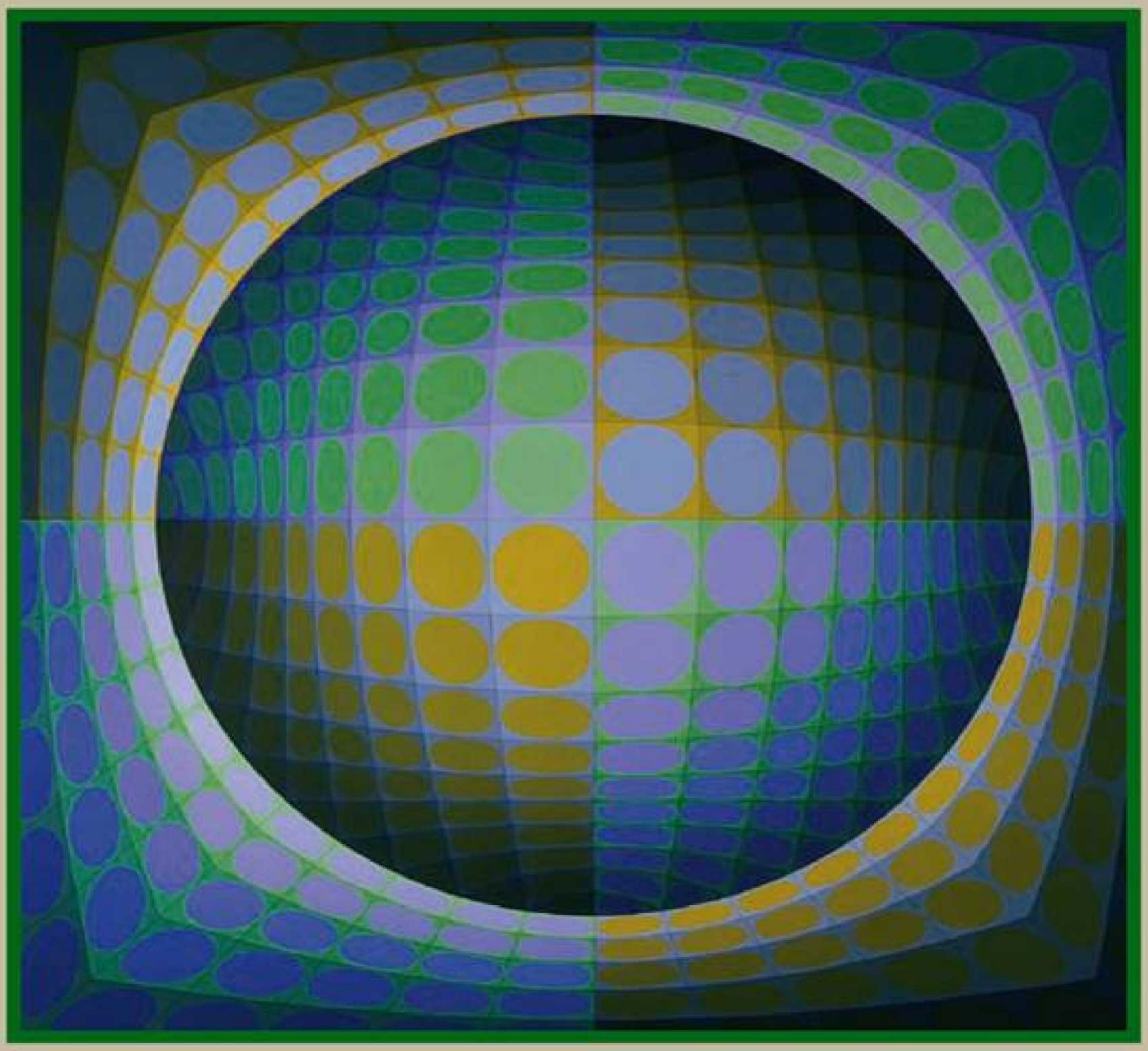
ALGORITHMS of Informatics



volume 1

ALGORITHMS

OF INFORMATICS

Volume 1

FOUNDATIONS

mondAt Kiadó

Budapest, 2007

The book appeared with the support of the
Department of Mathematics of Hungarian Academy of Science

Editor: Antal Iványi

Authors: Zoltán Kása (Chapter 1), Zoltán Csörnyei (2), Ulrich Tamm (3), Péter Gács (4), Gábor Ivanyos, Lajos Rónyai (5), Antal Járai, Attila Kovács (6), Jörg Rothe (7, 8), Csanád Imreh (9), Ferenc Szidarovszky (10), Zoltán Kása (11), Aurél Galántai, András Jeney (12), István Miklós (13), László Szirmay-Kalos (14), Ingo Althöfer, Stefan Schwarz (15), Burkhard Englert, Dariusz Kowalski, Grzegorz Malewicz, Alexander Allister Shvartsman (16), Tibor Gyires (17), Antal Iványi, Claudia Leopold (18), Eberhard Zehendner (19), Ádám Balog, Antal Iványi (20), János Demetrovics, Attila Sali (21, 22), Attila Kiss (23)

Validators: Zoltán Fülöp (1), Pál Dömösi (2), Sándor Fridli (3), Anna Gál (4), Attila Pethő (5), Lajos Rónyai (6), János Gonda (7), Gábor Ivanyos (8), Béla Vizvári (9), János Mayer (10), András Recski (11), Tamás Szántai (12), István Katsányi (13), János Vida (14), Tamás Szántai (15), István Majzik (16), János Sztrik (17), Dezső Sima (18, 19), László Varga (20), Attila Kiss (21, 22), András Benczúr (23)

Linguistical validators: Anikó Hörmann and Veronika Vöröss

Translators: Csaba Schneider (1), Milós Péter Pintér (10), László Orosz (14), Veronika Vöröss (17), Anikó Hörmann (23)

Cover art: Victor Vasarely, *Dirac*, 1978. With the permission of Museum of Fine Arts, Budapest. The used film is due to GOMA ZRt. Cover design by Antal Iványi

© Ingo Althöfer, Viktor Belényesi, Zoltán Csörnyei, János Demetrovics, Pál Dömösi, Burkhard Englert, Péter Gács, Aurél Galántai, Anna Gál, János Gonda, Tibor Gyires, Anikó Hörmann, Csanád Imreh, Anna Iványi, Antal Iványi, Gábor Ivanyos, Antal Járai, András Jeney, Zoltán Kása, István Katsányi, Attila Kiss, Attila Kovács, Dariusz Kowalski, Claudia Leopold, Kornél Locher, Gregorz Malewicz, János Mayer, István Miklós, Attila Pethő, András Recski, Lajos Rónyai, Jörg Rothe, Attila Sali, Stefan Schwarz, Alexander Allister Shvartsman, Dezső Sima, Tamás Szántai, Ferenc Szidarovszky, László Szirmay-Kalos, János Sztrik, Ulrich Tamm, László Varga, János Vida, Béla Vizvári, Veronika Vöröss, Eberhard Zehendner, 2007

ISBN of Volume 1: 978-963-87596-1-0;
ISBN of Volume 1 and Volume 2: 978-963-87596-0-3 Ö

Published by mondAt Kiadó
H-1158 Budapest, Jánoshida u. 18. Telephone/facsimile: +36 1 418-0062
Internet: http://www.mondat.hu/, E-mail: mondat@mondat.hu
Responsible publisher: ifj. László Nagy

Printed and bound by
mondAt Kft, Budapest

Contents

Preface . . . 8
Introduction . . . 9

I. AUTOMATA . . . 12
1. Automata and Formal Languages (Zoltán Kása) . . . 13
1.1. Languages and grammars . . . 13
1.1.1. Operations on languages . . . 14
1.1.2. Specifying languages . . . 14
1.1.3. Chomsky hierarchy of grammars and languages . . . 18
1.1.4. Extended grammars . . . 22
1.1.5. Closure properties in the Chomsky-classes . . . 24
1.2. Finite automata and regular languages . . . 26
1.2.1. Transforming nondeterministic finite automata in deterministic finite automata . . . 31
1.2.2. Equivalence of deterministic finite automata . . . 34
1.2.3. Equivalence of finite automata and regular languages . . . 36
1.2.4. Finite automata with empty input . . . 41
1.2.5. Minimization of finite automata . . . 45
1.2.6. Pumping lemma for regular languages . . . 47
1.2.7. Regular expressions . . . 50
1.3. Pushdown automata and context-free languages . . . 60
1.3.1. Pushdown automata . . . 60
1.3.2. Context-free languages . . . 69
1.3.3. Pumping lemma for context-free languages . . . 71
1.3.4. Normal forms of the context-free languages . . . 73
2. Compilers (Zoltán Csörnyei) . . . 80
2.1. The structure of compilers . . . 81
2.2. Lexical analysis . . . 84
2.2.1. The automaton of the scanner . . . 85
2.2.2. Special problems . . . 89
2.3. Syntactic analysis . . . 92
2.3.1. $LL(1)$ parser . . . 94

2.3.2. $LR(1)$ parsing . . . 109
3. Compression and Decompression (Ulrich Tamm) . . . 131
3.1. Facts from information theory . . . 132
3.1.1. The Discrete Memoryless Source . . . 132
3.1.2. Prefix codes . . . 133
3.1.3. Kraft's inequality and noiseless coding theorem . . . 135
3.1.4. Shannon-Fano-Elias-codes and the Shannon-Fano-algorithm 137
3.1.5. The Huffman coding algorithm . . . 139
3.2. Arithmetic coding and modelling . . . 141
3.2.1. Arithmetic coding . . . 142
3.2.2. Modelling . . . 146
3.3. Ziv-Lempel-coding . . . 153
3.3.1. LZ77 . . . 153
3.3.2. LZ78 . . . 154
3.4. The Burrows-Wheeler-transform . . . 155
3.5. Image compression . . . 160
3.5.1. Representation of data . . . 160
3.5.2. The discrete cosine transform . . . 161
3.5.3. Quantisation . . . 162
3.5.4. Coding . . . 163
4. Reliable computation (Péter Gács) . . . 169
4.1. Probability theory . . . 170
4.1.1. Terminology . . . 171
4.1.2. The law of large numbers (with "large deviations") . . . 172
4.2. Logic circuits . . . 174
4.2.1. Boolean functions and expressions . . . 174
4.2.2. Circuits . . . 176
4.2.3. Fast addition by a Boolean circuit . . . 178
4.3. Expensive fault-tolerance in Boolean circuits . . . 180
4.4. Safeguarding intermediate results . . . 184
4.4.1. Cables . . . 184
4.4.2. Compressors . . . 185
4.4.3. Propagating safety . . . 188
4.4.4. Endgame . . . 189
4.4.5. The construction of compressors . . . 191
4.5. The reliable storage problem . . . 194
4.5.1. Clocked circuits . . . 194
4.5.2. Storage . . . 197
4.5.3. Error-correcting codes . . . 198
4.5.4. Refreshers . . . 202

II. COMPUTER ALGEBRA . . . 216
5. Algebra (Gábor Ivanyos, Lajos Rónyai) . . . 217
5.1. Fields, vector spaces, and polynomials . . . 217
5.1.1. Ring theoretic concepts . . . 217

5.1.2. Polynomials . . . 221
5.2. Finite fields . . . 230
5.3. Factoring polynomials over finite fields . . . 237
5.3.1. Square-free factorisation . . . 237
5.3.2. Distinct degree factorisation . . . 239
5.3.3. The Cantor-Zassenhaus algorithm . . . 241
5.3.4. Berlekamp's algorithm . . . 242
5.4. Lattice reduction . . . 248
5.4.1. Lattices . . . 248
5.4.2. Short lattice vectors . . . 251
5.4.3. Gauss' algorithm for two-dimensional lattices . . . 252
5.4.4. A Gram-Schmidt orthogonalisation and weak reduction . . . 254
5.4.5. Lovász-reduction . . . 256
5.4.6. Properties of reduced bases . . . 257
5.5. Factoring polynomials in $\mathbb{Q}[x]$ . . . 259
5.5.1. Preparations . . . 260
5.5.2. The Berlekamp-Zassenhaus algorithm . . . 266
5.5.3. The LLL algorithm . . . 268
6. Computer Algebra (Antal Járai, Attila Kovács) . . . 275
6.1. Data representation . . . 276
6.2. Common roots of polynomials . . . 281
6.2.1. Classical and extended Euclidean algorithm . . . 281
6.2.2. Primitive Euclidean algorithm . . . 287
6.2.3. The resultant . . . 289
6.2.4. Modular greatest common divisor . . . 296
6.3. Gröbner basis . . . 300
6.3.1. Monomial order . . . 301
6.3.2. Multivariate division with remainder . . . 303
6.3.3. Monomial ideals and Hilbert's basis theorem . . . 304
6.3.4. Buchberger's algorithm . . . 305
6.3.5. Reduced Gröbner basis . . . 307
6.3.6. The complexity of computing Gröbner bases . . . 307
6.4. Symbolic integration . . . 309
6.4.1. Integration of rational functions . . . 310
6.4.2. The Risch integration algorithm . . . 315
6.5. Theory and practice . . . 326
6.5.1. Other symbolic algorithms . . . 327
6.5.2. An overview of computer algebra systems . . . 328
7. Cryptology (Jörg Rothe) . . . 332
7.1. Foundations . . . 333
7.1.1. Cryptography . . . 334
7.1.2. Cryptanalysis . . . 338
7.1.3. Algebra, number theory, and graph theory . . . 339
7.2. Diffie and Hellman's secret-key agreement protocol . . . 346
7.3. RSA and factoring . . . 349

