Dynamisches Programmieren

Eine Technik, die in der Informatik und insbesondere beim Entwurf von Algorithmen sehr häufig eingesetzt wird, ist die *Rekursion*. Dabei versucht man, die Lösung eines Problems auf kleinere Probleme der gleichen Art zurückzuführen, siehe auch Kap. 3. Das so genannte *Dynamische Programmieren* ist eine spezielle Variante dieser Technik, bei der man Zwischenresultate von rekursiv definierten Teilproblemen in einer Tabelle verwaltet.

Wir nehmen an, dass die $n$ Städte mit den Zahlen von 1 bis $n$ durchnummeriert sind, d. h., wir können die Menge der Städte als Menge $S = \{ 1, 2, \ldots, n \}$ schreiben, und dass die Entfernungen oder Kosten in einer Tabelle $DIST$ gegeben sind, d. h., $DIST[i, j]$ bezeichnet die Entfernung von Stadt $i$ nach Stadt $j$. Da eine Rundreise in jeder beliebigen Stadt beginnen und enden kann, nehmen wir zur Vereinfachung an, dass *jede* Rundreise in der Stadt 1 beginnt, dann zu einer Stadt $i \in \{ 2, \ldots, n \}$ führt, dann alle restlichen Städte $\{ 2, \ldots, i-1, i+1, \ldots, n \}$ besucht und schließlich zur Stadt 1 zurückkehrt.

Sei $i$ eine beliebige Stadt und $A$ eine Menge von Städten. Dann definiere $L(i, A)$ als die Länge eines kürzesten Weges,

- der in der Stadt $i$ beginnt,
- jede Stadt in der Menge $A$ genau einmal besucht
- und in Stadt 1 endet.

Für die Berechung der $L(i, A)$-Werte sind folgende Beobachtungen nützlich.

1. Für jede Stadt $i \in S$ ist $L(i, \emptyset) = DIST[i, 1]$.
   Es ist klar, dass der kürzeste Weg von $i$ nach 1, der keine andere Stadt besucht, die direkte Verbindung von $i$ nach 1 ist.
2. Für jede Stadt $i \in S$ und Teilmenge $A \subseteq S \setminus \{1, i\}$ gilt:
   $L(i, A) = \min\{DIST[i, j] + L(j, A \setminus \{j\}) \mid j \in A \}$.
   Wenn die Menge $A$ nicht leer ist, dann kann man den optimalen Weg wie folgt rekursiv definieren: Probiere für alle $j \in A$ den Weg aus, der von $i$ aus zunächst nach $j$ führt. Für den besten dieser Wege muss gelten, dass

auch der Rest des Weges (von $j$ nach 1) optimal ist. Dieser hat aber nach unserer Definiton die Länge $L(j, A \setminus \{j\})$.

3. $L(1, \{\ 2, \ldots, n\ \})$ ist die Länge einer optimalen Rundreise.
   Dies folgt aus der Tatsache, dass jede Rundreise in der Stadt 1 beginnt, dann alle anderen Städte besucht und schließlich nach 1 zurückkehrt.

Aus diesen Beobachtungen folgt ein rekursiver Algorithmus, der die Werte von $L(i, A)$ für immer größere Teilmengen $A$ in eine Tabelle einträgt. Dabei ist der Fall $A = \emptyset$ die Verankerung der Rekursion.

**Algorithmus TSP mit dynamischer Programmierung**

```
1   for i := 1 to n do L(i, ∅) := DIST[i, 1] endfor;
2   for k := 1 to n − 1 do
3       forall A ⊆ S \ {1} with |A| = k do
4           for i := 1 to n do
5               if i ∉ A then
6                   L(i, A) = min{DIST[i, j] + L(j, A \ {j}) | j ∈ A }.
7               endif
8           endfor
9       endfor
10  endfor
11  return L(1, {2, ..., n});
```

**Tabelle 42.2.** Größe der Tabelle und Laufzeit beim Dynamischen Programmieren

| $n$ Städte | Größe der Tabelle $(n^2 \cdot 2^n)$ | Laufzeit |
|---|---|---|
| 3 | 72 | 72 msec |
| 4 | 256 | 0,4 sec |
| 5 | 800 | 0,8 sec |
| 6 | 2304 | 2,3 sec |
| 7 | 6272 | 6,3 sec |
| 8 | 16.384 | 16 sec |
| 9 | 41.472 | 41 sec |
| 10 | 102.400 | 102 sec |
| 11 | 247.808 | 4,1 Minuten |
| 12 | 589.824 | 9,8 Minuten |
| 13 | 1.384.448 | 23 Minuten |
| 14 | 3.211.264 | 54 Minuten |
| 15 | 7.372.800 | 2 Stunden |
| 16 | 16.777.216 | **4,7 Stunden** |

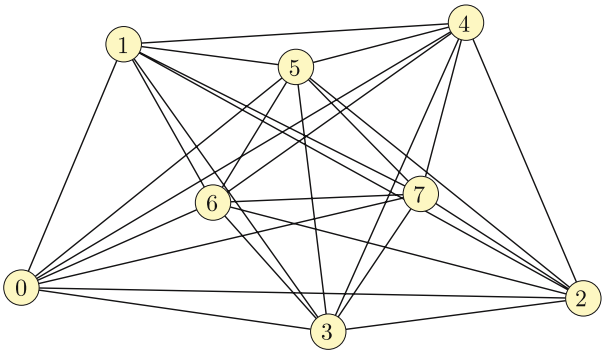



**Abb. 42.2.** Der vollständige Graph aller direkten Reisewege

Der Algorithmus füllt eine Tabelle mit $n$ Zeilen ($i = 1, \ldots, n$) und maximal $2^n$ Spalten (für alle Teilmengen der Größe $k <= n - 1$), d. h., die Tabelle enthält maximal $n \cdot 2^n$ Einträge. Pro Eintrag wird eine Minimumssuche über $n$ Werte ausgeführt (Zeilen 4 bis 7). Damit wird Zeile 6 des Algorithmus maximal $n \cdot n \cdot 2^n = n^2 \cdot 2^n$ mal ausgeführt.

Tabelle 42.2 zeigt, wie sich diese Zahl mit wachsendem $n$ entwickelt. Außerdem gibt sie in der letzten Spalte eine Abschätzung für die Laufzeit, wenn wir annehmen, dass die Berechnung eines Tabelleneintrags eine Millisekunde Zeit kostet. Man sieht, dass die Laufzeit immer noch mehr als exponentiell mit $n$ wächst. Allerdings ergibt sich im Vergleich zur Holzhammer-Methode doch schon eine gewaltige Verbesserung. Die Zeit zur Berechnung einer optimalen Rundreise durch 16 Städte hat sich von 20 Jahren auf 5 Stunden verringert. *Hinweis*: Ein ernstes Problem beim Dynamischen Programmieren ist die exponentiell wachsende Tabelle und der damit verbundene Speicherbedarf.

## Näherungslösungen

Der Ansatz des Dynamischen Programmierens hat zwar eine Verbesserung der Laufzeit bewirkt. Aber diese ist immer noch exponentiell und daher für große Werte von $n$ völlig unbrauchbar. Auch der gewaltige Speicherbedarf ist ein Problem. In diesem Abschnitt lernen wir ein Verfahren kennen, das nicht unbedingt die optimale Rundreise findet, aber eine relativ gute Lösung berechnet. Genauer gesagt, der nun vorgestellte Algorithmus findet eine Rundreise, die *nicht mehr als doppelt so lang* wie die kürzeste Reise ist. Ein solches Verfahren nennt man auch eine *Heuristik*.

Wir modellieren hierzu TSP als ein Problem auf einem Graphen, wobei die Knoten die Städte darstellen und eine Kante zwischen zwei Knoten $i$ und $j$ die direkte Reise von $i$ nach $j$ beschreibt und entsprechend mit der Distanz bzw. den Kosten für diese Reise beschriftet ist. Da beim TSP-Problem für jedes

Paar $(i, j)$ von Städten eine direkte Reise von $i$ nach $j$ möglich ist, handelt es sich bei dem so konstruierten Graphen $G = (V, E)$ um den *vollständigen Graphen*, der alle Knotenpaare als Kanten enthält, d. h., $E = V \times V$. Eine Rundreise wird in diesem Modell durch einen Kreis in $G$ dargestellt, der jeden Knoten genau einmal enthält. Ein solcher Kreis wird auch als *Hamilton-Kreis* bezeichnet.

Es gibt einen einfachen Zusammenhang zwischen möglichen Rundreisen (d. h. Hamilton-Kreisen) und speziellen Teilgraphen von $G$, den so genannen aufspannenden Bäumen. Ein *aufspannender Baum* ist ein kreisfreier Teilgraph, der alle Knoten von $G$ miteinander verbindet. Ein *minimal aufspannender Baum* ist ein aufspannender Baum, dessen Gesamtkosten (d. h. die Summe aller Kantenkosten) minimal sind.

Wenn man aus einer Rundreise eine beliebige Kante entfernt, so erhält man einen so genannten *Hamilton-Pfad*. Da ein Hamilton-Pfad die Bedingungen eines aufspannenden Baumes erfüllt (ein kreisfreier Teilgraph, der alle Knoten miteinander verbindet), sind dessen Gesamtkosten mindestens so groß wie die eines minimal aufspannenden Baumes. Mit anderen Worten, die Gesamtkosten eines minimal aufpannenden Baumes sind kleiner oder gleich den Gesamtkosten eines bestmöglichen Hamilton-Pfades und damit auch einer optimalen Rundreise.

Algorithmen zur Berechnung eines minimal aufspannenden Baumes werden im Kap. 35 dieses Buches behandelt. Diese Algorithmen sind sehr effizient. Der teuerste Schritt ist das Sortieren der Kanten nach ihren Kosten. Ausgehend von einem minimal aufspannenden Baum kann man nun sehr leicht eine TSP-Tour berechnen. Das vollständige Verfahren wird in den folgenden Schritten beschrieben. Die dazugehörigen Abbildungen wurden mit einem Programm erzeugt, das man von der Seite `http://www-i1.informatik.rwth-aachen.de/~algorithmus/Algorithmen/algo40/tsp.exe` herunterladen kann.

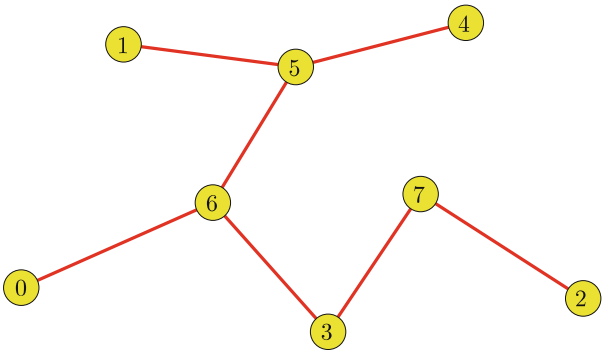



**Abb. 42.3.** Der minimal aufspannende Baum

## Algorithmus

1. Konstruiere den vollständigen Graphen $G = (V, E)$, wobei $V$ die Menge der Städte ist und $E = V \times V$ (siehe Abb. 42.2).
2. Berechne einen minimal aufspannenden Baum $T$ von $G$. Dabei sind die Kosten einer Kante $(v, w)$ gleich der Entfernung zwischen den Städten $v$ und $w$ ($DIST[v, w]$). Abbildung 42.3 zeigt das Ergebnis dieses Rechenschrittes für unser Beispielproblem.
3. Im nächsten Schritt konstruiert man aus dem Baum $T$ eine erste Tour, indem man $T$ einfach entlang seiner Kanten einmal umrundet (siehe Abb. 42.4). Die Gesamtlänge dieser Reise ist offensichtlich genau doppelt so groß ist wie die Kosten von $T$, da man jede Kante zweimal verwendet. Aus unseren Vorüberlegungen schließen wir, dass die Gesamtlänge dieser Reise höchstens *doppelt so groß* wie die Länge einer optimalen TSP-Rundreise ist.
4. Die im letzten Schritt konstruierte Tour ist offensichtlich nicht optimal und eigentlich sogar keine korrekte Rundreise, da sie jeden Knoten zweimal besucht. Man kann sie aber leicht in eine korrekte und im Allgemeinen auch kürzere Rundreise umwandeln, indem man jeweils drei aufeinanderfolgende Knoten $a, b, c$ untersucht und testet, ob man die beiden Kanten $a \longrightarrow b \longrightarrow c$ durch die Abkürzung $a \longrightarrow c$ ersetzen kann, ohne dass der Knoten $b$ isoliert wird. Hierbei wird vorausgesetzt, dass die Dreiecksungleichung gilt. Das Resultat dieses Schrittes sehen wir in Abb. 42.5.

Tabelle 42.3 zeigt die Laufzeit des MST-Algorithmus für zufällig ausgewählte Städte. Das hier verwendete Programm kann in einer Millisekunde einen minimal aufspannenden Baum für einen Graphen mit 1000 Kanten berechnen. Wie man sieht, findet der Algorithmus eine Näherungslösung für 1000 Städte in etwa einer Minute. In weiteren Bearbeitungsschritten kann man die Rundreise weiter verbessern. Es existieren viele Heuristiken, um dies

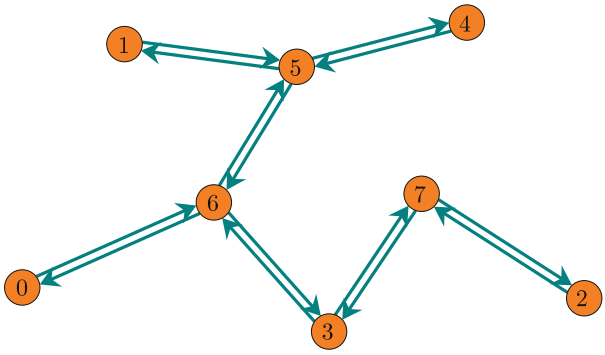



**Abb. 42.4.** Einmal um den Minimum-Spanning-Tree

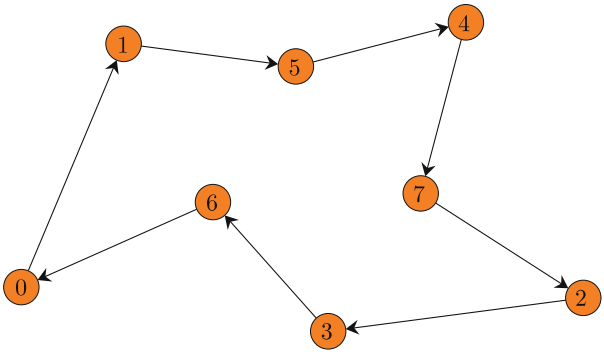



**Abb. 42.5.** Die Minimum-Spanning-Tour

**Tabelle 42.3.** Die Laufzeit der MST-Heuristik

| Städte | Laufzeit |
|---|---|
| 100 | 0,01 sec |
| 200 | 0,08 sec |
| 300 | 0,36 sec |
| 400 | 1,30 sec |
| 500 | 3,62 sec |
| 600 | 8,27 sec |
| 700 | 16,07 sec |
| 800 | 29,35 sec |
| 900 | 50,22 sec |
| 1000 | 85,38 sec |

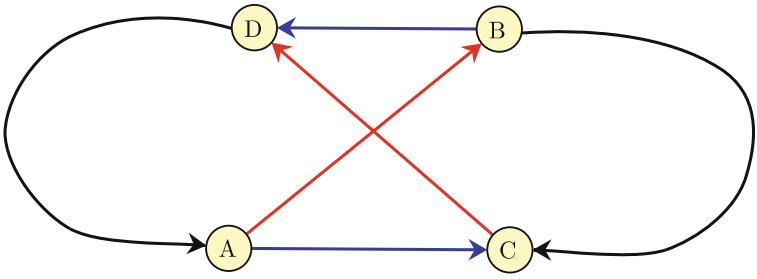



**Abb. 42.6.** Ein Schritt der 2-OPT-Heuristik

zu erreichen, z. B. das **2-OPT** Verfahren, dessen Arbeitsweise in Abb. 42.6 dargestellt ist. Man betrachtet jeweils 2 Kanten der berechneten Tour, sagen wir $A \longrightarrow B$ und $C \longrightarrow D$. Wenn man diese beiden Kanten entfernt, zerfällt die Tour in zwei Teilstücke von $B$ nach $C$ und von $D$ nach $A$. Dann wird getestet, ob man diese Teile durch Einfügen der Kanten $A \longrightarrow C$ und $D \longrightarrow B$ zu einer kürzeren Tour zusammenfügen kann.

## Einige Bemerkungen zum Schluss

- Es gibt eine Reihe weiterer Heuristiken, die oft zu sehr guten Ergebnissen führen und häufig sogar die optimale Rundreise liefern.
- Auch bei der Berechnung von exakten Lösungen für TSP hat man in den letzten Jahren erhebliche Fortschritte erzielt. Es gibt mittlerweile Programme, die in der Praxis vorkommende TSP-Probleme mit mehreren tausend Städten in wenigen Stunden Rechenzeit exakt lösen können.
- Im schlechtesten Fall bleibt das Lösen von TSP-Problemen sehr schwierig und erfordert exponentiell viele Rechenschritte.

## Zum Weiterlesen

1. Wikipedia: Problem des Handlungsreisenden:
   `http://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden`
2. Eine Einführung von Martin Grötschel und Manfred Padberg:
   `http://elib.zib.de/pub/UserHome/Groetschel/Spektrum/index2.html`
3. The Travelling Salesman Problem Home Page:
   `http://www.tsp.gatech.edu`
4. Kapitel 41 (Das Rucksackproblem)

   Ein Optimierungproblem mit einer ähnlich schnell wachsenden Anzahl von Möglichkeiten.
5. Kapitel 3 (Schnelle Sortieralgorithmen)

   Eine Anwendung und Einführung in die Technik der rekursiven Algorithmen.
6. Ein Demo-Programm, das die Abbildungen dieses Kapitels lieferte:
   `http://www-i1.informatik.rwth-aachen.de/~algorithmus/Algorithmen/algo40/tsp.exe`

# 43

# Simulated Annealing

Peter Rossmanith

RWTH Aachen

Betrachten wir ein einfaches kombinatorisches Puzzlespiel: Wir haben verschiedene quadratische Dominosteine zur Verfügung, die in einer Holzkiste auf $12 \times 8$ Positionen gesetzt werden dürfen. Anfangs mögen diese Steine wie in Abb. 43.1 gezeigt angeordnet sein. Wie dort zu sehen ist, haben die Steine vier Seiten mit den Farben Blau, Gelb, Grün und Orange.