7.3.1. RSA . . . 349
7.3.2. Digital RSA signatures . . . 353
7.3.3. Security of RSA . . . 353
7.4. The protocols of Rivest, Rabi, and Sherman . . . 355
7.5. Interactive proof systems and zero-knowledge . . . 356
7.5.1. Interactive proof systems, Arthur-Merlin games, and zero-knowledge protocols . . . 356
7.5.2. Zero-knowledge protocol for graph isomorphism . . . 359
8. Complexity Theory (Jörg Rothe) . . . 364
8.1. Foundations . . . 365
8.2. NP-completeness . . . 371
8.3. Algorithms for the satisfiability problem . . . 373
8.3.1. A deterministic algorithm . . . 373
8.3.2. A randomised algorithm . . . 375
8.4. Graph isomorphism and lowness . . . 378
8.4.1. Reducibilities and complexity hierarchies . . . 378
8.4.2. Graph isomorphism is in the low hierarchy . . . 383
8.4.3. Graph isomorphism is in SPP . . . 386

III. NUMERICAL METHODS . . . 394
9. Competitive Analysis (Imreh Csanád) . . . 395
9.1. Notions, definitions . . . 395
9.2. The k-server problem . . . 397
9.3. Models related to computer networks . . . 403
9.3.1. The data acknowledgement problem . . . 403
9.3.2. The file caching problem . . . 405
9.3.3. On-line routing . . . 408
9.4. On-line bin packing models . . . 412
9.4.1. On-line bin packing . . . 412
9.4.2. Multidimensional models . . . 416
9.5. On-line scheduling . . . 419
9.5.1. On-line scheduling models . . . 419
9.5.2. LIST model . . . 420
9.5.3. TIME model . . . 425
10. Game Theory (Ferenc Szidarovszky) . . . 429
10.1. Finite games . . . 430
10.1.1. Enumeration . . . 431
10.1.2. Games represented by finite trees . . . 433
10.2. Continuous games . . . 437
10.2.1. Fixed-point methods based on best responses . . . 437
10.2.2. Applying Fan's inequality . . . 438
10.2.3. Solving the Kuhn-Tucker conditions . . . 440
10.2.4. Reduction to optimization problems . . . 441
10.2.5. Method of fictitious play . . . 449
10.2.6. Symmetric matrix games . . . 450

10.2.7. Linear programming and matrix games . . . 452
10.2.8. The method of von Neumann . . . 454
10.2.9. Diagonally strictly concave games . . . 457
10.3. The oligopoly problem . . . 465
11. Recurrences (Zoltán Kása) . . . 478
11.1. Linear recurrence equations . . . 479
11.1.1. Linear homogeneous equations with constant coefficients . . . 479
11.1.2. Linear nonhomogeneous recurrence equations with constant coefficients . . . 484
11.2. Generating functions and recurrence equations . . . 486
11.2.1. Definition and operations . . . 486
11.2.2. Solving recurrence equations by generating functions . . . 490
11.2.3. The Z-transform method . . . 497
11.3. Numerical solution . . . 500
12. Scientific Computations (Aurél Galántai, András Jeney) . . . 502
12.1. Floating point arithmetic and error analysis . . . 502
12.1.1. Classical error analysis . . . 502
12.1.2. Forward and backward errors . . . 504
12.1.3. Rounding errors and floating point arithmetic . . . 505
12.1.4. The floating point arithmetic standard . . . 510
12.2. Linear systems of equations . . . 512
12.2.1. Direct methods for solving linear systems . . . 512
12.2.2. Iterative methods for linear systems . . . 523
12.2.3. Error analysis of linear algebraic systems . . . 525
12.3. Eigenvalue problems . . . 534
12.3.1. Iterative solutions of the eigenvalue problem . . . 537
12.4. Numerical program libraries and software tools . . . 543
12.4.1. Standard linear algebra subroutines . . . 544
12.4.2. Mathematical software . . . 547
Bibliography . . . 552
Subject Index . . . 563
Name Index . . . 572

Preface

It is a special pleasure for me to recommend to the Readers the book *Algorithms of Computer Science,* edited with great care by Antal Iványi. Computer algorithms form a very important and fast developing branch of computer science. Design and analysis of large computer networks, large scale scientific computations and simulations, economic planning, data protection and cryptography and many other applications require effective, carefully planned and precisely analysed algorithms.

Many years ago we wrote a small book with Péter Gács under the title *Algorithms.* The two volumes of the book *Algorithms of Computer Science* show how this topic developed into a complex area that branches off into many exciting directions. It gives a special pleasure to me that so many excellent representatives of Hungarian computer science have cooperated to create this book. It is obvious to me that this book will be one of the most important reference books for students, researchers and computer users for a long time.

Budapest, July 2007

László Lovász

Introduction

The first volume of the book *Informatikai algoritmusok* (in English: *Algorithms of Informatics*) appeared in 2004, and the second volume of the book appeared in 2005. Two volumes contained 31 chapters: 23 chapters of the present book, and further chapters on clustering (author: András Lukács), frequent elements in data bases (author: Ferenc Bodon), geoinformatics (authors: István Elek, Csaba Sidló), inner-point methods (authors: Tibor Illés, Marianna Nagy, Tamás Terlaky), number theory (authors: Gábor Farkas, Imre Kátai), Petri nets (authors: Zoltán Horváth, Máté Tejfel), queueing theory (authors: László Lakatos, László Szeidl, Miklós Telek), scheduling (author: Béla Vizvári).

The Hungarian version of the first volume contained those chapters which were finished until May of 2004, and the second volume contained the chapters finished until April of 2005.

English version contains the chapters submitted until April of 2007. Volume 1 contains the chapters belonging to the fundamentals of informatics, while the second volume contains the chapters having closer connection with some applications.

The chapters of the first volume are divided into three parts. The chapters of Part 1 are connected with automata: *Automata and Formal Languages* (written by Zoltán Kása, Babes-Bolyai University of Cluj-Napoca), *Compilers* (Zoltán Csörnyei, Eötvös Loránd University), *Compression and Decompression* (Ulrich Tamm, Chemnitz University of Technology Commitment), *Reliable Computations* (Péter Gács, Boston University).

The chapters of Part 2 have algebraic character: here are the chapters *Algebra* (written by Gábor Ivanyos, Lajos Rónyai, Budapest University of Technology and Economics), *Computer Algebra* (Antal Járai, Attila Kovács, Eötvös Loránd University), further *Cryptology* and *Complexity Theory* (Jörg Rothe, Heinrich Heine University).

The chapters of Part 3 have numeric character: *Competitive Analysis* (Csanád Imreh, University of Szeged), *Game Theory* (Ferenc Szidarovszky, The University of Arizona) and *Scientific Computations* (Aurél Galántai, András Jeney, University of Miskolc).

The second volume is also divided into three parts. The chapters of Part 4 are connected with computer networks: *Distributed Algorithms* (Burkhard Englert, California State University; Dariusz Kowalski, University of Liverpool; Grzegorz Malewicz, University of Alabama; Alexander Allister Shvartsman, University of Connecticut), *Network Simulation* (Tibor Gyires, Illinois State University), *Parallel Algorithms* (Antal Iványi, Eötvös Loránd University; Claudia Leopold, University of Kassel), and *Systolic Systems* (Eberhard Zehendner, Friedrich Schiller University).

The chapters of Part 5 are *Memory Management* (Ádám Balogh, Antal Iványi, Eötvös Loránd University), *Relational Databases* and *Query in Relational Databases* (János Demetrovics, Eötvös Loránd University; Attila Sali, Alfréd Rényi Institute of Mathematics), *Semi-structured Data Bases* (Attila Kiss, Eötvös Loránd University).

The chapters of Part 6 of the second volume have close connections with biology: *Bioinformatics* (István Miklós, Eötvös Loránd University), *Human-Computer Interactions* (Ingo Althöfer, Stefan Schwarz, Friedrich Schiller University), and *Computer Graphics* (László Szirmay-Kalos, Budapest University of Technology and Economics).

The chapters are validated by Gábor Ivanyos, Lajos Rónyai, András Recski, and Tamás Szántai (Budapest University of Technology and Economics), Sándor Fridli, János Gonda, and Béla Vizvári (Eötvös Loránd University), Pál Dömösi, and Attila Pethő (University of Debrecen), Zoltán Fülöp (University of Szeged), Anna Gál (University of Texas), János Mayer (University of Zürich).

The validators of the chapters which appeared only in the Hungarian version: István Pataricza, Lajos Rónyai (Budapest University of Economics and Technology), András A. Benczúr (Computer and Automation Research Institute), Antal Járai (Eötvös Loránd University), Attila Meskó (Hungarian Academy of Sciences), János Csirik (University of Szeged), János Mayer (University of Zürich),

The book contains verbal description, pseudocode and analysis of over 200 algorithms, and over 350 figures and 120 examples illustrating how the algorithms work. Each section ends with exercises and each chapter ends with problems. In the book you can find over 330 exercises and 70 problems.

We have supplied an extensive bibliography, in the section *Chapter notes* of each chapter. The web site of the book contains the maintained living version of the bibliography in which the names of authors, journals and publishers are usually links to the corresponding web site.

The LaTeX style file was written by Viktor Belényesi. The figures was drawn or corrected by Kornél Locher. Anna Iványi transformed the bibliography into hypertext.

The linguistical validators of the book are Anikó Hörmann and Veronika Vöröss. Some chapters were translated by Anikó Hörmann (Eötvös Loránd University), László Orosz (University of Debrecen), Miklós Péter Pintér (Corvinus University of Budapest), Csaba Schneider (Budapest University of Technology and Economics), and Veronika Vöröss (Eötvös Loránd University).

The publication of the book was supported by Department of Mathematics of Hungarian Academy of Science.

We plan to publish the corrected and extended version of this book in printed and electronic form too. This book has a web site: http://elek.inf.elte.hu/EnglishBooks. You can use this website to obtain a list of known errors, report errors, or make suggestions (using the data of the colofon page you can contact with any of the creators of the book). The website contains the maintaned PDF version of the bibliography in which the names of the authors, journals and publishers are usually active links to the corresponding web sites (the living elements are underlined in the printed bibliography). We welcome ideas for new exercises and problems.

Budapest, July 2007

Antal Iványi (tony@compalg.inf.elte.hu)

I. AUTOMATA

1. Automata and Formal Languages

Automata and formal languages play an important role in projecting and realizing compilers. In the first section grammars and formal languages are defined. The different grammars and languages are discussed based on Chomsky hierarchy. In the second section we deal in detail with the finite automata and the languages accepted by them, while in the third section the pushdown automata and the corresponding accepted languages are discussed. Finally, references from a rich bibliography are given.

1.1. Languages and grammars

A finite and nonempty set of symbols is called an ***alphabet***. The elements of an alphabet are ***letters,*** but sometimes are named also ***symbols.***

With the letters of an alphabet words are composed. If $a_1, a_2, \ldots, a_n \in \Sigma, n \geq 0$, then $a_1 a_2 \ldots a_n$ a Σ is a ***word*** over the alphabet Σ (the letters a_i are not necessary distinct). The number of letters of a word, with their multiplicities, constitutes the ***length*** of the word. If $w = a_1 a_2 \ldots a_n$, then the length of w is $|w| = n$. If $n = 0$, then the word is an ***empty word,*** which will be denoted by ε (sometimes λ in other books). The set of words over the alphabet Σ will be denoted by Σ^*:

$$\Sigma^* = \left\{ a_1 a_2 \ldots a_n \mid a_1, a_2, \ldots, a_n \in \Sigma,\, n \geq 0 \right\} .$$

For the set of nonempty words over Σ the notation $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ will be used. The set of words of length n over Σ will be denoted by Σ^n, and $\Sigma^0 = \{\varepsilon\}$. Then

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \cdots \cup \Sigma^n \cup \cdots \quad \text{and} \quad \Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \cdots \cup \Sigma^n \cup \cdots .$$

The words $u = a_1 a_2 \ldots a_m$ and $v = b_1 b_2 \ldots b_n$ are equal (i.e. $u = v$), if $m = n$ and $a_i = b_i,\ \ i = 1, 2, \ldots, n$.

We define in Σ^* the binary operation called ***concatenation***. The concatenation (or product) of the words $u = a_1 a_2 \ldots a_m$ and $v = b_1 b_2 \ldots b_n$ is the word $uv = a_1 a_2 \ldots a_m b_1 b_2 \ldots b_n$. It is clear that $|uv| = |u| + |v|$. This operation is associative but not commutative. Its neutral element is ε, because $\varepsilon u = u\varepsilon = u$ for all $u \in \Sigma^*$. Σ^* with the concatenation is a monoid.

We introduce the power operation. If $u \in \Sigma^*$, then $u^0 = \varepsilon$, and $u^n = u^{n-1}u$ for $n \geq 1$.

The ***reversal*** (or ***mirror image***) of the word $u = a_1 a_2 \dots a_n$ is $u^{-1} = a_n a_{n-1} \dots a_1$. The reversal of u sometimes is denoted by u^R or $\tilde{u}$. It is clear that $\left(u^{-1}\right)^{-1} = u$ and $(uv)^{-1} = v^{-1}u^{-1}$.

Word v is a ***prefix*** of the word u if there exists a word z such that $u = vz$. If $z \neq \varepsilon$ then v is a proper prefix of u. Similarly v is a ***suffix*** of u if there exists a word x such that $u = xv$. The proper suffix can also be defined. Word v is a ***subword*** of the word u if there are words p and q such that $u = pvq$. If $pq \neq \varepsilon$ then v is a ***proper subword***.

A subset L of Σ^* is called a ***language*** over the alphabet Σ. Sometimes this is called a ***formal language*** because the words are here considered without any meanings. Note that $\emptyset$ is the empty language while $\{\varepsilon\}$ is a language which contains the empty word.

1.1.1. Operations on languages

If L, L_1, L_2 are languages over Σ we define the following operations

- *union*
$$L_1 \cup L_2 = \{u \in \Sigma^* \mid u \in L_1 \text{ or } u \in L_2\} ,$$
- *intersection*
$$L_1 \cap L_2 = \{u \in \Sigma^* \mid u \in L_1 \text{ and } u \in L_2\} ,$$
- *difference*
$$L_1 \setminus L_2 = \{u \in \Sigma^* \mid u \in L_1 \text{ and } u \notin L_2\} ,$$
- *complement*
$$\overline{L} = \Sigma^* \setminus L ,$$
- *multiplication*
$$L_1 L_2 = \{uv \mid u \in L_1, v \in L_2\} ,$$
- *power*
$$L^0 = \{\varepsilon\}, \qquad L^n = L^{n-1}L, \text{ if } n \geq 1 ,$$
- *iteration* or *star operation*
$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L \cup L^2 \cup \dots \cup L^i \cup \dots ,$$
- *mirror*
$$L^{-1} = \{u^{-1} \mid u \in L\} .$$

We will use also the notation L^+
$$L^+ = \bigcup_{i=1}^{\infty} L^i = L \cup L^2 \cup \dots \cup L^i \cup \dots .$$
The union, product and iteration are called ***regular operations***.

1.1.2. Specifying languages

Languages can be specified in several ways. For example a language can be specified using

1) the enumeration of its words,

2) a property, such that all words of the language have this property but other word have not,

3) a grammar.

Specifying languages by listing their elements For example the following are languages

$L_1 = \{\varepsilon, 0, 1\}$,

$L_2 = \{a, aa, aaa, ab, ba, aba\}$.

Even if we cannot enumerate the elements of an infinite set infinite languages can be specified by enumeration if after enumerating the first some elements we can continue the enumeration using a rule. The following is such a language

$L_3 = \{\varepsilon, ab, aabb, aaabbb, aaaabbbb, \ldots\}$.

Specifying languages by properties The following sets are languages

$L_4 = \{a^n b^n \mid n = 0, 1, 2, \ldots\}$,

$L_5 = \{uu^{-1} \mid u \in \Sigma^*\}$,

$L_6 = \{u \in \{a, b\}^* \mid n_a(u) = n_b(u)\}$,

where $n_a(u)$ denotes the number of letters a in word u and $n_b(u)$ the number of letters b.

Specifying languages by grammars Define the ***generative grammar*** or shortly the ***grammar***.

Definition 1.1 *A **grammar** is an ordered quadruple $G = (N, T, P, S)$, where*

- *N is the alphabet of **variables** (or **nonterminal symbols**),*
- *T is the alphabet of **terminal symbols**, where $N \cap T = \emptyset$,*
- *$P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$ is a finite set, that is P is the finite set of **productions** of the form (u, v) , where $u, v \in (N \cup T)^*$ and u contains at least a nonterminal symbol,*
- *$S \in N$ is the **start symbol**.*

Remarks. Instead of the notation (u, v) sometimes $u \to v$ is used.

In the production $u \to v$ or (u, v) word u is called the left-hand side of the production while v the right-hand side. If for a grammar there are more than one production with the same left-hand side, then these production

$$u \to v_1,\ u \to v_2, \ldots, u \to v_r \quad \text{can be written as} \quad u \to v_1 \mid v_2 \mid \ldots \mid v_r\ .$$

We define on the set $(N \cup T)^*$ the relation called ***direct derivation***

$$u \Longrightarrow v, \qquad \text{if} \quad u = p_1 p p_2, \quad v = p_1 q p_2 \quad \text{and} \quad (p, q) \in P\ .$$

In fact we replace in u an appearance of the subword p by q and we get v. Another notations for the same relation can be $\vdash$ or $\models$.

If we want to emphasize the used grammar G, then the notation $\Longrightarrow$ can be replaced by $\underset{G}{\Longrightarrow}$. Relation $\overset{*}{\Longrightarrow}$ is the reflexive and transitive closure of $\Longrightarrow$, while $\overset{+}{\Longrightarrow}$ denotes its transitive closure. Relation $\overset{*}{\Longrightarrow}$ is called a ***derivation***.

From the definition of a reflexive and transitive relation we can deduce the following: $u \overset{*}{\Longrightarrow} v$, if there exist the words $w_0, w_1, \ldots, w_n \in (N \cup T)^*$, $n \geq 0$ and $u = w_0$, $w_0 \Longrightarrow w_1$, $w_1 \Longrightarrow w_2$, $\ldots, w_{n-1} \Longrightarrow w_n$, $w_n = v$. This can be written shortly $u = w_0 \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow \ldots \Longrightarrow w_{n-1} \Longrightarrow w_n = v$. If $n = 0$ then $u = v$. The same way we can define the relation $u \overset{+}{\Longrightarrow} v$ except that $n \geq 1$ always, so at least one direct derivation will de used.

Definition 1.2 *The **language generated** by grammar $G = (N, T, P, S)$ is the set*

$$L(G) = \{u \in T^* \mid S \overset{*}{\Longrightarrow} u\} .$$

So $L(G)$ contains all words over the alphabet T which can be derived from the start symbol S using the productions from P.

Example 1.1 Let $G = (N, T, P, S)$ where
$N = \{S\}$,
$T = \{a, b\}$,
$P = \{S \to aSb,\ S \to ab\}$.
It is easy to see than $L(G) = \{a^n b^n \mid n \geq 1\}$ because

$$S \underset{G}{\Longrightarrow} aSb \underset{G}{\Longrightarrow} a^2Sb^2 \underset{G}{\Longrightarrow} \cdots \underset{G}{\Longrightarrow} a^{n-1}Sb^{n-1} \underset{G}{\Longrightarrow} a^n b^n ,$$

where up to the last but one replacement the first production ($S \to aSb$) was used, while at the last replacement the production $S \to ab$. This derivation can be written $S \overset{*}{\underset{G}{\Longrightarrow}} a^n b^n$. Therefore $a^n b^n$ can be derived from S for all n and no other words can be derived from S.

Definition 1.3 *Two grammars G_1 and G_2 are **equivalent**, and this is denoted by $G_1 \cong G_2$ if $L(G_1) = L(G_2)$.*

Example 1.2 The following two grammars are equivalent because both of them generate the language $\{a^n b^n c^n \mid n \geq 1\}$.
$G_1 = (N_1, T, P_1, S_1)$, where
$N_1 = \{S_1, X, Y\}$, $T = \{a, b, c\}$,
$P_1 = \{S_1 \to abc,\ S_1 \to aXbc,\ Xb \to bX,\ Xc \to Ybcc,\ bY \to Yb,\ aY \to aaX,\ aY \to aa\}$.
$G_2 = (N_2, T, P_2, S_2)$, where
$N_2 = \{S_2, A, B, C\}$,
$P_2 = \{S_2 \to aS_2BC,\ S_2 \to aBC,\ CB \to BC,\ aB \to ab,\ bB \to bb,\ bC \to bc,\ cC \to cc\}$.
First let us prove by mathematical induction that for $n \geq 2$ $S_1 \overset{*}{\underset{G_1}{\Longrightarrow}} a^{n-1}Yb^n c^n$. If $n = 2$ then

$$S_1 \underset{G_1}{\Longrightarrow} aXbc \underset{G_1}{\Longrightarrow} abXc \underset{G_1}{\Longrightarrow} abYbcc \underset{G_1}{\Longrightarrow} aYb^2c^2 .$$

The inductive hypothesis is $S_1 \overset{*}{\underset{G_1}{\Longrightarrow}} a^{n-2}Yb^{n-1}c^{n-1}$. We use production $aY \to aaX$, then $(n-1)$ times production $Xb \to bX$, and then production $Xc \to Ybcc$, afterwards again $(n-1)$ times production $bY \to Yb$. Therefore

$$S_1 \underset{G_1}{\Longrightarrow} a^{n-2}Yb^{n-1}c^{n-1} \underset{G_1}{\Longrightarrow} a^{n-1}Xb^{n-1}c^{n-1} \overset{*}{\underset{G_1}{\Longrightarrow}}$$
$$a^{n-1}b^{n-1}Xc^{n-1} \underset{G_1}{\Longrightarrow} a^{n-1}b^{n-1}Ybc^n \overset{*}{\underset{G_1}{\Longrightarrow}} a^{n-1}Yb^n c^n .$$

If now we use production $aY \to aa$ we get $S_1 \underset{G_1}{\overset{*}{\Longrightarrow}} a^n b^n c^n$ for $n \geq 2$, but $S_1 \underset{G_1}{\Longrightarrow} abc$ by the production $S_1 \to abc$, so $a^n b^n c^n \in L(G_1)$ for any $n \geq 1$. We have to prove in addition that using the productions of the grammar we cannot derive only words of the form $a^n b^n c^n$. It is easy to see that a successful derivation (which ends in a word containing only terminals) can be obtained only in the presented way.
Similarly for $n \geq 2$

$$S_2 \underset{G_2}{\Longrightarrow} aS_2BC \underset{G_2}{\overset{*}{\Longrightarrow}} a^{n-1}S_2(BC)^{n-1} \underset{G_2}{\Longrightarrow} a^n(BC)^n \underset{G_2}{\overset{*}{\Longrightarrow}} a^n B^n C^n$$

$$\underset{G_2}{\Longrightarrow} a^n bB^{n-1}C^n \underset{G_2}{\overset{*}{\Longrightarrow}} a^n b^n C^n \underset{G_2}{\Longrightarrow} a^n b^n cC^{n-1} \underset{G_2}{\overset{*}{\Longrightarrow}} a^n b^n c^n \ .$$

Here orderly were used the productions $S_2 \to aS_2BC$ ($n-1$ times), $S_2 \to aBC$, $CB \to BC$ ($n-1$ times), $aB \to ab$, $bB \to bb$ ($n-1$ times), $bC \to bc$, $cC \to cc$ ($n-1$ times). But $S_2 \underset{G_2}{\Longrightarrow} aBC \underset{G_2}{\Longrightarrow} abC \underset{G_2}{\Longrightarrow} abc$, So $S_2 \underset{G_2}{\overset{*}{\Longrightarrow}} a^n b^n c^n$, $n \geq 1$. It is also easy to see than other words cannot be derived using grammar G_2.