Bei diesem Puzzle dürfen wir die Steine beliebig umsetzen, aber nicht verdrehen. Unser Ziel ist es, möglichst viele gleichfarbige benachbarte Flächen

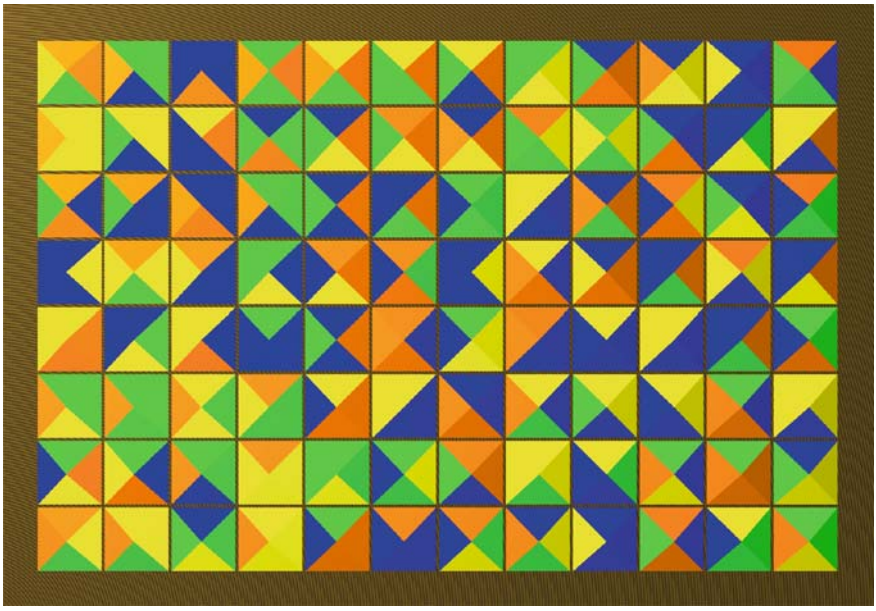


**Abb. 43.1.** Ausgangslage des Dominospiels: Die Steine liegen zufällig in der Kiste. Auch in dieser Lage ergeben sich immerhin schon 36 Punkte

zu erhalten: Für jedes solcher Paare erhalten wir einen Punkt. Es gibt genau 172 benachbarte Flächen, so dass wir auf keinen Fall mehr als 172 Punkte erreichen können: Jede Zeile enthält 11 Paare und jede Spalte 7. Es gibt 8 Zeilen und 12 Spalten: Insgesamt macht das $11 \cdot 8 + 7 \cdot 12 = 172$.

Die Lösung dieses Dominospiels begleitet uns in diesem Kapitel stellvertretend für viele Probleme der kombinatorischen Optimierung. Manche der Optimierungsprobleme in diesem Buch lassen sich recht effizient lösen (Kap. 34, 35, 36 und 37), andere können nur für relativ kleine Eingaben noch exakt gelöst werden (Kap. 41 und 42). Wir interessieren uns in diesem Zusammenhang hauptsächlich für letztere Probleme, die durch Probieren aller Möglichkeiten oder Backtracking nicht mehr gelöst werden können. Ein Verfahren, das geeignet ist, eine gute Lösung für viele solcher Fälle zu finden, ist *Simulated Annealing*.

## Was ist Simulated Annealing?

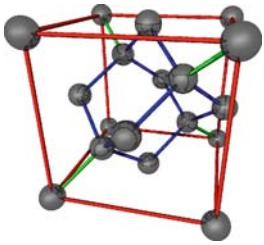
*Simulated Annealing* könnte man mit „simuliertes langsames Abkühlen“ übersetzen. Es gibt wohl kein deutsches Wort, das dem englischen *annealing* entspricht. Die eigentliche Bedeutung ist „Erhitzen und dann langsam Abkühlen“. In der Technik wird ein solches Verfahren vielfach eingesetzt. Je nachdem, ob erhitzte Metalle schnell oder langsam aus dem rotglühendem Zustand abgekühlt werden, haben sie ganz unterschiedliche Eigenschaften. Warum macht man das? Überlegen wir uns doch einmal, was dabei mit den kleinen Teilchen (Atomen) passiert, aus denen das Metall besteht:

Diese Atome sind ursprünglich in ein festes Gitter eingebunden. Wenn man das Metall nun erhitzt, fangen sie an, sich aus ihren Bindungen zu lösen und zu bewegen. Die ursprüngliche Struktur wird also zerstört. Wenn man das Metall anschließend langsam abkühlt, suchen sich die Teilchen neue Bindungen. Erstaunlicherweise verteilen sie sich dabei oft regelmäßiger als vorher. Wenn man alles richtig macht, wird das Metall weicher, flexibler und enthält weniger Unregelmäßigkeiten.

Um den Effekt des langsamen Abkühlens besser nachvollziehen zu können, kann man sich die Teilchen als kleine Kugeln vorstellen. Wenn man diese einfach in ein Gefäß wirft, dann liegen sie vielleicht ziemlich unordentlich herum. Was könnte man tun, um sie zu ordnen? Schütteln! Dadurch erhöht sich natürlich erst mal die Unordnung, und die Kugeln fliegen herum – dem Schütteln entspricht das Erhitzen. Wenn man jetzt aber immer langsamer schüttelt, dann ordnen sich die Kugeln ganz von selbst! Beendet man das Schütteln allzu plötzlich, werden die Kugeln nicht ganz dicht liegen.

Wichtig ist dies auch bei der Herstellung von Halbleitern aus Silizium, aus welchen letztendlich die Mikroprozessoren und Speicherbausteine in Compu-

tern bestehen. Hier wird ein besonders reines Siliziumkristall benötigt, das insbesondere keine Unregelmäßigkeiten enthält.



Silizium ist normalerweise polykristallin: Ähnlich wie Sandkörner liegen kleine Einkristalle eng beieinander. Ein solcher Einkristall wiederum besteht aus sehr vielen kleinen Elementarwürfeln, die fehlerfrei neben-, hinter- und übereinander gepackt sind. Links ist ein solcher Elementarwürfel eines Siliziumkristalls dargestellt. Außen bilden acht Siliziumatome einen Würfel, in dessen Inneren sich weitere zehn Siliziumatome befinden. Für die Halbleiterherstellung werden recht große Monokristalle benötigt. Zu diesem Zwecke wird eine große „Siliziumsäule“ langsam aus einer Siliziumschmelze herauskristallisiert. Anschließend werden mittels einer Säge Scheiben hergestellt, aus welchen schließlich die Halbleiterbausteine entstehen.

Der monokristalline Zustand des Siliziums ist der mit der geringsten Energie: Die Bindungen zwischen den Atomen sind so am stärksten. Nun ist der Unterschied zu unserem Dominospiel gar nicht mehr so groß. Auch hier haben wir es mit kleinen Elementarbausteinen zu tun, deren Lage zueinander geändert werden kann. Die Bindung zwischen zwei Dominosteinen soll stärker sein, wenn die gegenüberliegenden Farben gleich sind, und wir suchen einen Gesamtzustand mit möglichst geringer Energie. Wäre unser Dominospiel ein stark erhitzter Kristall, würden die Steine wild durcheinanderspringen. Je geringer aber die Temperatur wird, desto weniger können Steine aus ihren Löchern herausgerissen werden. Je mehr stabile Bindungen sie zu ihren Nachbarn vorweisen können, desto fester sitzen sie.

Es ist nicht leicht, unser Dominospiel dazu zu bringen, sich so zu verhalten, obwohl es vielleicht nicht unmöglich ist: Möglicherweise lassen sich die stärkeren Bindungen zwischen Flächen gleicher Farbe durch geschickt angebrachte Magnete verwirklichen und wir könnten das ganze Spielbrett auf einem regelbaren Rütteltisch montieren. Andererseits ist es aber viel einfacher, dieses Verfahren mithilfe eines Computers zu simulieren. Dem Schütteln entspricht in der Computersimulation das Vertauschen zweier Dominosteine. Dabei können wir leicht ausrechnen, ob und wie viel der Punktestand steigt oder sinkt. Bei großen Temperaturen tolerieren wir noch Vertauschungen, die die Punktezahl senken, während wir bei kleiner werdender Temperatur immer mehr dazu über-

gehen, nur „gute“ Veränderungen zu erlauben. Der im folgenden angegebene Algorithmus kann leicht in jeder gängigen Programmiersprache implementiert werden:

**Dominospiel**

Wiederhole sehr oft:

1. Erniedrige die Temperatur ein bisschen.
2. Vertausche zwei zufällig gewählte Dominosteine.
3. Falls dadurch die Punktzahl sinkt:
   - Entscheide zufällig, ob diese Vertauschung beibehalten wird.
   - Die Wahrscheinlichkeit dafür nimmt mit der Temperatur ab.
   - Bei negativer Entscheidung mache die Vertauschung rückgängig.