The grammars
$G_3 = (\{S\}, \{a, b\}, \{S \to aSb,\ S \to \varepsilon\}, S)$ and
$G_4 = (\{S\}, \{a, b\}, \{S \to aSb,\ S \to ab\}, S)$
are not equivalent because $L(G_3) \setminus \{\varepsilon\} = L(G_4)$.

Theorem 1.4 *Not all languages can be generated by grammars.*

Proof We encode grammars for the proof as words on the alphabet $\{0, 1\}$. For a given grammar $G = (N, T, P, S)$ let $N = \{S_1, S_2, \ldots, S_n\}$, $T = \{a_1, a_2, \ldots, a_m\}$ and $S = S_1$. The encoding is the following:

the code of S_i is $10\underbrace{11\ldots11}_{i \text{ times}}01$, the code of a_i is $100\underbrace{11\ldots11}_{i \text{ times}}001$.

In the code of the grammar the letters are separated by 000, the code of the arrow is 0000, and the productions are separated by 00000.

It is enough, of course, to encode the productions only. For example, consider the grammar

$G = (\{S\}, \{a, b\}, \{S \to aSb,\ S \to ab\}, S)$.

The code of S is 10101, the code of a is 1001001, the code of b is 10011001. The code of the grammar is

$$\underbrace{10101}0000\underbrace{1001001}000\underbrace{10101}000\underbrace{10011001}00000\underbrace{10101}0000\underbrace{1001001}000$$

$$\underbrace{10011001}\ .$$

From this encoding results that the grammars with terminal alphabet T can be enumerated [1] as $G_1, G_2, \ldots, G_k, \ldots$, and the set of these grammars is a denumerable infinite set.

[1] Let us suppose that in the alphabet $\{0, 1\}$ there is a linear order $<$, let us say $0 < 1$. The words which are codes of grammars can be enumerated by ordering them first after their lengths, and inside the equal length words, alphabetically, using the order of their letters. But we can use equally the lexicographic order, which means that $u < v$ (u is before v) if u is a proper prefix of v or there exists the decompositions $u = xay$ and $v = xby'$, where x, y, y' are subwords, a and b letters with $a < b$.

Consider now the set of all languages over T denoted by $\mathcal{L}_T = \{L \mid L \subseteq T^*\}$, that is $\mathcal{L}_T = \mathcal{P}(T^*)$. The set T^* is denumerable because its words can be ordered. Let this order $s_0, s_1, s_2, \ldots$, where $s_0 = \varepsilon$. We associate to each language $L \in \mathcal{L}_T$ an infinite binary sequence $b_0, b_1, b_2, \ldots$ the following way:

$$b_i = \begin{cases} 1, & \text{if } s_i \in L \\ 0, & \text{if } s_i \notin L \end{cases} \qquad i = 0, 1, 2, \ldots \ .$$

It is easy to see that the set of all such binary sequences is not denumerable, because each sequence can be considered as a positive number less than 1 using its binary representation (The decimal point is considered to be before the first digit). Conversely, to each positive number less than 1 in binary representation a binary sequence can be associated. So, the cardinality of the set of infinite binary sequences is equal to cardinality of interval $[0, 1]$, which is of continuum power. Therefore the set $\mathcal{L}_T$ is of continuum cardinality. Now to each grammar with terminal alphabet T associate the corresponding generated language over T. Since the cardinality of the set of grammars is denumerable, there will exist a language from $\mathcal{L}_T$, without associated grammar, a language which cannot be generated by a grammar. ■

1.1.3. Chomsky hierarchy of grammars and languages

Putting some restrictions on the form of productions, four type of grammars can be distinguished.

Definition 1.5 *Define for a grammar $G = (N, T, P, S)$ the following four types.*

*A grammar G is of type 0 (**phrase-structure grammar**) if there are no restrictions on productions.*

*A grammar G is of type 1 **(context-sensitive grammar)** if all of its productions are of the form $\alpha A \gamma \to \alpha \beta \gamma$, where $A \in N$, $\alpha, \gamma \in (N \cup T)^*$, $\beta \in (N \cup T)^+$. A production of the form $S \to \varepsilon$ can also be accepted if the start symbol S does not occur in the right-hand side of any production.*

*A grammar G is of type 2 **(context-free grammar)** if all of its productions are of the form $A \to \beta$, where $A \in N$, $\beta \in (N \cup T)^+$. A production of the form $S \to \varepsilon$ can also be accepted if the start symbol S does not occur in the right-hand side of any production.*

*A grammar G is of type 3 **(regular grammar)** if its productions are of the form $A \to aB$ or $A \to a$, where $a \in T$ and $A, B \in N$. A production of the form $S \to \varepsilon$ can also be accepted if the start symbol S does not occur in the right-hand side of any production.*

If a grammar G is of type i then language $L(G)$ is also of type i.

This classification was introduced by Noam Chomsky.

A language L is of type i $(i = 0, 1, 2, 3)$ if there exists a grammar G of type i which generates the language L, so $L = L(G)$.

Denote by $\mathcal{L}_i$ $(i = 0, 1, 2, 3)$ the class of the languages of type i. Can be proved that

$$\mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3 \ .$$

By the definition of different type of languages, the inclusions ($\supseteq$) are evident, but the strict inclusions ($\supset$) must be proved.

Example 1.3 We give an example for each type of context-sensitive, context-free and regular grammars.
Context-sensitive grammar. $G_1 = (N_1, T_1, P_1, S_1)$, where $N_1 = \{S_1, A, B, C\}$, $T_1 = \{a, 0, 1\}$.

Elements of P_1 are:
$S_1 \rightarrow ACA$,
$AC \rightarrow AACA \mid ABa \mid AaB$,
$B \rightarrow AB \mid A$,
$A \rightarrow 0 \mid 1$.

Language $L(G_1)$ contains words of the form uav with $u, v \in \{0,1\}^*$ and $|u| \neq |v|$.
Context-free grammar. $G_2 = (N_2, T_2, P_2, S)$, where $N_2 = \{S, A, B\}$, $T_2 = \{+, *, (,), a\}$.

Elements of P_2 are:
$S \rightarrow S + A \mid A$,
$A \rightarrow A * B \mid B$,
$B \rightarrow (S) \mid a$.

Language $L(G_2)$ contains algebraic expressions which can be correctly built using letter a, operators $+$ and $*$ and brackets.
Regular grammar. $G_3 = (N_3, T_3, P_3, S_3)$, where $N_3 = \{S_3, A, B\}$, $T_3 = \{a, b\}$.

Elements of P_3 are:
$S_3 \rightarrow aA$
$A \rightarrow aB \mid a$
$B \rightarrow aB \mid bB \mid a \mid b$.

Language $L(G_3)$ contains words over the alphabet $\{a, b\}$ with at least two letters a at the beginning.

It is easy to prove that any finite language is regular. The productions will be done to generate all words of the language. For example, if $u = a_1 a_2 \dots a_n$ is in the language, then we introduce the productions: $S \rightarrow a_1 A_1$, $A_1 \rightarrow a_2 A_2$, $\dots A_{n-2} \rightarrow a_{n-1} A_{n-1}$, $A_{n-1} \rightarrow a_n$, where S is the start symbol of the language and $A_1, \dots, A_{n-1}$ are distinct nonterminals. We define such productions for all words of the language using different nonterminals for different words, excepting the start symbol S. If the empty word is also an element of the language, then the production $S \rightarrow \varepsilon$ is also considered.

The empty set is also a regular language, because the regular grammar $G = (\{S\}, \{a\}, \{S \rightarrow aS\}, S)$ generates it.

Eliminating unit productions A production of the form $A \rightarrow B$ is called a ***unit production***, where $A, B \in N$. Unit productions can be eliminated from a grammar in such a way that the new grammar will be of the same type and equivalent to the first one.

Let $G = (N, T, P, S)$ be a grammar with unit productions. Define an equivalent grammar $G' = (N, T, P', S)$ without unit productions. The following algorithm will construct the equivalent grammar.

ELIMINATE-UNIT-PRODUCTIONS(G)

1 if the unit productions $A \to B$ and $B \to C$ are in P put also
the unit production $A \to C$ in P while P can be extended,
2 if the unit production $A \to B$ and the production $B \to \alpha$ ($\alpha \notin N$) are in P
put also the production $A \to \alpha$ in P,
3 let P' be the set of productions of P except unit productions
4 **return** G'

Clearly, G and G' are equivalent. If G is of type $i \in \{0,1,2,3\}$ then G' is also of type i.

Example 1.4 Use the above algorithm in the case of the grammar $G = \{S,A,B,C\},\{a,b\},P,S$, where P contains

$S \to A$, $A \to B$, $B \to C$, $C \to B$, $D \to C$,
$S \to B$, $A \to D$, $C \to Aa$,
$A \to aB$,
$A \to b$.

Using the first step of the algorithm, we get the following new unit productions:

$S \to D$ (because of $S \to A$ and $A \to D$),
$S \to C$ (because of $S \to B$ and $B \to C$),
$A \to C$ (because of $A \to B$ and $B \to C$),
$B \to B$ (because of $B \to C$ and $C \to B$),
$C \to C$ (because of $C \to B$ and $B \to C$),
$D \to B$ (because of $D \to C$ and $C \to B$).

In the second step of the algorithm will be considered only productions with A or C in the right-hand side, since productions $A \to aB$, $A \to b$ and $C \to Aa$ can be used (the other productions are all unit productions). We get the following new productions:

$S \to aB$ (because of $S \to A$ and $A \to aB$),
$S \to b$ (because of $S \to A$ and $A \to b$),
$S \to Aa$ (because of $S \to C$ and $C \to Aa$),
$A \to Aa$ (because of $A \to C$ and $C \to Aa$),
$B \to Aa$ (because of $B \to C$ and $C \to Aa$).

The new grammar $G' = \{S,A,B,C\},\{a,b\},P',S$ will have the productions:

$S \to b$, $A \to b$, $B \to Aa$, $C \to Aa$,
$S \to aB$, $A \to aB$,
$S \to Aa$ $A \to Aa$,

Grammars in normal forms A grammar is to be said a ***grammar in normal form*** if its productions have no terminal symbols in the left-hand side.

We need the following notions. For alphabets Σ_1 and Σ_2 a *homomorphism* is a function $h : \Sigma_1^* \to \Sigma_2^*$ for which $h(u_1u_2) = h(u_1)h(u_2)$, $\forall u_1, u_2 \in \Sigma_1^*$. It is easy to see that for arbitrary $u = a_1a_2 \ldots a_n \in \Sigma_1^*$ value $h(u)$ is uniquely determined by the restriction of h on Σ_1, because $h(u) = h(a_1)h(a_2) \ldots h(a_n)$.

If a homomorphism h is a bijection then h an *isomorphism*.

Theorem 1.6 *To any grammar an equivalent grammar in normal form can be associated.*

Proof Grammars of type 2 and 3 have in left-hand side of any productions only a nonterminal, so they are in normal form. The proof has to be done for grammars of type 0 and 1 only.

Let $G = (N, T, P, S)$ be the original grammar and we define the grammar in normal form as $G' = (N', T, P', S)$.

Let $a_1, a_2, \ldots, a_k$ be those terminal symbols which occur in the left-hand side of productions. We introduce the new nonterminals $A_1, A_2, \ldots, A_k$. The following notation will be used: $T_1 = \{a_1, a_2, \ldots, a_k\}$, $T_2 = T \setminus T_1$, $N_1 = \{A_1, A_2, \ldots, A_k\}$ and $N' = N \cup N_1$.