Dieser Algorithmus funktioniert erstaunlich gut für das Dominorätsel. Während der Algorithmus ausgeführt wird, ändert sich die Punktzahl sowohl nach oben als auch nach unten. Anfangs sind die Schwankungen sehr groß, aber je länger wir warten – und je kleiner die Temperatur dabei wird –, desto geringer fallen sie aus. Schließlich können wir keine Veränderung mehr beobachten. Abbildung 43.2 zeigt, wie sich die Punktzahl im Laufe der Zeit ändert, wobei nur Punktzahlen über 140 wiedergegeben werden. Dieser Bereich wird sehr schnell erreicht, wobei weitere Verbesserungen zum Ende dann immer seltener werden.

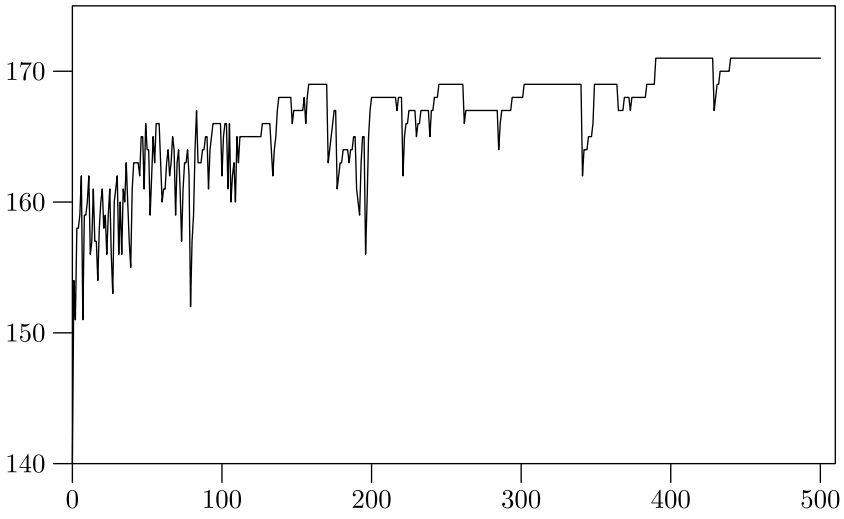



**Abb. 43.2.** Wie sich der Punktestand durch fortgesetztes Vertauschen von Steinen ändert: Die $y$-Achse zeigt den Punktestand und die $x$-Achse die Anzahl der Schritte in Einheiten von 10000. Insgesamt werden 500000 Schritte durchgeführt. Am Ende beträgt der Punktestand 171

Hier stellt sich die Frage, warum wir Vertauschungen, die die Punktzahl verringern statt erhöhen, nicht grundsätzlich zurücknehmen – auf diese Weise würden wir nie Punkte freiwillig aufgeben. Ein solches Vorgehen ist oft tatsächlich eine gute Strategie und trägt daher einen eigenen Namen: *Methode des steilsten Anstiegs*. Hier gehen wir prinzipiell nur aufwärts, nie abwärts. Wollen wir den höchsten Berggipfel erklimmen, gehen wir stets in die Richtung, die nach oben führt, bis es keine solche Richtung mehr gibt und daher ein Gipfel erreicht sein muss. Ist es aber der höchste Gipfel? Nicht unbedingt!

> *Wer den höchsten Gipfel finden will, muss bereit sein, zwischendurch auch einmal abzusteigen.*

Eine Anwendung der Methode des steilsten Anstiegs auf unser Dominoproblem führte zu einer Lösung mit 167 Punkten, von welcher aus keine weitere Verbesserung durch Vertauschen von Steinen existierte. Dies ist durchaus ein sehr gutes Ergebnis. Allerdings erreicht *Simulated Annealing* die bessere Anordnung mit 171 Punkten (siehe Abb. 43.3). Nur ein einziger Punkt ist verloren gegangen, den wir auf den ersten Blick gar nicht leicht finden können. (Tipp: Ein schräger Blick entlang der sich visuell abzeichnenden Diagonalen hilft.)

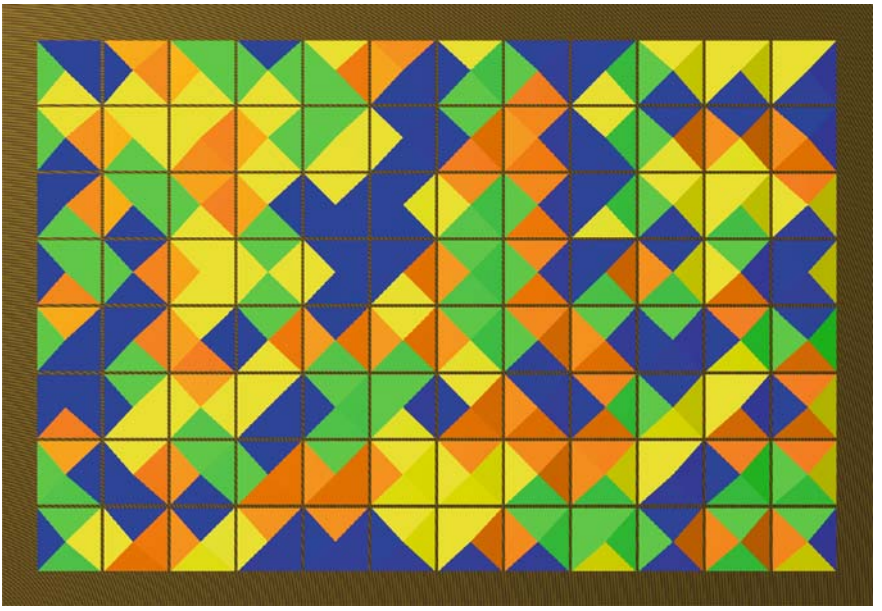


**Abb. 43.3.** Positionen der Steine nach Anwendung des Simulated Annealings: Es ergeben sich 171 Punkte und ein genauerer Blick zeigt, dass fast alle möglichen Punkte erzielt wurden

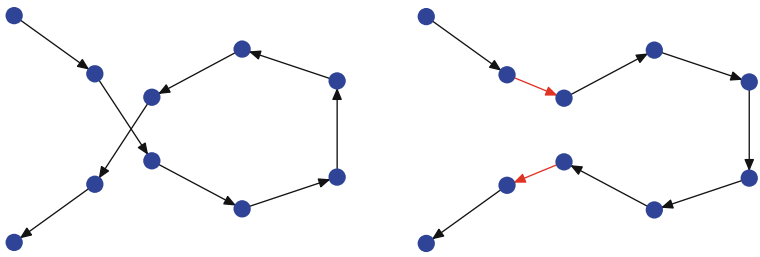
Mit 171 von theoretisch 172 möglichen Punkten hat Simulated Annealing also ein ausgezeichnetes Ergebnis geliefert. Ob sogar 172 Punkte erreicht werden können, sei an dieser Stelle als offene Frage belassen.

## Das Problem des Handlungsreisenden

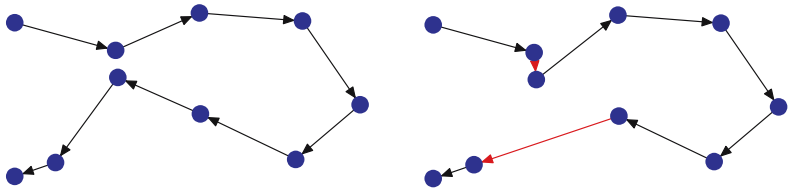
Nehmen wir einmal ein typisches Problem der Informatik, das Problem des Handelsreisenden, welches Gegenstand des Kap. 42 dieses Buches war: Wieder will ein Handelsreisender eine Menge von Städten besuchen und dabei möglichst wenig Zeit brauchen. Es geht also darum, die Reihenfolge, in der er die Städte besucht, möglichst geschickt festzulegen, so dass der zurückgelegte Weg möglichst kurz wird. Das ist erstaunlicherweise ein sehr schwieriges Problem.

Vielleicht können wir eine möglichst gute Lösung finden, indem wir *Simulated Annealing* verwenden. Irgendeine Lösung zu finden ist ja einfach: Wir nehmen eine beliebige Reihenfolge! Wir beginnen mit einer zufälligen Tour. Es ist natürlich ziemlich unwahrscheinlich, dass diese sehr kurz ist.

Wie können wir diese Tour verbessern, indem wir kleine Änderungen vornehmen? Nun, wir könnten einfach zufällig ein Teilstück wählen und dieses fortan in umgekehrter Reihenfolge durchlaufen:



Eine andere Möglichkeit besteht darin, eine Stadt zu einem anderen Zeitpunkt zu besuchen und die Reihenfolge aller anderen Städte beizubehalten:



Nach Änderungen beider Arten kann die jeweilige Tour besser oder schlechter sein. Wenn sie besser ist, behalten wir die Änderung, sonst machen wir sie rückgängig.