Define the isomorphism $h : N \cup T \longrightarrow N' \cup T_2$, where

$$\begin{array}{ll} h(a_i) = A_i, & \text{if} \quad a_i \in T_1, \\ h(X) = X, & \text{if} \quad X \in N \cup T_2 \end{array}$$

Define the set P' of production as

$$P' = \Big\{ h(\alpha) \to h(\beta) \;\Big|\; (\alpha \to \beta) \in P \Big\} \cup \Big\{ A_i \longrightarrow a_i \;\Big|\; i = 1, 2, \ldots, k \Big\}$$

In this case $\alpha \underset{G}{\overset{*}{\Longrightarrow}} \beta$ if and only if $h(\alpha) \underset{G'}{\overset{*}{\Longrightarrow}} h(\beta)$. From this the theorem immediately results because $S \underset{G}{\overset{*}{\Longrightarrow}} u \;\Leftrightarrow\; S = h(S) \underset{G'}{\overset{*}{\Longrightarrow}} h(u) = u$. ∎

Example 1.5 Let $G = (\{S, D, E\}, \{a, b, c, d, e\}, P, S)$, where P contains

$$\begin{array}{lcl} S & \to & aebc \mid aDbc \\ Db & \to & bD \\ Dc & \to & Ebccd \\ bE & \to & Eb \\ aE & \to & aaD \mid aae \end{array}$$

In the left-hand side of productions the terminals a, b, c occur, therefore consider the new nonterminals A, B, C, and include in P' also the new productions $A \to a$, $B \to b$ and $C \to c$.

Terminals a, b, c will be replaced by nonterminals A, B, C respectively, and we get the set P' as

$$\begin{array}{lcl} S & \to & AeBC \mid ADBC \\ DB & \to & BD \\ DC & \to & EBCCd \\ BE & \to & EB \\ AE & \to & AAD \mid AAe \\ A & \to & a \\ B & \to & b \\ C & \to & c. \end{array}$$

Let us see what words can be generated by this grammars. It is easy to see that $aebc \in L(G')$, because $S \Longrightarrow AeBC \overset{*}{\Longrightarrow} aebc$.

$S \Longrightarrow ADBC \Longrightarrow ABDC \Longrightarrow ABEBCCd \Longrightarrow AEBBCCd \Longrightarrow AAeBBCCd \overset{*}{\Longrightarrow} aaebbccd$, so $aaebbccd \in L(G')$.

We prove, using the mathematical induction, that $S \overset{*}{\Longrightarrow} A^{n-1}EB^nC(Cd)^{n-1}$ for $n \geq 2$. For $n = 2$ this is the case, as we have seen before. Continuing the derivation we get $S \overset{*}{\Longrightarrow} A^{n-1}EB^nC(Cd)^{n-1} \Longrightarrow A^{n-2}AADB^nC(Cd)^{n-1} \overset{*}{\Longrightarrow} A^nB^nDC(Cd)^{n-1} \Longrightarrow A^nB^nEBCCd(Cd)^{n-1} \overset{*}{\Longrightarrow} A^nEB^{n+1}CCd(Cd)^{n-1} = A^nEB^{n+1}C(Cd)^n$, and this is what we had to prove.

But $S \overset{*}{\Longrightarrow} A^{n-1}EB^nC(Cd)^{n-1} \Longrightarrow A^{n-2}AAeB^nC(Cd)^{n-1} \overset{*}{\Longrightarrow} a^neb^nc(cd)^{n-1}$. So $a^neb^nc(cd)^{n-1} \in L(G'),\ n \geq 1$. These words can be generated also in G.

1.1.4. Extended grammars

In this subsection extended grammars of type 1, 2 and 3 will be presented.

Extended grammar of type 1. All productions are of the form $\alpha \to \beta$, where $|\alpha| \leq |\beta|$, excepted possibly the production $S \to \varepsilon$.

Extended grammar of type 2. All productions are of the form $A \to \beta$, where $A \in N, \beta \in (N \cup T)^*$.

Extended grammar of type 3. All productions are of the form $A \to uB$ or $A \to u$, Where $A, B \in N, u \in T^*$.

Theorem 1.7 *To any extended grammar an equivalent grammar of the same type can be associated.*

Proof Denote by G_{ext} the extended grammar and by G the corresponding equivalent grammar of the same type.

Type 1. Define the productions of grammar G by rewriting the productions $\alpha \to \beta$, where $|\alpha| \leq |\beta|$ of the extended grammar G_{ext} in the form $\gamma_1\delta\gamma_2 \to \gamma_1\gamma\gamma_2$ allowed in the case of grammar G by the following way.

Let $X_1X_2 \dots X_m \to Y_1Y_2 \dots Y_n$ $(m \leq n)$ be a production of G_{ext}, which is not in the required form. Add to the set of productions of G the following productions, where $A_1,\ A_2,\ \dots,\ A_m$ are new nonterminals:

$$
\begin{array}{lcl}
X_1X_2 \dots X_m & \to & A_1X_2X_3 \dots X_m \\
A_1X_2 \dots X_m & \to & A_1A_2X_3 \dots X_m \\
 & \dots & \\
A_1A_2 \dots A_{m-1}X_m & \to & A_1A_2 \dots A_{m-1}A_m \\
A_1A_2 \dots A_{m-1}A_m & \to & Y_1A_2 \dots A_{m-1}A_m \\
Y_1A_2 \dots A_{m-1}A_m & \to & Y_1Y_2 \dots A_{m-1}A_m \\
 & \dots & \\
Y_1Y_2 \dots Y_{m-2}A_{m-1}A_m & \to & Y_1Y_2 \dots Y_{m-2}Y_{m-1}A_m \\
Y_1Y_2 \dots Y_{m-1}A_m & \to & Y_1Y_2 \dots Y_{m-1}Y_mY_{m+1} \dots Y_n.
\end{array}
$$

Furthermore, add to the set of productions of G without any modification the productions of G_{ext} which are of permitted form, i.e. $\gamma_1\delta\gamma_2 \to \gamma_1\gamma\gamma_2$.

Inclusion $L(G_{ext}) \subseteq L(G)$ can be proved because each used production of G_{ext} in a derivation can be simulated by productions G obtained from it. Furthermore, since the productions of G can be used only in the prescribed order, we could not obtain other words, so $L(G) \subseteq L(G_{ext})$ also is true.

Type 2. Let $G_{ext} = (N, T, P, S)$. Productions of form $A \to \varepsilon$ have to be eliminated, only $S \to \varepsilon$ can remain, if S doesn't occur in the right-hand side of productions. For this define the following sets:

$U_0 = \{A \in N \mid (A \to \varepsilon) \in P\}$

$U_i = U_{i-1} \cup \{A \in N \mid (A \to w) \in P,\ w \in U_{i-1}^+\}$.

Since for $i \geq 1$ we have $U_{i-1} \subseteq U_i$, $U_i \subseteq N$ and N is a finite set, there must exists such a k for which $U_{k-1} = U_k$. Let us denote this set as U. It is easy to see

that a nonterminal A is in U if and only if $A \stackrel{*}{\Longrightarrow} \varepsilon$. (In addition $\varepsilon \in L(G_{ext})$ if and only if $S \in U$.)

We define the productions of G starting from the productions of G_{ext} in the following way. For each production $A \to \alpha$ with $\alpha \neq \varepsilon$ of G_{ext} add to the set of productions of G this one and all productions which can be obtained from it by eliminating from α one or more nonterminals which are in U, but only in the case when the right-hand side does not become ε.

It in not difficult to see that this grammar G generates the same language as G_{ext} does, except the empty word ε. So, if $\varepsilon \notin L(G_{ext})$ then the proof is finished. But if $\varepsilon \in L(G_{ext})$, then there are two cases. If the start symbol S does not occur in any right-hand side of productions, then by introducing the production $S \to \varepsilon$, grammar G will generate also the empty word. If S occurs in a production in the right-hand side, then we introduce a new start symbol S' and the new productions $S' \to S$ and $S' \to \varepsilon$. Now the empty word ε can also be generated by grammar G.

Type 3. First we use for G_{ext} the procedure defined for grammars of type 2 to eliminate productions of the form $A \to \varepsilon$. From the obtained grammar we eliminate the unit productions using the algorithm ELIMINATE-UNIT-PRODUCTIONS (see page 20).

In the obtained grammar for each production $A \to a_1 a_2 \dots a_n B$, where $B \in N \cup \{\varepsilon\}$, add to the productions of G also the followings

$$\begin{aligned} A &\to a_1 A_1, \\ A_1 &\to a_2 A_2, \\ &\dots \\ A_{n-1} &\to a_n B, \end{aligned}$$

where $A_1, A_2, \dots, A_{n-1}$ are new nonterminals. It is easy to prove that grammar G built in this way is equivalent to G_{ext}. ■

Example 1.6 Let $G_{ext} = (N, T, P, S)$ be an extended grammar of type 1, where $N = \{S, B, C\}$, $T = \{a, b, c\}$ and P contains the following productions:

$$\begin{array}{llll} S \to aSBC \mid aBC & & CB \to BC \\ aB \to ab & & bB \to bb \\ bC \to bc & & cC \to cc\,. \end{array}$$

The only production which is not context-sensitive is $CB \to BC$. Using the method given in the proof, we introduce the productions:

$$\begin{aligned} CB &\to AB \\ AB &\to AD \\ AD &\to BD \\ BD &\to BC \end{aligned}$$

Now the grammar $G = (\{S, A, B, C, D\}, \{a, b, c\}, P', S)$ is context-sensitive, where the elements of P' are

$$\begin{array}{llll} S \to aSBC \mid aBC & & & \\ CB \to AB & & aB \to ab \\ AB \to AD & & bB \to bb \\ AD \to BD & & bC \to bc \\ BD \to BC & & cC \to cc. \end{array}$$

It can be proved that $L(G_{ext}) = L(G) = \{a^n b^n c^n \mid n \geq 1\}$.

Example 1.7 Let $G_{ext} = (\{S, B, C\}, \{a, b, c\}, P, S)$ be an extended grammar of type 2, where P contains:

$S \rightarrow aSc \mid B$
$B \rightarrow bB \mid C$
$C \rightarrow Cc \mid \varepsilon.$

Then $U_0 = \{C\}$, $U_1 = \{B, C\}$, $U_3 = \{S, B, C\} = U$. The productions of the new grammar are:

$S \rightarrow aSc \mid ac \mid B$
$B \rightarrow bB \mid b \mid C$
$C \rightarrow Cc \mid c.$

The original grammar generates also the empty word and because S occurs in the right-hand side of a production, a new start symbol and two new productions will be defined: $S' \rightarrow S, S' \rightarrow \varepsilon$. The context-free grammar equivalent to the original grammar is $G = (\{S', S, B, C\}, \{a, b, c\}, P', S')$ with the productions:

$S' \rightarrow S \mid \varepsilon$
$S \rightarrow aSc \mid ac \mid B$
$B \rightarrow bB \mid b \mid C$
$C \rightarrow Cc \mid c.$

Both of these grammars generate language $\{a^m b^n c^p \mid p \geq m \geq 0, n \geq 0\}$.

Example 1.8 Let $G_{ext} = (\{S, A, B\}, \{a, b\}, P, S)$ be the extended grammar of type 3 under examination, where P:

$S \rightarrow abA$
$A \rightarrow bB$
$B \rightarrow S \mid \varepsilon.$

First, we eliminate production $B \rightarrow \varepsilon$. Since $U_0 = U = \{B\}$, the productions will be

$S \rightarrow abA$
$A \rightarrow bB \mid b$
$B \rightarrow S.$

The latter production (which a unit production) can also be eliminated, by replacing it with $B \rightarrow abA$. Productions $S \rightarrow abA$ and $B \rightarrow abA$ have to be transformed. Since, both productions have the same right-hand side, it is enough to introduce only one new nonterminal and to use the productions $S \rightarrow aC$ and $C \rightarrow bA$ instead of $S \rightarrow abA$. Production $B \rightarrow abA$ will be replaced by $B \rightarrow aC$. The new grammar is $G = (\{S, A, B, C\}, \{a, b\}, P', S)$, where P':

$S \rightarrow aC$
$A \rightarrow bB \mid b$
$B \rightarrow aC$
$C \rightarrow bA.$

Can be proved that $L(G_{ext}) = L(G) = \{(abb)^n \mid n \geq 1\}$.

1.1.5. Closure properties in the Chomsky-classes

We will prove the following theorem, by which the Chomsky-classes of languages are closed under the regular operations, that is, the union and product of two languages of type i is also of type i, the iteration of a language of type i is also of type i $(i = 0, 1, 2, 3)$.

Theorem 1.8 *The class $\mathcal{L}_i$ $(i = 0, 1, 2, 3)$ of languages is closed under the regular operations.*

Proof For the proof we will use extended grammars. Consider the extended grammars $G_1 = (N_1, T_1, P_1, S_1)$ and $G_2 = (N_2, T_2, P_2, S_2)$ of type i each. We can suppose that $N_1 \cap N_2 = \emptyset$.

Union. Let $G_\cup = (N_1 \cup N_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$.