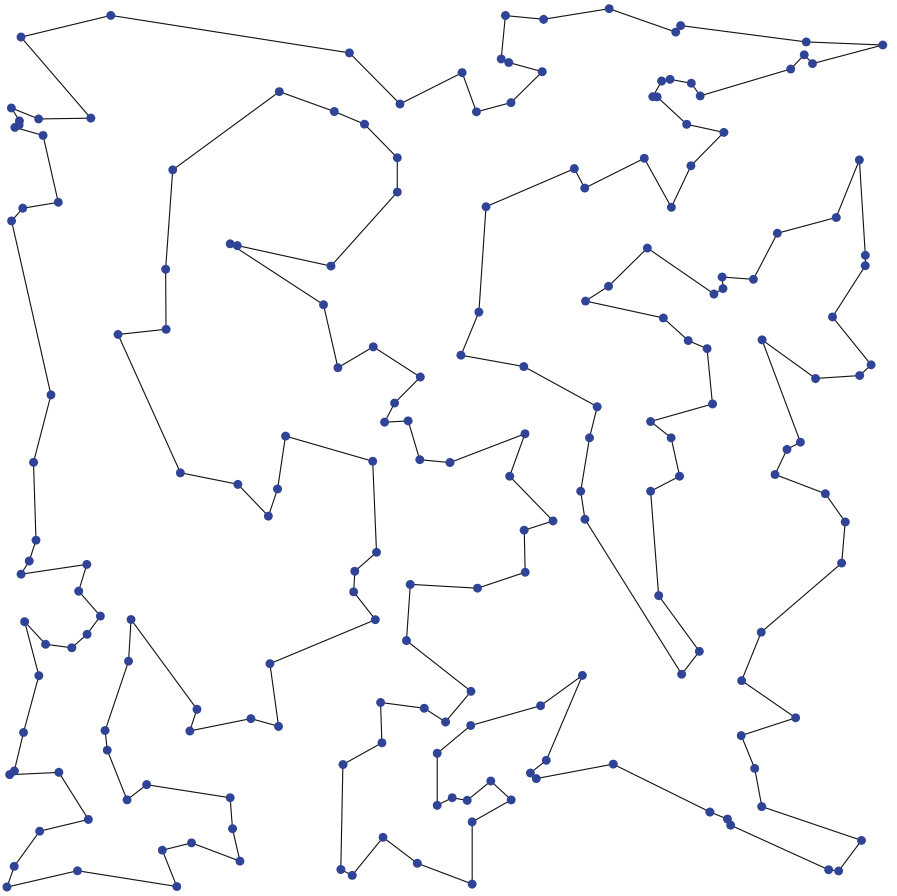


**Abb. 43.4.** Eine Rundreise, die 200 Städte miteinander verbindet

Ein große Rundreise, die über 200 Städte führt, findet sich in Abb. 43.4. Die gezeigte Tour wurde natürlich mithilfe der beiden oben aufgeführten Veränderungsregeln und Simulated Annealing gefunden.

## Weitere Anwendungen

Simulated Annealing läßt sich anwenden, wenn die folgenden Bedingungen erfüllt sind:

1. Die Güte der Lösung muss sich durch eine Zahl darstellen lassen.
2. Eine anfängliche Lösung muss leicht zu berechnen sein.
3. Es gibt einfache Veränderungsregeln, die eine Lösung lokal verändern.
4. Jede Lösung muss von jeder anderen Lösung durch Anwendung dieser Regeln erzeugt werden können.

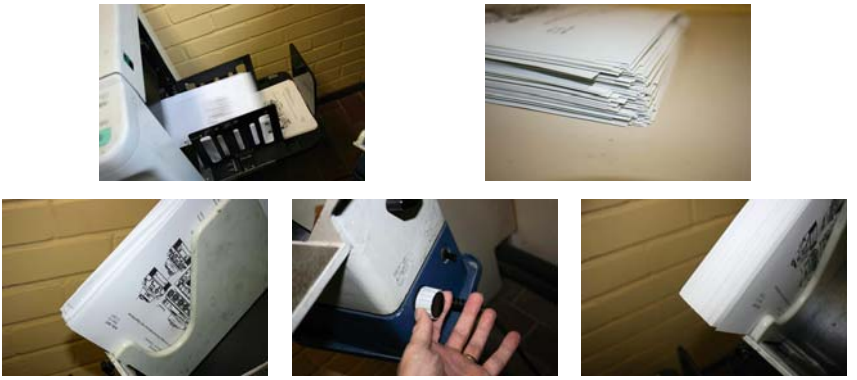


**Abb. 43.5.** Ein Bogenrüttler erzeugt durch Vibration einen passgenauen Papierstapel

Da diese Bedingungen alles andere als restriktiv sind, stellt sich heraus, dass erstaunlich viele Probleme durch Simulated Annealing gelöst werden können.

Zuletzt sei erwähnt, dass die Informatikerinnen und Informatiker nicht die ersten waren, die Kristallwachstum bei langsamer Abkühlung zum Vorbild für Lösungen technischer Probleme anwendeten. Nimmt man z. B. dieses Buch zur Hand, fällt vielleicht der passgenaue Abschluß der Seitenränder auf, welcher einer Schneidemaschine zu verdanken ist. Blättert man das Buch schnell durch, fällt aber auch auf, das die Ränder des *Textes* genau übereinanderliegen. Das liegt daran, dass die bedruckten Bögen vor der Bindung sehr genau ausgerichtet wurden. Letzteres ist keine einfache Angelegenheit: Hat man einen Stapel loser Papierseiten (oder Spielkarten), ist es schwer sie wirklich bündig auszurichten. Auch Gewalt hilft hier nicht viel.

Die technische Lösung für dieses Problem ist ein *Bogenrüttler*. Durch Vibration richtet dieser die Seiten perfekt aus. Abbildung 43.5 zeigt eine solche Maschine für Handbetrieb mit von einer Druckmaschine hergestellten Übungsblättern. Der Drehregler dient dazu, die Vibration von stark auf schwach herunterzuregeln: Simulated Annealing!

Die Übungsblätter kommen aus der Druckmaschine. Die Kanten sind nicht bündig. Der Bogenrüttler schüttelt sie durch und am Ende erhalten wir einen passgenauen Papierstapel.

## Zum Weiterlesen

Simulated Annealing wurde hier recht vereinfacht dargestellt. Obwohl man auch mit dem hier beschriebenen Verfahren sehr gute Resultate erzielen kann, gibt es viele Details, deren Beachtung zu noch besseren Ergebnissen führt. Der Eintrag in Wikipedia[1] ist zur Zeit noch recht knapp. Die englische Fas-

[1] `http://de.wikipedia.org/wiki/Simulierte_Abkühlung`

sung[2] ist aber recht lesenswert. Lehrbücher zu Simulated Annealing sind oft sehr speziell. Daher sei hier zum Weiterlesen ein Buch empfohlen, dass neben Simulated Annealing auch weitere interessante Verfahren zum Lösen schwerer Probleme enthält:

> Juraj Hromkovič: *Algorithmics for Hard Problems (Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics)*, Springer-Verlag, Heidelberg, zweite erweiterte Auflage, 2002.

Im Internet lassen sich viele Applets zur interaktiven Demonstration von Simulated Annealing finden. Zum Beispiel auf der Seite zum 41. Algorithmus der Woche:
`http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo41.php`

[2] `http://en.wikipedia.org/wiki/Simulated_annealing`

# Die Autoren

Prof. Dr. rer. nat. Susanne Albers

Lehrstuhl für Informations- und Kodierungstheorie
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 79, 79110 Freiburg

*Forschungsschwerpunkte:* Entwicklung und Analyse von Algorithmen, Online- und Approximationsalgorithmen, Algorithmen für Probleme in großen Netzwerken, algorithmische Spieltheorie, Algorithm-Engineering

*Buchbeitrag:* Kapitel 39 (Online-Algorithmen)

Prof. Dr. rer. nat. Helmut Alt

Institut für Informatik, Freie Universität Berlin
Takustr. 9, 14195 Berlin

*Forschungsschwerpunkte:* Algorithmen, algorithmische Geometrie, geometrische Methoden der Mustererkennung

*Buchbeiträge:* Kapitel 3 (Schnelle Sortieralgorithmen), Überblick zu Teil III (Planen, strategisches Handeln und Computersimulationen)

Dr. rer. nat. Michael Behrisch

Institut für Informatik, Humboldt-Universität zu Berlin
Rudower Chaussee 25, 12489 Berlin

*Forschungsschwerpunkte:* zufällige Graphen, komplexe Netzwerke, Graphenalgorithmen

*Buchbeitrag:* Kapitel 29 (Die Eulertour)

Dr.-Ing. Rene Beier

D1: Algorithmen und Komplexität, Max-Planck-Institut für Informatik
Campus E1 4, 66123 Saarbrücken

*Forschungsschwerpunkte:* Entwicklung und Analyse von Algorithmen, insbesondere randomisierte Algorithmen und probabilistische Analyse von Algorithmen, kombinatorische Optimierung

*Buchbeitrag:* Kapitel 41 (Das Rucksackproblem)

Prof. Dr. rer. nat. Johannes Blömer

Institut für Informatik, Universität Paderborn
Fürstenallee 11, 33102 Paderborn

*Forschungsschwerpunkte:* Kryptographie, Komplexitätstheorie, Kodierungstheorie

*Buchbeitrag:* Kapitel 17 (Teilen von Geheimnissen)