We will show that $L(G_\cup) = L(G_1) \cup L(G_2)$. If $i = 0, 2, 3$ then from the assumption that G_1 and G_2 are of type i follows by definition that $G_\cup$ also is of type i. If $i = 1$ and one of the grammars generates the empty word, then we eliminate from $G_\cup$ the corresponding production (possibly the both) $S_k \to \varepsilon$ $(k = 1, 2)$ and replace it by production $S \to \varepsilon$.

Product. Let $G_\times = (N_1 \cup N_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$.

We will show that $L(G_\times) = L(G_1)L(G_2)$. By definition, if $i = 0, 2$ then $G_\times$ will be of the same type. If $i = 1$ and there is production $S_1 \to \varepsilon$ in P_1 but there is no production $S_2 \to \varepsilon$ in P_2 then production $S_1 \to \varepsilon$ will be replaced by $S \to S_2$. We will proceed the same way in the symmetrical case. If there is in P_1 production $S_1 \to \varepsilon$ and in P_2 production $S_2 \to \varepsilon$ then they will be replaced by $S \to \varepsilon$.

In the case of regular grammars ($i = 3$), because $S \to S_1 S_2$ is not a regular production, we need to use another grammar $G_\times = (N_1 \cup N_2, T_1 \cup T_2, P_1' \cup P_2, S_1)$, where the difference between P_1' and P_1 lies in that instead of productions in the form $A \to u, u \in T^*$ in P_1' will exist production of the form $A \to uS_2$.

Iteration. Let $G_* = (N_1 \cup \{S\}, T_1, P, S)$.

In the case of grammars of type 2 let $P = P_1 \cup \{S \to S_1 S,\ S \to \varepsilon\}$. Then G_* also is of type 2.

In the case of grammars of type 3, as in the case of product, we will change the productions, that is $P = P_1' \cup \{S \to S_1,\ S \to \varepsilon\}$, where the difference between P_1' and P_1 lies in that for each $A \to u$ $(u \in T^*)$ will be replaced by $A \to uS$, and the others will be not changed. Then G_* also will be of type 3.

The productions given in the case of type 2 are not valid for $i = 0, 1$, because when applying production $S \to S_1 S$ we can get the derivations of type $S \overset{*}{\Longrightarrow} S_1 S_1$, $S_1 \overset{*}{\Longrightarrow} \alpha_1 \beta_1$, $S_1 \overset{*}{\Longrightarrow} \alpha_2 \beta_2$, where $\beta_1 \alpha_2$ can be a left-hand side of a production. In this case, replacing $\beta_1 \alpha_2$ by its right-hand side in derivation $S \overset{*}{\Longrightarrow} \alpha_1 \beta_1 \alpha_2 \beta_2$, we can generate a word which is not in the iterated language. To avoid such situations, first let us assume that the language is in normal form, i.e. the left-hand side of productions does not contain terminals (*see* page 20), second we introduce a new nonterminal S', so the set of nonterminals now is $N_1 \cup \{S, S'\}$, and the productions are the following:

$$P = P_1 \cup \{S \to \varepsilon,\ S \to S_1 S'\} \cup \{aS' \to aS \mid a \in T_1\}\ .$$

Now we can avoid situations in which the left-hand side of a production can extend over the limits of words in a derivation because of the iteration. The above derivations can be used only by beginning with $S \Longrightarrow S_1 S'$ and getting derivation $S \overset{*}{\Longrightarrow} \alpha_1 \beta_1 S'$. Here we can not replace S' unless the last symbol in β_1 is a terminal symbol, and only after using a production of the form $aS' \to aS$.

It is easy to show that $L(G_*) = L(G_1)^*$ for each type. ■

Exercises

1.1-1 Give a grammar which generates language $L = \left\{uu^{-1} \mid u \in \{a, b\}^*\right\}$ and determine its type.

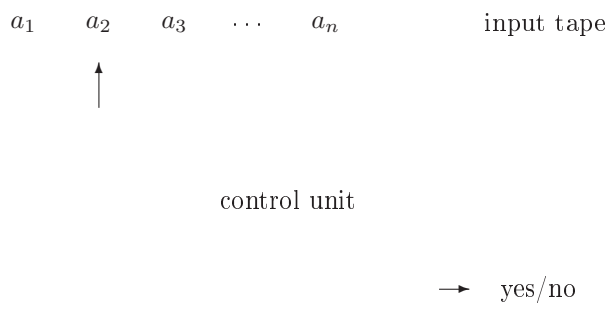


Figure 1.1. Finite automaton.

1.1-2 Let $G = (N, T, P, S)$ be an extended context-free grammar, where
$N = \{S, A, C, D\}$, $T = \{a, b, c, d, e\}$,
$P = \{S \to abCADe,\ C \to cC,\ C \to \varepsilon,\ D \to dD,\ D \to \varepsilon,\ A \to \varepsilon,\ A \to dDcCA\}$.
Give an equivalent context-free grammar.

1.1-3 Show that Σ^* and Σ^+ are regular languages over arbitrary alphabet Σ.

1.1-4 Give a grammar to generate language $L = \{u \in \{0,1\}^* \mid n_0(u) = n_1(u)\}$, where $n_0(u)$ represents the number of 0's in word u and $n_1(u)$ the number of 1's.

1.1-5 Give a grammar to generate all natural numbers.

1.1-6 Give a grammar to generate the following languages, respectively:
$L_1 = \{a^n b^m c^p \mid n \geq 1, m \geq 1, p \geq 1\}$,
$L_2 = \{a^{2n} \mid n \geq 1\}$,
$L_3 = \{a^n b^m \mid n \geq 0, m \geq 0\ \}$,
$L_4 = \{a^n b^m \mid n \geq m \geq 1\}$.

1.1-7 Let $G = (N, T, P, S)$ be an extended grammar, where $N = \{S, A, B, C\}$, $T = \{a\}$ and P contains the productions:

$$S \to BAB,\ BA \to BC,\ CA \to AAC,\ CB \to AAB,\ A \to a,\ B \to \varepsilon\ .$$

Determine the type of this grammar. Give an equivalent, not extended grammar with the same type. What language it generates?

1.2. Finite automata and regular languages

Finite automata are computing models with input tape and a finite set of states (Fig. 1.1). Among the states some are called initial and some final. At the beginning the automaton read the first letter of the input word written on the input tape. Beginning with an initial state, the automaton read the letters of the input word one after another while change its states, and when after reading the last input letter the current state is a final one, we say that the automaton accepts the given word. The set of words accepted by such an automaton is called the language accepted (recognized) by the automaton.

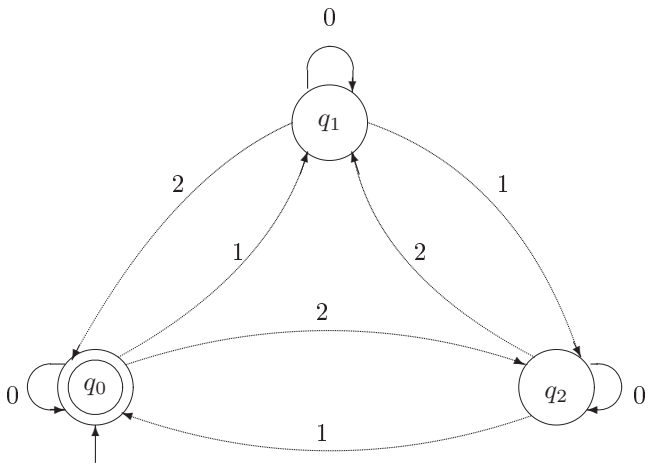


Figure 1.2. The finite automaton of Example 1.9.

Definition 1.9 *A **nondeterministic finite automaton** (NFA) is a system* $\mathrm{A} = (Q, \Sigma, E, I, F)$*, where*

- Q *is a finite, nonempty set of **states**,*
- Σ *is the **input alphabet**,*
- E *is the set of **transitions** (or of edges), where* $E \subseteq Q \times \Sigma \times Q$*,*
- $I \subseteq Q$ *is the set of **initial states**,*
- $F \subseteq Q$ *is the set of **final states**.*

An NFA is in fact a directed, labelled graph, whose vertices are the states and there is a (directed) edge labelled with a from vertex p to vertex q if $(p, a, q) \in E$. Among vertices some are initial and some final states. Initial states are marked by a small arrow entering the corresponding vertex, while the final states are marked with double circles. If two vertices are joined by two edges with the same direction then these can be replaced by only one edge labelled with two letters. This graph can be called a transition graph.

Example 1.9 Let $\mathrm{A} = (Q, \Sigma, E, I, F)$, where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, 2\}$,
$E = \ (q_0, 0, q_0), (q_0, 1, q_1), (q_0, 2, q_2),$
$(q_1, 0, q_1), (q_1, 1, q_2), (q_1, 2, q_0),$
$(q_2, 0, q_2), (q_2, 1, q_0), (q_2, 2, q_1)$
$I = \{q_0\}$, $F = \{q_0\}$.
The automaton can be seen in Fig. 1.2.

In the case of an edge (p, a, q) vertex p is the start-vertex, q the end-vertex and a the label. Now define the notion of the ***walk*** as in the case of graphs. A sequence

$$(q_0, a_1, q_1), (q_1, a_2, q_2), \ldots, (q_{n-2}, a_{n-1}, q_{n-1}), (q_{n-1}, a_n, q_n)$$

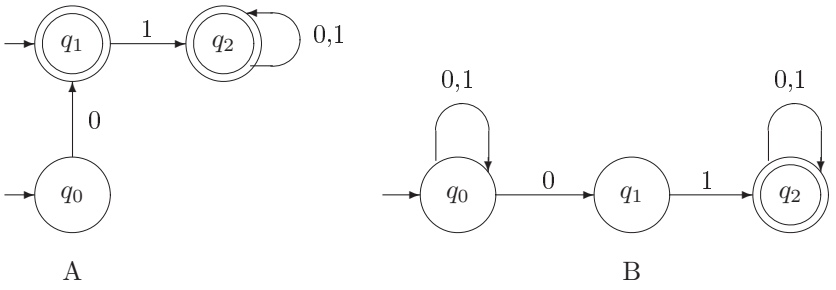


Figure 1.3. Nondeterministic finite automata.

δ	0	1
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$

A

δ	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$

B

Figure 1.4. Transition tables of the NFA in Fig. 1.3.

of edges of a NFA is a walk with the label $a_1a_2\dots a_n$. If $n = 0$ then $q_0 = q_n$ and $a_1a_2\dots a_n = \varepsilon$. Such a walk is called an ***empty walk***. For a walk the notation

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n \ ,$$

will be used, or if $w = a_1a_2\dots a_n$ then we write shortly $q_0 \xrightarrow{w} q_n$. Here q_0 is the start-vertex and q_n the end-vertex of the walk. The states in a walk are not necessary distinct.

A walk is ***productive*** if its start-vertex is an initial state and its end-vertex is a final state. We say that an NFA ***accepts*** or ***recognizes*** a word if this word is the label of a productive walk. The empty word ε is accepted by an NFA if there is an empty productive walk, i.e. there is an initial state which is also a final state.

The set of words accepted by an NFA will be called the language accepted by this NFA. The ***language accepted*** or ***recognized*** by NFA A is

$$L(\mathrm{A}) = \left\{ w \in \Sigma^* \mid \exists p \in I,\ \exists q \in F,\ \exists p \xrightarrow{w} q \right\} .$$

The NFA A_1 and A_2 are ***equivalent*** if $L(\mathrm{A}_1) = L(\mathrm{A}_2)$.

Sometimes it is useful the following ***transition function***:

$$\delta : Q \times \Sigma \to P(Q), \quad \delta(p, a) = \{q \in Q \mid (p, a, q) \in E\} \ .$$

This function associate to a state p and input letter a the set of states in which the automaton can go if its current state is p and the head is on input letter a.

Denote by $|H|$ the cardinal (the number of elements) of H.[2] An NFA is a ***deterministic finite automaton*** (DFA) if

$$|I| = 1 \text{ and } |\delta(q,a)| \leq 1,\ \forall q \in Q,\ \forall a \in \Sigma\ .$$

In Fig. 1.2 a DFA can be seen.

Condition $|\delta(q,a)| \leq 1$ can be replaced by

$$(p,a,q) \in E,\ (p,a,r) \in E \implies q = r\ , \forall p,q,r \in Q, \forall a \in \Sigma\ .$$

If for a DFA $|\delta(q,a)| = 1$ for each state $q \in Q$ and for each letter $a \in \Sigma$ then it is called a ***complete DFA***.

Every DFA can be transformed in a complete DFA by introducing a new state, which can be called a snare state. Let $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ be a DFA. An equivalent and complete DFA will be $\mathrm{A}' = (Q \cup \{s\}, \Sigma, E', \{q_0\}, F)$, where s is the new state and $E' = E \cup \big\{(p,a,s) \mid \delta(p,a) = \emptyset, p \in Q, a \in \Sigma\big\} \cup \big\{(s,a,s) \mid a \in \Sigma\big\}$. It is easy to see that $L(\mathrm{A}) = L(\mathrm{A}')$.

Using the transition function we can easily define the transition table. The rows of this table are indexed by the elements of Q, its columns by the elements of Σ. At the intersection of row $q \in Q$ and column $a \in \Sigma$ we put $\delta(q,a)$. In the case of Fig. 1.2, the transition table is:

δ	0	1	2
q_0	$\{q_0\}$	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_1\}$	$\{q_2\}$	$\{q_0\}$
q_2	$\{q_2\}$	$\{q_0\}$	$\{q_1\}$

The NFA in 1.3 are not deterministic: the first (automaton A) has two initial states, the second (automaton B) has two transitions with 0 from state q_0 (to states q_0 and q_1). The transition table of these two automata are in Fig. 1.4. $L(\mathrm{A})$ is set of words over $\Sigma = \{0,1\}$ which do not begin with two zeroes (of course ε is in language), $L(\mathrm{B})$ is the set of words which contain 01 as a subword.

Eliminating inaccessible states Let $\mathrm{A} = (Q, \Sigma, E, I, F)$ be a finite automaton. A state is *accessible* if it is on a walk which starts by an initial state. The following algorithm determines the inaccessible states building a sequence U_0, U_1, U_2, ... of sets, where U_0 is the set of initial states, and for any $i \geq 1$ U_i is the set of accessible states, which are at distance at most i from an initial state.

[2] The same notation is used for the cardinal of a set and length of a word, but this is no matter of confusion because for word we use lowercase letters and for set capital letters. The only exception is $\delta(q,a)$, but this could not be confused with a word.

INACCESSIBLE-STATES(A)

```
1  U_0 ← I
2  i ← 0
3  repeat
4     i ← i + 1
5     for all q ∈ U_{i-1}
6           do for all a ∈ Σ
7                 do U_i ← U_{i-1} ∪ δ(q, a)
8  until U_i = U_{i-1}
9  U ← Q \ U_i
10 return U
```

The inaccessible states of the automaton can be eliminated without changing the accepted language.

If $|Q| = n$ and $|\Sigma| = m$ then the running time of the algorithm (the number of steps) in the worst case is $O(n^2m)$, because the number of steps in the two embedded loops is at most nm and in the loop **repeat** at most n.

Set U has the property that $L(\mathrm{A}) \neq \emptyset$ if and only if $U \cap F \neq \emptyset$. The above algorithm can be extended by inserting the $U \cap F \neq \emptyset$ condition to decide if language $L(\mathrm{A})$ is or not empty.

Eliminating nonproductive states Let $\mathrm{A} = (Q, \Sigma, E, I, F)$ be a finite automaton. A state is *productive* if it is on a walk which ends in a terminal state. For finding the productive states the following algorithm uses the function δ^{-1}:

$$\delta^{-1} : Q \times \Sigma \to \mathcal{P}(Q), \quad \delta^{-1}(p, a) = \{q \mid (q, a, p) \in E\}.$$

This function for a state p and a letter a gives the set of all states from which using this letter a the automaton can go into the state p.

NONPRODUCTIVE-STATES(A)

```
1  V_0 ← F
2  i ← 0
3  repeat
4     i ← i + 1
5     for all p ∈ V_{i-1} do
6           do for all a ∈ Σ
7                 do V_i ← V_{i-1} ∪ δ^{-1}(p, a)
8  until V_i = V_{i-1}
9  V ← Q \ V_i
10 return V
```

The nonproductive states of the automaton can be eliminated without changing the accepted language.

If n is the number of states, m the number of letters in the alphabet, then

the running time of the algorithm is also $O(n^2m)$ as in the case of the algorithm INACCESSIBLE-STATES.

The set V given by the algorithm has the property that $L(\mathrm{A}) \neq \emptyset$ if and only if $V \cap I \neq \emptyset$. So, by a little modification it can be used to decide if language $L(\mathrm{A})$ is or not empty.

1.2.1. Transforming nondeterministic finite automata in deterministic finite automata

As follows we will show that any NFA can be transformed in an equivalent DFA.

Theorem 1.10 *For any NFA one may construct an equivalent DFA.*

Proof Let $\mathrm{A} = (Q, \Sigma, E, I, F)$ be an NFA. Define a DFA $\overline{\mathrm{A}} = (\overline{Q}, \Sigma, \overline{E}, \overline{I}, \overline{F})$, where

- $\overline{Q} = \mathcal{P}(Q) \setminus \emptyset$,
- edges of $\overline{E}$ are those triplets (S, a, R) for which $R, S \in \overline{Q}$ are not empty, $a \in \Sigma$ and $R = \bigcup_{p \in S} \delta(p, a)$,
- $\overline{I} = \{I\}$,
- $\overline{F} = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$.

We prove that $L(\mathrm{A}) = L(\overline{\mathrm{A}})$.

a) First prove that $L(\mathrm{A}) \subseteq L(\overline{\mathrm{A}})$. Let $w = a_1 a_2 \dots a_k \in L(\mathrm{A})$. Then there exists a walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} q_{k-1} \xrightarrow{a_k} q_k, \quad q_0 \in I,\ q_k \in F.$$

Using the transition function $\overline{\delta}$ of NFA $\overline{\mathrm{A}}$ we construct the sets $S_0 = \{q_0\}$, $\overline{\delta}(S_0, a_1) = S_1$, $\dots \overline{\delta}(S_{k-1}, a_k) = S_k$. Then $q_1 \in S_1, \dots, q_k \in S_k$ and since $q_k \in F$ we get $S_k \cap F \neq \emptyset$, so $S_k \in \overline{F}$. Thus, there exists a walk

$$S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} S_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} S_{k-1} \xrightarrow{a_k} S_k, \quad S_0 \subseteq I,\ S_k \in \overline{F}.$$

There are sets $S'_0, \dots, S'_k$ for which $S'_0 = I$, and for $i = 0, 1, \dots, k$ we have $S_i \subseteq S'_i$, and

$$S'_0 \xrightarrow{a_1} S'_1 \xrightarrow{a_2} S'_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} S'_{k-1} \xrightarrow{a_k} S'_k$$

is a productive walk. Therefore $w \in L(\overline{\mathrm{A}})$. That is $L(\mathrm{A}) \subseteq L(\overline{\mathrm{A}})$.

b) Now we show that $L(\overline{\mathrm{A}}) \subseteq L(\mathrm{A})$. Let $w = a_1 a_2 \dots a_k \in L(\overline{\mathrm{A}})$. Then there is a walk

$$\overline{q_0} \xrightarrow{a_1} \overline{q_1} \xrightarrow{a_2} \overline{q_2} \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} \overline{q}_{k-1} \xrightarrow{a_k} \overline{q}_k, \quad \overline{q}_0 \in \overline{I},\ \overline{q}_k \in \overline{F}.$$

Using the definition of $\overline{F}$ we have $\overline{q}_k \cap F \neq \emptyset$, i.e. there exists $q_k \in \overline{q}_k \cap F$, that is by the definitions of $q_k \in F$ and $\overline{q}_k$ there is q_{k-1} such that $(q_{k-1}, a_k, q_k) \in E$. Similarly, there are the states $q_{k-2}, \dots, q_1, q_0$ such that $(q_{k-2}, a_k, q_{k-1}) \in E, \dots, (q_0, a_1, q_1) \in E$, where $q_0 \in \overline{q}_0 = I$, thus, there is a walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} q_{k-1} \xrightarrow{a_k} q_k, \quad q_0 \in I,\ q_k \in F,$$

so $L(\overline{\mathrm{A}}) \subseteq L(\mathrm{A})$. ■

In constructing DFA we can use the corresponding transition function $\overline{\delta}$:

$$\overline{\delta}(\overline{q}, a) = \left\{ \bigcup_{q \in \overline{q}} \delta(q, a) \right\},\ \forall \overline{q} \in \overline{Q}, \forall a \in \Sigma.$$

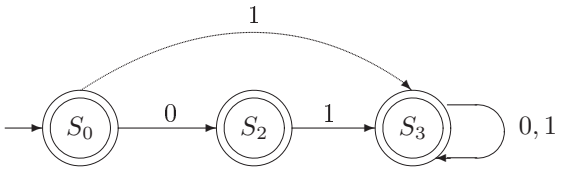


Figure 1.5. The equivalent DFA with NFA A in Fig. 1.3.

The empty set was excluded from the states, so we used here $\emptyset$ instead of $\{\emptyset\}$.

Example 1.10 Apply Theorem 1.10 to transform NFA A in Fig. 1.3. Introduce the following notation for the states of the DFA:

$$S_0 := \{q_0, q_1\}, \quad S_1 := \{q_0\}, \quad S_2 := \{q_1\}, \quad S_3 := \{q_2\},$$
$$S_4 := \{q_0, q_2\}, \quad S_5 := \{q_1, q_2\}, \quad S_6 := \{q_0, q_1, q_2\}\ ,$$

where S_0 is the initial state. Using the transition function we get the transition table:

$\overline{\delta}$	0	1
S_0	$\{S_2\}$	$\{S_3\}$
S_1	$\{S_2\}$	$\emptyset$
S_2	$\emptyset$	$\{S_3\}$
S_3	$\{S_3\}$	$\{S_3\}$
S_4	$\{S_5\}$	$\{S_3\}$
S_5	$\{S_3\}$	$\{S_3\}$
S_6	$\{S_5\}$	$\{S_3\}$

This automaton contains many inaccessible states. By algorithm INACCESSIBLE-STATES we determine the accessible states of DFA:

$$U_0 = \{S_0\}, \quad U_1 = \{S_0, S_2, S_3\}, \quad U_2 = \{S_0, S_2, S_3\} = U_1 = U.$$

Initial state S_0 is also a final state. States S_2 and S_3 are final states. States S_1, S_4, S_5, S_6 are inaccessible and can be removed from the DFA. The transition table of the resulted DFA is

$\overline{\delta}$	0	1
S_0	$\{S_2\}$	$\{S_3\}$
S_2	$\emptyset$	$\{S_3\}$
S_3	$\{S_3\}$	$\{S_3\}$

The corresponding transition graph is in Fig. 1.5.

The algorithm given in Theorem 1.10 can be simplified. It is not necessary to consider all subset of the set of states of NFA. The states of DFA $\overline{\mathrm{A}}$ can be obtained successively. Begin with the state $\overline{q}_0 = I$ and determine the states $\overline{\delta}(\overline{q}_0, a)$ for all $a \in \Sigma$. For the newly obtained states we determine the states accessible from them. This can be continued until no new states arise.

In our previous example $\overline{q}_0 := \{q_0, q_1\}$ is the initial state. From this we get

$\overline{\delta}(\overline{q}_0, 0) = \{\overline{q_1}\}$, where $\overline{q}_1 := \{q_1\}$, $\quad \overline{\delta}(\overline{q}_0, 1) = \{\overline{q}_2\}$, where $\overline{q}_2 := \{q_2\}$,
$\overline{\delta}(\overline{q}_1, 0) = \emptyset$, $\quad \overline{\delta}(\overline{q}_1, 1) = \{\overline{q}_2\}$,
$\overline{\delta}(\overline{q}_2, 0) = \{\overline{q}_2\}$, $\quad \overline{\delta}(\overline{q}_2, 1) = \{\overline{q}_2\}$.

The transition table is

$\overline{\delta}$	0	1
$\overline{q}_0$	$\{\overline{q}_1\}$	$\{\overline{q}_2\}$
$\overline{q}_1$	$\emptyset$	$\{\overline{q}_2\}$
$\overline{q}_2$	$\{\overline{q}_2\}$	$\{\overline{q}_2\}$

which is the same (excepted the notation) as before.

The next algorithm will construct for an NFA $\mathrm{A} = (Q, \Sigma, E, I, F)$ the transition table M of the equivalent DFA $\overline{\mathrm{A}} = (\overline{Q}, \Sigma, \overline{E}, \overline{I}, \overline{F})$, but without to determine the final states (which can easily be included). Value of $\textsc{IsIn}(\overline{q}, \overline{Q})$ in the algorithm is true if state $\overline{q}$ is already in $\overline{Q}$ and is false otherwise. Let $a_1, a_2, \ldots, a_m$ be an ordered list of the letters of Σ.