Prof. Dr. Norbert Blum

Institut für Informatik V, Rheinische Friedrich-Wilhelms-Universität Bonn
Römerstrasse 164, 53117 Bonn

*Forschungsschwerpunkte:* Umsetzung bekannter und Entwicklung neuer Methoden der diskreten Mathematik für konkrete Anwendungen im Hintergrund

*Buchbeitrag:* Kapitel 32 (Dynamische Programmierung)

Dr. rer. nat. Dirk Bongartz

Gymnasium St. Wolfhelm
Turmstr. 2, 41366 Schwalmtal

*Forschungsschwerpunkte:* Didaktik der Informatik, Bioinformatik, Kryptographie

*Buchbeitrag:* Kapitel 16 (Public-Key-Kryptographie)

Prof. Dr. rer. nat. Ulrik Brandes

Fachbereich Informatik & Informationswissenschaft, Universität Konstanz
Fach D 67, 78457 Konstanz

*Forschungsschwerpunkte:* Entwicklung und Analyse von Algorithmen, insbesondere Graphenalgorithmen für die Analyse und Visualisierung von Netzwerken; Soziale Netzwerke; Informationsvisualisierung; Algorithm Engineering

*Buchbeitrag:* Kapitel 10 (PageRank)

Prof. Dr. rer. nat. Dr.h.c. Volker Claus

Institut für Formale Methoden, Universität Stuttgart
Universitätsstr. 38, 70569 Stuttgart

*Forschungsschwerpunkte:* Formale Systeme, Evolutionäre Algorithmen, diskrete Optimierung, Anwendungen im Verkehr, Ausbildung in Informatik

*Buchbeitrag:* Kapitel 37 (Partnerschaftsvermittlung)

Dr. Amin Coja-Oghlan

Institut für Informatik, Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin

*Forschungsschwerpunkt:* Zufall bei Entwurf und Analyse von Algorithmen

*Buchbeitrag:* Kapitel 29 (Die Eulertour)

Prof. Dr. rer. nat. habil. Volker Diekert

Institut für Formale Methoden der Informatik, Universität Stuttgart
Universitätsstr. 38, 70569 Stuttgart

*Forschungsschwerpunkte:* Theorie nebenläufiger Systeme, Temporale Logiken, Automatentheorie und Formale Sprachen, Algebraische Methoden der Informatik

*Buchbeitrag:* Kapitel 37 (Partnerschaftsvermittlung)

Prof. Dr. rer. nat. (USA) Martin Dietzfelbinger

Fachgebiet Komplexitätstheorie und Effiziente Algorithmen
Institut für Theoretische Informatik, Fakultät für Informatik
und Automatisierung, Technische Universität Ilmenau
Helmholtzplatz 1, 98693 Ilmenau

*Forschungsschwerpunkte:* Randomisierte Algorithmen, Datenstrukturen, insbesondere speichereffiziente Speicherung von Daten, Kommunikationskomplexität

*Buchbeiträge:* Kapitel 19 (Fingerprinting), Überblick zu Teil I (Suchen und Sortieren)

Dipl.-Inf. Michael Dom

Lehrstuhl Theoretische Informatik I, Friedrich-Schiller-Universität Jena
Ernst-Abbe-Platz 2, 07743 Jena

*Forschungsschwerpunkte:* NP-schwere Probleme, Matrixprobleme, Festparameteralgorithmen

*Buchbeitrag:* Kapitel 7 (Tiefensuche)

Dipl.-Math. Gabi Dorfmüller

Fachbereich Informatik & Informationswissenschaft, Universität Konstanz
Fach D 67, 78457 Konstanz

*Forschungsschwerpunkt:* Graphenalgorithmen

*Buchbeitrag:* Kapitel 10 (PageRank)

Dipl.-Inform. Arno Eigenwillig

D1: Algorithmen und Komplexität, Max-Planck-Institut für Informatik
Campus E1 4, 66123 Saarbrücken

*Forschungsschwerpunkte:* Exakte algorithmische Geometrie

*Buchbeitrag:* Kapitel 11 (Multiplikation langer Zahlen)

Prof. Dr.-Ing. Friedrich Eisenbrand

Lehrstuhl für diskrete Optimierung, Universität Paderborn
Warburgerstr. 100, 33098 Paderborn

*Forschungsschwerpunkte:* Entwicklung und Analyse von Algorithmen, kombinatorische Optimierung, ganzzahlige lineare Programmierung und angewandte Optimierung

*Buchbeitrag:* Kapitel 12 (Der Euklidische Algorithmus)

Dipl.-Inform. Jost Enderle

Lehrstuhl für Informatik 9 (Datenmanagement und -exploration)
RWTH Aachen
Ahornstr. 55, 52074 Aachen

*Forschungsschwerpunkte:* Indexstrukturen, effiziente Datenbanktechnologien

*Buchbeitrag:* Kapitel 1 (Binäre Suche)

Prof. Dr. rer. nat. Thomas Erlebach

Department of Computer Science, University of Leicester
University Road, Leicester, LE1 7RH, England

*Forschungsschwerpunkte:* Approximations- und Online-Algorithmen für kombinatorische Optimierungsprobleme, algorithmische Aspekte von Kommunikationsnetzen, algorithmische Graphentheorie

*Buchbeitrag:* Kapitel 24 (Mehrheitsbestimmung)

Dipl.-Inf. Christoph Freundl

Lehrstuhl für Informatik 10 (Systemsimulation)
Friedrich-Alexander-Universität Erlangen-Nürnberg
Cauerstr. 6, 91058 Erlangen

*Forschungsschwerpunkt:* Programmiertechniken für Höchstleistungsrechner

*Buchbeitrag:* Kapitel 31 (Gauß-Seidel Iteration zur Berechnung physikalischer Probleme)

Dipl.-Inform. Joachim Gehweiler

Heinz Nixdorf Institut und Institut für Informatik, Universität Paderborn
Fürstenallee 11, 33102 Paderborn

*Forschungsschwerpunkte:* Paralleles Rechnen, insbes. Web Computing, Lastbalancierung

*Buchbeitrag:* Kapitel 40 (Bin Packing)

Dipl.-Math.-tech. Robert Görke

Institut für Theoretische Informatik, Universität Karlsruhe (TH)
Am Fasanengarten 5, 76131 Karlsruhe

*Forschungsschwerpunkte:* Algorithmen zum Clustern von statischen und dynamischen Graphen, Analytische Visualisierungen großer Graphen

*Buchbeitrag:* Kapitel 36 (Maximale Flüsse)

Dipl.-Inf. Markus Hinkelmann

Institut für Theoretische Informatik, Universität zu Lübeck
Ratzeburger Allee 160, 23538 Lübeck

*Forschungsschwerpunkte:* Privacy, Kryptologie

*Buchbeitrag:* Kapitel 14 (Einwegfunktionen)

Dr.-Ing. Hagen Höpfner

School of Information Technology, International University in Germany
Campus 3, 76646 Bruchsal

*Forschungsschwerpunkte:* mobile Datenbanken und Informationssysteme, kontextbewusste Informationssysteme, Datenbanktheorie, Anfrageindizierung

*Buchbeitrag:* Kapitel 5 (Topologisches Sortieren)

Dr. rer. nat. Falk Hüffner

Lehrstuhl Theoretische Informatik I, Friedrich-Schiller-Universität Jena
Ernst-Abbe-Platz 2, 07743 Jena

*Forschungsschwerpunkte:* NP-schwere Probleme, Graphprobleme, Festparameteralgorithmen, Bioinformatik

*Buchbeitrag:* Kapitel 7 (Tiefensuche)

Tim Jonischkat

Institut für Informatik und Wirtschaftsinformatik
Universität Duisburg-Essen (Campus Essen)
Schützenbahn 70, 45117 Essen

*Forschungsschwerpunkte:* Software Engineering, insbesondere Softwareproduktlinienentwicklung, Qualitätssicherung und Requirements Engineering

*Buchbeitrag:* Kapitel 25 (Zufallszahlen)

Dr. rer. nat. Tom Kamphans

IBR Abteilung Algorithmik, Technische Universität Braunschweig
Mühlenpfordtstr. 23, 38106 Braunschweig

*Forschungsschwerpunkte:* Bewegungsplanung für Roboter, Online Algorithmen, Algorithmische Geometrie

*Buchbeitrag:* Kapitel 8 (Der Pledge-Algorithmus)

Prof. Dr. rer. nat. Rolf Klein

Lehrstuhl für Informatik I, Universität Bonn
Römerstr. 164, 53117 Bonn

*Forschungsschwerpunkte:* Algorithmische Geometrie, Online Algorithmen, Algorithmen und Datenstrukturen

*Buchbeitrag:* Kapitel 8 (Der Pledge-Algorithmus)

Juniorprof. Dr. rer. nat. Sigrid Knust

Institut für Informatik, Universität Osnabrück
Albrechtstr. 28, 49069 Osnabrück

*Forschungsschwerpunkte:* kombinatorische Optimierung, Scheduling, Timetabling

*Buchbeitrag:* Kapitel 27 (Turnier- und Sportligaplanung)

Prof. Dr. rer. nat. Leif Kobbelt

Lehrstuhl für Informatik 8 (Computergraphik und Multimedia)
RWTH Aachen
Ahornstr. 55, 52074 Aachen

*Forschungsschwerpunkte:* Computergraphik, Geometrieverarbeitung, CAD/CAM, Visual Computing & Simulation, Computer Vision, Bildsynthese, Bild- und Videoverarbeitung, verteilte Multimedia Applikationen

*Buchbeitrag:* Kapitel 30 (Kreise zeichnen mit Turbo)

Assistant Professor Jochen Könemann, Ph.D.

Department of Combinatorics and Optimization, University of Waterloo
200 University Avenue West, Waterloo, ON, N2L 3G1, Kanada

*Forschungsschwerpunkte:* Entwicklung und Analyse von (Approximations-) Algorithmen, kombinatorische Optimierung, algorithmische Spieltheorie

*Buchbeitrag:* Kapitel 26 (Gewinnstrategie für ein Streichholzspiel)

Prof. Dr. rer. nat. habil. Wolfgang P. Kowalk

Department für Informatik (Rechnernetze)
Carl-von-Ossietzky-Universität Oldenburg
Ammerländer Heerstr. 114–118, 26121 Oldenburg

*Forschungsschwerpunkte:* Rechnernetze, Fehlererkennung und Sicherheit, Algorithmen, Simulation, Integration, Programmierung, 3D-Modellierung

*Buchbeitrag:* Kapitel 2 (Sortieren durch Einfügen)

Matthias Kretschmer

Institut für Informatik V, Rheinische Friedrich-Wilhelms-Universität Bonn
Römerstr. 164, 53117 Bonn

*Forschungsschwerpunkte:* Algorithmen, Spieltheorie

*Buchbeitrag:* Kapitel 32 (Dynamische Programmierung)

Dipl.-Inf. Peter Liske

Institut für Informatik, Humboldt-Universität zu Berlin
Rudower Chaussee 25, 12489 Berlin

*Forschungsschwerpunkte:* Algorithmen und Komplexitätstheorie

*Buchbeitrag:* Kapitel 29 (Die Eulertour)

Dr. rer. nat. Ulf Lorenz

Fachbereich Mathematik, Fachgebiet Diskrete Optimierung
Technische Universität Darmstadt
Schlossgartenstr. 7, 64289 Darmstadt

*Forschungsschwerpunkte:* Optimierung in dynamischer Umgebung, Parallele Systeme

*Buchbeitrag:* Kapitel 28 (Der Alphabeta-Algorithmus für Spielbäume)

Dipl.-Math. Steffen Mecke

Institut für Theoretische Informatik, Universität Karlsruhe (TH)
Am Fasanengarten 5, 76131 Karlsruhe

*Forschungsschwerpunkt:* Algorithmen für und Analyse von Sensor- und Ad-hoc-Netzwerken

*Buchbeitrag:* Kapitel 36 (Maximale Flüsse)

Prof. Dr. Dr. Ing. E.h. Kurt Mehlhorn

D1: Algorithmen und Komplexität, Max-Planck-Institut für Informatik
Campus E1 4, 66123 Saarbrücken

*Forschungsschwerpunkte:* effiziente Algorithmen, Datenstrukturen, Graphenalgorithmen, Kombinatorische Optimierung, Algorithmische Geometrie, Algorithm Engineering, Softwarebibliotheken

*Buchbeitrag:* Kapitel 11 (Multiplikation langer Zahlen)

Prof. Dr. math. Friedhelm Meyer auf der Heide

Heinz Nixdorf Institut und Institut für Informatik, Universität Paderborn
Fürstenallee 11, 33102 Paderborn

*Forschungsschwerpunkte:* Algorithmen und Komplexität

*Buchbeitrag:* Kapitel 40 (Bin Packing)

Prof. Dr. rer. nat. Rolf Möhring

Institut für Mathematik, Technische Universität Berlin
Straße des 17. Juni 136, 10623 Berlin

*Forschungsschwerpunkte:* Graphenalgorithmen, Kombinatorische Optimierung, Scheduling, Industrielle Anwendungen

*Buchbeitrag:* Kapitel 13 (Das Sieb des Eratosthenes)

Prof. Dr. rer. nat. Burkhard Monien

Universität Paderborn, Fakultät EIM-I
Fürstenallee 11, 33102 Paderborn

*Forschungsschwerpunkte:* Entwicklung und Analyse von Algorithmen, insbesondere Algorithmen zur effizienten Nutzung von Parallelrechnern, algorithmische Spieltheorie

*Buchbeitrag:* Kapitel 28 (Der Alphabeta-Algorithmus für Spielbäume)

Prof. Dr. rer. nat. Bruno Müller-Clostermann

Lehrstuhl für Praktische Informatik (Systemmodellierung)
Institut für Informatik und Wirtschaftsinformatik
Universität Duisburg-Essen (Campus Essen)
Schützenbahn 70, 45117 Essen

*Forschungsschwerpunkte:* Stochastische Modelle, Simulation, Leistungsbewertung und Kapazitätsplanung von Rechensystemen und Rechnernetzen

*Buchbeitrag:* Kapitel 25 (Zufallszahlen)

Prof. Dr. Stefan Näher

Fachbereich IV – Informatik, Universität Trier
Campus II, Behringstr. 21, 54296 Trier

*Forschungsschwerpunkte:* Algorithmen und Datenstrukturen, Graphalgorithmen, Algorithmische Geometrie, Algorithm Engineering

*Buchbeitrag:* Kapitel 42 (Das Travelling Salesman Problem)

Prof. Dr. phil. nat. Markus E. Nebel

Fachbereich Informatik, AG Algorithmen und Komplexität
TU Kaiserslautern
Gottlieb Daimler Straße 48, 67663 Kaiserslautern

*Forschungsschwerpunkte:* Entwurf und Analyse von Algorithmen und Datenstrukturen, insbesondere Average-Case Analyse; Algorithm Engineering; Algorithmen der Bioinformatik

*Buchbeitrag:* Kapitel 6 (Texte durchsuchen – aber schnell!)

Prof. Dr. rer. nat. Rolf Niedermeier

Lehrstuhl für Theoretische Informatik I/Komplexitätstheorie
Institut für Informatik, Friedrich-Schiller-Universität Jena
Ernst-Abbe-Platz 2, 07743 Jena

*Forschungsschwerpunkte:* NP-schwere Probleme, algorithmische Graphtheorie, algorithmische Bioinformatik, parametrisierte Algorithmik, diskrete Optimierungsprobleme

*Buchbeitrag:* Kapitel 7 (Tiefensuche)

Dipl.-Math. Martin Oellrich

Arbeitsgruppe Kombinatorische Optimierung und Graphenalgorithmen
Institut für Mathematik, TU Berlin
Str. des 17. Juni 136, 10623 Berlin

*Forschungsschwerpunkte:* algorithmische Graphentheorie, kombinatorische Optimierung, Datenstrukturen und Algorithmen.

*Buchbeitrag:* Kapitel 13 (Das Sieb des Eratosthenes)

Dr. rer. nat. habil. Holger Petersen

FMI, Universität Stuttgart
Universitätsstr. 38, 70569 Stuttgart

*Forschungsschwerpunkte:* Algorithmen und Datenstrukturen, konkrete Komplexität

*Buchbeitrag:* Kapitel 37 (Partnerschaftsvermittlung)

Prof. Dr. math. Rüdiger Reischuk

Institut für Theoretische Informatik, Universität zu Lübeck
Ratzeburger Allee 160, 23538 Lübeck

*Forschungsschwerpunkte:* Algorithmische Komplexität, Parallelverarbeitung, Fehlertoleranz, Kryptologie und Steganographie, Algorithmisches Lernen

*Buchbeiträge:* Kapitel 14 (Einwegfunktionen), Überblick zu Teil III (Planen, strategisches Handeln und Computersimulationen)

Prof. Dr. rer. nat. Peter Rossmanith

Lehr- und Forschungsgebiet Theoretische Informatik, RWTH Aachen
Ahornstr. 55, 52074 Aachen

*Forschungsschwerpunkt:* Effiziente Algorithmen

*Buchbeitrag:* Kapitel 43 (Simulated Annealing)

Prof. Dr. rer. nat. Ulrich Rüde

Lehrstuhl für Informatik 10 (Systemsimulation)
Friedrich-Alexander-Universität Erlangen-Nürnberg
Cauerstr. 6, 91058 Erlangen

*Forschungsschwerpunkte:* Höchstleistungsrechnen, Multilevel-Verfahren, Lattice-Boltzmann-Methoden, Simulation in Naturwissenschaft und Technik

*Buchbeitrag:* Kapitel 31 (Gauß-Seidel Iteration zur Berechnung physikalischer Probleme)

Prof. Dr. rer. nat. Peter Sanders

Institut für Theoretische Informatik, Universität Karlsruhe (TH)
Am Fasenengarten 5, 76128 Karlsruhe

*Forschungsschwerpunkte:* Algorithmik, insbesondere Algorithm Engineering, Graphenalgorithmen, parallele Algorithmen, Algorithmen für große Datenmengen, randomisierte Algorithmen

*Buchbeitrag:* Kapitel 34 (Kürzeste Wege)

Prof. Dr. rer. nat. Christian Scheideler

Lehrstuhl für Informatik 14 (Effiziente Algorithmen)
Technische Universität München
Boltzmannstr. 3, 85748 Garching

*Forschungsschwerpunkte:* Verteilte Algorithmen und Datenstrukturen, Sicherheit in verteilten Systemen, randomisierte Algorithmen und stochastische Prozesse, diskrete Mathematik

*Buchbeiträge:* Kapitel 22 (Broadcasting), Überblick zu Teil I (Suchen und Sortieren)

Prof. Dr. rer. nat. Christian Schindelhauer

Rechnernetze und Telematik, Institut für Informatik
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 51, 79110 Freiburg im Breisgau

*Forschungsschwerpunkte:* Peer-to-Peer-Netzwerke, Mobile Ad-hoc-Netzwerke, Storage-Area-Netzwerke, Drahtlose Sensor-Netze, Algorithmen für Computer-Netzwerke und Netzwerkprotokolle

*Buchbeitrag:* Kapitel 20 (Hashing)

Prof. Dr. Holger Schlingloff

Institut für Informatik, Humboldt Universität zu Berlin
Rudower Chaussee 25, 12489 Berlin

Fraunhofer Institut für Rechnerarchitektur und Softwaretechnik FIRST
Kekulestr. 7, 12489 Berlin

*Forschungsschwerpunkte:* Spezifikation, Verifikation und Test eingebetteter Software

*Buchbeitrag:* Kapitel 9 (Zyklensuche in Graphen)

Swen Schmelzer

Lehrstuhl für Informations- und Kodierungstheorie
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 79, 79110 Freiburg

*Forschungsschwerpunkt:* Online-Algorithmen

*Buchbeitrag:* Kapitel 39 (Online-Algorithmen)

Priv.-Doz. Dr. rer. nat. habil. Lothar Schmitz

Institut für Softwaretechnologie der Fakultät für Informatik, UniBw München
85577 Neubiberg

*Forschungsschwerpunkte:* Grundlagen und Werkzeuge des Compilerbaus, Programmiermethodik, insbesondere formale Programmentwicklung und Wiederverwendung, Langzeitarchivierung

*Buchbeitrag:* Kapitel 23 (Zahlen auf Deutsch aussprechen)

Prof. Raimund Seidel

Fachrichtung Informatik, Universität des Saarlandes
Campus E1 3, 66123 Saarbrücken

*Forschungsschwerpunkte:* Algorithmen, Datenstrukturen, Algorithmische und Kombinatorische Geometrie, Randomisierung

*Buchbeitrag:* Kapitel 33 (Faires Teilen)

Prof. Dr. rer. nat. Thomas Seidl

Lehrstuhl für Informatik 9 (Datenmanagement und -exploration)
RWTH Aachen
Ahornstr. 55, 52074 Aachen

*Forschungsschwerpunkte:* Data Mining, inhaltsbasierte Ähnlichkeitssuche, Multimedia-Datenbanken, Indexstrukturen, effiziente Datenbanktechnologien

*Buchbeitrag:* Kapitel 1 (Binäre Suche)

Dipl.-Inform. Dominik Sibbing

Lehrstuhl für Informatik 8 (Computergraphik und Multimedia)
RWTH Aachen
Ahornstr. 55, 52074 Aachen

*Forschungsschwerpunkte:* Computergraphik, Geometrieverarbeitung, Computer Vision, Animation und Synthese von Gesichtsausdrücken

*Buchbeitrag:* Kapitel 30 (Kreise zeichnen mit Turbo)

Priv.-Doz. Dr. Detlef Sieling

Lehrstuhl Informatik 2 (Effiziente Algorithmen und Komplexitätstheorie)
Technische Universität Dortmund
Otto-Hahn-Str. 14, 44221 Dortmund

*Forschungsschwerpunkte:* Effiziente Algorithmen und Komplexitätstheorie, insbesondere in den Bereichen Binary Decision Diagrams und Quantenrechner

*Buchbeitrag:* Kapitel 18 (Poker per E-Mail)

Dipl.-Inform. Johannes Singler

Institut für Theoretische Informatik, Universität Karlsruhe (TH)
Kaiserstr. 12, 76128 Karlsruhe

*Forschungsschwerpunkte:* Parallele Algorithmen, Algorithmen-Bibliotheken

*Buchbeitrag:* Kapitel 34 (Kürzeste Wege)

Dr. rer. nat. Katharina Skutella

Institut für Mathematik, Technische Universität Berlin
Straße des 17. Juni 136, 10623 Berlin

*Forschungsschwerpunkte:* Kombinatorische Optimierung, Schnittstelle Schule/Universität

*Buchbeitrag:* Kapitel 35 (Minimale aufspannende Bäume)

Prof. Dr. rer. nat. Martin Skutella

Institut für Mathematik, Technische Universität Berlin
Straße des 17. Juni 136, 10623 Berlin

*Forschungsschwerpunkte:* Kombinatorische Optimierung, Netzwerkflüsse, Scheduling, Approximationsalgorithmen

*Buchbeitrag:* Kapitel 35 (Minimale aufspannende Bäume)

Dr. sc. Alexander Souza

Institut für Informatik, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 79, 79110 Freiburg im Breisgau

*Forschungsschwerpunkte:* Average-Case-Analyse, Online- und Approximations-Algorithmen und algorithmische Spieltheorie

*Buchbeitrag:* Kapitel 21 (Fehlererkennende Codes)

Prof. Angelika Steger

Institut für Theoretische Informatik, ETH Zürich
Universitätstrasse 6, 8092 Zürich

*Forschungsschwerpunkte:* Graphentheorie, Randomisierte Algorithmen, Probabilistische Methoden, Kombinatorische Optimierung, Approximationsalgorithmen

*Buchbeitrag:* Kapitel 21 (Fehlererkennende Codes)

Prof. Dr. rer. nat. Till Tantau

Institut für Theoretische Informatik, Universität zu Lübeck
Ratzeburger Allee 160, 23538 Lübeck

*Forschungsschwerpunkte:* Komplexitätstheorie, Bioinformatik, Rekursionstheorie

*Buchbeitrag:* Kapitel 15 (Der One-Time-Pad-Algorithmus)

Priv.-Doz. Dr. rer. nat. Walter Unger

Lehrstuhl für Informatik 1 (Algorithmen und Komplexität), RWTH Aachen
Ahornstr. 55, 52074 Aachen

*Forschungsschwerpunkte:* Entwicklung und Analyse von Algorithmen, Komplexitätstheorie und Graphentheorie, kombinatorische Optimierung, Algorithmen für Computernetze

*Buchbeitrag:* Kapitel 16 (Public-Key-Kryptographie)

Prof. Dr. rer. nat. Berthold Vöcking

Lehrstuhl für Informatik 1 (Algorithmen und Komplexität), RWTH Aachen
Ahornstr. 55, 52074 Aachen

*Forschungsschwerpunkte:* Entwicklung und Analyse von Algorithmen, insbesondere randomisierte Algorithmen und probabilistische Analyse von Algorithmen, kombinatorische Optimierung, Algorithmen für Computernetze, algorithmische Spieltheorie

*Buchbeiträge:* Kapitel 41 (Das Rucksackproblem), Überblick zu Teil II (Rechnen, Verschlüsseln und Codieren)

Prof. Dr. rer. nat. Heribert Vollmer

Institut für Theoretische Informatik, Leibniz Universität Hannover
Appelstr. 4, 30167 Hannover

*Forschungsschwerpunkte:* Komplexitätstheorie, Schaltkreiskomplexität, Komplexität von Constraint-Satisfaction-Problemen, Komplexität von Erfüllbarkeitsproblemen

*Buchbeitrag:* Überblick zu Teil IV (Optimieren)

Prof. Dr. rer. nat. Dorothea Wagner

Institut für Theoretische Informatik, Universität Karlsruhe (TH)
Am Fasanengarten 5, 76131 Karlsruhe

*Forschungsschwerpunkte:* Graphenalgorithmen, Visualisierung, Clustern, Experimentelle Algorithmik

*Buchbeiträge:* Kapitel 36 (Maximale Flüsse),
Überblick zu Teil IV (Optimieren)

Prof. Dr. rer. nat. Rolf Wanka

Lehrstuhl für Informatik 12 (Hardware-Software-Co-Design)
Department Informatik, Universität Erlangen-Nürnberg
Am Weichselgarten 3, 91058 Erlangen

*Forschungsschwerpunkte:* Effiziente Algorithmen und kombinatorische Optimierung

*Buchbeitrag:* Kapitel 4 (Paralleles Sortieren)

Daniel Warner

Institut für Informatik, Universität Paderborn
Fürstenallee 11, 33102 Paderborn

*Forschungsschwerpunkte:* Effiziente Algorithmen und Komplexitätstheorie, Online-Algorithmen

*Buchbeitrag:* Kapitel 28 (Der Alphabeta-Algorithmus für Spielbäume)

Prof. Dr. techn. Emo Welzl

Institut für Theoretische Informatik, ETH Zürich
Universitätstrasse 6, 8092 Zürich, Schweiz

*Forschungsschwerpunkte:* Algorithmen und Kombinatorik

*Buchbeitrag:* Kapitel 38 (Kleinster umschließender Kreis)