NFA-DFA(A)

```
1  q̄_0 ← I
2  Q̄ ← {q̄_0}
3  i ← 0                                          ▷ i counts the rows.
4  k ← 0                                          ▷ k counts the states.
5  repeat
6        for j = 1, 2, ..., m                     ▷ j counts the columns.
7            do q̄ ← ⋃_{p∈q̄_i} δ(p, a_j)
8               if q̄ ≠ ∅
9                  then if IsIn(q̄, Q̄)
10                           then M[i, j] ← {q̄}
11                           else k ← k + 1
12                                q̄_k ← q̄
13                                M[i, j] ← {q̄_k}
14                                Q̄ ← Q̄ ∪ {q̄_k}
15                 else M[i, j] ← ∅
16       i ← i + 1
17 until i = k + 1
18 return transition table M of Ā
```

Since loop **repeat** is executed as many times as the number of states of new automaton, in worst case the running time can be exponential, because, if the number of states in NFA is n, then DFA can have even $2^n - 1$ states. (The number of subsets of a set of n elements is 2^n, including the empty set.)

Theorem 1.10 will have it that to any NFA one may construct an equivalent DFA. Conversely, any DFA is also an NFA by definition. So, the nondeterministic finite automata accepts the same class of languages as the deterministic finite automata.

1.2.2. Equivalence of deterministic finite automata

In this subsection we will use complete deterministic finite automata only. In this case $\delta(q,a)$ has a single element. In formulae, sometimes, instead of set $\delta(q,a)$ we will use its single element. We introduce for a set $A=\{a\}$ the function $elem(A)$ which give us the single element of set A, so $elem(A)=a$. Using walks which begin with the initial state and have the same label in two DFA's we can determine the equivalence of these DFA's. If only one of these walks ends in a final state, then they could not be equivalent.

Consider two DFA's over the same alphabet $A=(Q,\Sigma,E,\{q_0\},F)$ and $A'=(Q',\Sigma,E',\{q_0'\},F')$. We are interested to determine if they are or not equivalent. We construct a table with elements of form (q,q'), where $q\in Q$ and $q'\in Q'$. Beginning with the second column of the table, we associate a column to each letter of the alphabet Σ. If the first element of the ith row is (q,q') then at the cross of ith row and the column associated to letter a will be the pair $\Big(elem\big(\delta(q,a)\big),elem\big(\delta'(q',a)\big)\Big)$.

	$\ldots \quad a \quad \ldots$
$\ldots$	$\ldots$
(q,q')	$elem\ \delta(q,a)\ ,elem\ \delta'(q',a)$
$\ldots$	$\ldots$

In the first column of the first row we put (q_0,q_0') and complete the first row using the above method. If in the first row in any column there occur a pair of states from which one is a final state and the other not then the algorithm ends, the two automata are ***not equivalent***. If there is no such a pair of states, every new pair is written in the first column. The algorithm continues with the next unfilled row. If no new pair of states occurs in the table and for each pair both of states are final or both are not, then the algorithm ends and the two DFA are ***equivalent***.

If $|Q|=n$, $|Q'|=n'$ and $|\Sigma|=m$ then taking into account that in worst case loop **repeat** is executed nn' times, loop **for** m times, the running time of the algorithm in worst case will be $O(nn'm)$, or if $n=n'$ then $O(n^2m)$.

Our algorithm was described to determine the equivalence of two complete DFA's. If we have to determine the equivalence of two NFA's, first we transform them into complete DFA's and after this we can apply the above algorithm.

DFA-EQUIVALENCE(A, A′)

1 write in the first column of the first row the pair (q_0,q_0')
2 $i\leftarrow 0$

3 **repeat**
4 $i \leftarrow i+1$
5 let (q, q') be the pair in the first column of the ith row
6 **for** all $a \in \Sigma$
7 **do** write in the column associated to a in the ith row
the pair $\Big(elem\big(\delta(q,a)\big), elem\big(\delta'(q',a)\big)\Big)$
8 **if** one state in $\Big(elem\big(\delta(q,a)\big), elem\big(\delta'(q',a)\big)\Big)$ is final and the other not
9 **then return** NO
10 **else** write pair $\Big(elem\big(\delta(q,a)\big), elem\big(\delta'(q',a)\big)\Big)$ in the next empty row
of the first column, if not occurred already in the first column
11 **until** the first element of $(i+1)$th row becomes empty
12 **return** YES

Example 1.11 Determine if the two DFA's in Fig. 1.6 are equivalent or not. The algorithm gives the table

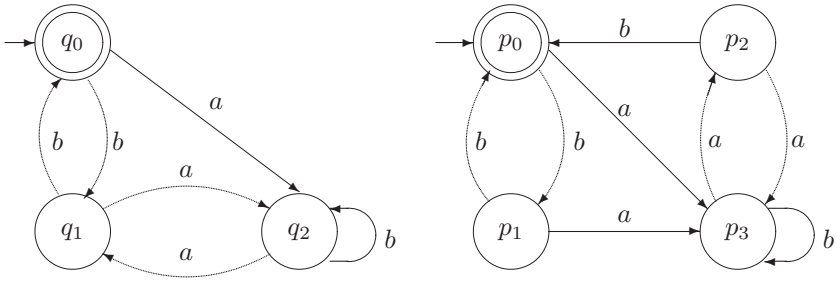


Figure 1.6. Equivalent DFA's (Example 1.11).

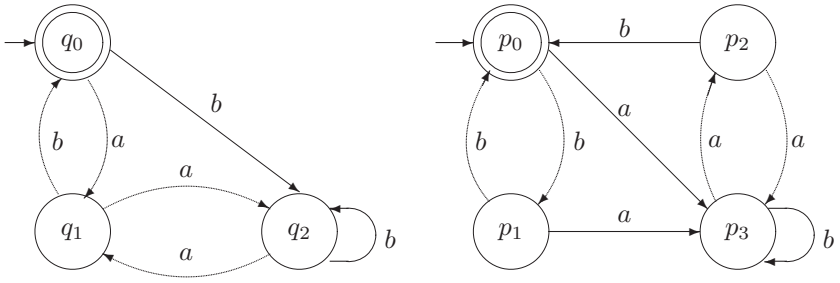


Figure 1.7. Non equivalent DFA's (Example 1.12).

	a	b
(q_0, p_0)	(q_2, p_3)	(q_1, p_1)
(q_2, p_3)	(q_1, p_2)	(q_2, p_3)
(q_1, p_1)	(q_2, p_3)	(q_0, p_0)
(q_1, p_2)	(q_2, p_3)	(q_0, p_0)

The two DFA's are equivalent because all possible pairs of states are considered and in every pair both states are final or both are not final.

Example 1.12 The table of the two DFA's in Fig. 1.7 is:

	a	b
(q_0, p_0)	(q_1, p_3)	(q_2, p_1)
(q_1, p_3)	(q_2, p_2)	(q_0, p_3)
(q_2, p_1)		
(q_2, p_2)		

These two DFA's are not equivalent, because in the last column of the second row in the pair (q_0, p_3) the first state is final and the second not.

1.2.3. Equivalence of finite automata and regular languages

We have seen that NFA's accept the same class of languages as DFA's. The following theorem states that this class is that of regular languages.

Theorem 1.11 *If L is a language accepted by a DFA, then one may construct a regular grammar which generates language L.*

Proof Let $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ be the DFA accepting language L, that is $L = L(\mathrm{A})$. Define the regular grammar $G = (Q, \Sigma, P, q_0)$ with the productions:

- If $(p, a, q) \in E$ for $p, q \in Q$ and $a \in \Sigma$, then put production $p \to aq$ in P.
- If $(p, a, q) \in E$ and $q \in F$, then put also production $p \to a$ in P.

Prove that $L(G) = L(\mathrm{A}) \setminus \{\varepsilon\}$.

Let $u = a_1 a_2 \dots a_n \in L(\mathrm{A})$ and $u \neq \varepsilon$. Thus, since A accepts word u, there is a walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n, \quad q_n \in F.$$

Then there are in P the productions

$$q_0 \to a_1 q_1, \quad q_1 \to a_2 q_2, \quad \dots, \quad q_{n-2} \to a_{n-1} q_{n-1}, \quad q_{n-1} \to a_n$$

(in the right-hand side of the last production q_n does not occur, because $q_n \in F$), so there is the derivation

$$q_0 \Longrightarrow a_1 q_1 \Longrightarrow a_1 a_2 q_2 \Longrightarrow \dots \Longrightarrow a_1 a_2 \dots a_{n-1} q_{n-1} \Longrightarrow a_1 a_2 \dots a_n.$$

Therefore, $u \in L(G)$.

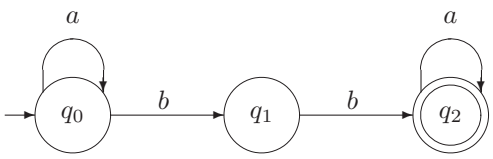


Figure 1.8. DFA of the Example 1.13.

Conversely, let $u = a_1 a_2 \dots a_n \in L(G)$ and $u \neq \varepsilon$. Then there exists a derivation

$$q_0 \Longrightarrow a_1 q_1 \Longrightarrow a_1 a_2 q_2 \Longrightarrow \dots \Longrightarrow a_1 a_2 \dots a_{n-1} q_{n-1} \Longrightarrow a_1 a_2 \dots a_n,$$

in which productions

$$q_0 \to a_1 q_1, \quad q_1 \to a_2 q_2, \quad \dots, \quad q_{n-2} \to a_{n-1} q_{n-1}, \quad q_{n-1} \to a_n$$

were used, which by definition means that in DFA A there is a walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n,$$

and since q_n is a final state, $u \in L(\mathrm{A}) \setminus \{\varepsilon\}$.

If the DFA accepts also the empty word ε, then in the above grammar we introduce a new start symbol q_0' instead of q_0, consider the new production $q_0' \to \varepsilon$ and for each production $q_0 \to \alpha$ introduce also $q_0' \to \alpha$. ■

Example 1.13 Let $\mathrm{A} = (\{q_0, q_1, q_2\}, \{a, b\}, E, \{q_0\}, \{q_2\})$ be a DFA, where $E = \{(q_0, a, q_0), (q_0, b, q_1), (q_1, b, q_2), (q_2, a, q_2)\}$. The corresponding transition table is

δ	a	b
q_0	$\{q_0\}$	$\{q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\emptyset$

The transition graph of A is in Fig. 1.8. By Theorem 1.11 we define regular grammar $G = (\{q_0, q_1, q_2\}, \{a, b\}, P, q_0)$ with the productions in P

$$q_0 \to aq_0 \mid bq_1, \qquad q_1 \to bq_2 \mid b, \qquad q_2 \to aq_2 \mid a.$$

One may prove that $L(\mathrm{A}) = \{a^m bba^n \mid m \geq 0, n \geq 0\}$.

The method described in the proof of Theorem 1.11 easily can be given as an algorithm. The productions of regular grammar $G = (Q, \Sigma, P, q_0)$ obtained from the DFA $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ can be determined by the following algorithm.

REGULAR-GRAMMAR-FROM-DFA(A)

```
1  P ← ∅
2  for all p ∈ Q
3      do for all a ∈ Σ
4             do for all q ∈ Q
5                    do if (p, a, q) ∈ E
6                          then P ← P ∪ {p → aq}
7                               if q ∈ F
8                                  then P ← P ∪ {p → a}
9  if q_0 ∈ F
10    then P ← P ∪ {q_0 → ε}
11 return G
```

It is easy to see that the running time of the algorithm is $\Theta(n^2m)$, if the number of states is n and the number of letter in alphabet is m. In lines 2–4 we can consider only one loop, if we use the elements of E. Then the worst case running time is $\Theta(p)$, where p is the number of transitions of DFA. This is also $O(n^2m)$, since all transitions are possible. This algorithm is:

REGULAR-GRAMMAR-FROM-DFA'(A)

```
1  P ← ∅
2  for all (p, a, q) ∈ E
3      do P ← P ∪ {p → aq}
4         if q ∈ F
5            then P ← P ∪ {p → a}
6  if q_0 ∈ F
7      then P ← P ∪ {q_0 → ε}
8  return G
```

Theorem 1.12 *If $L = L(G)$ is a regular language, then one may construct an NFA that accepts language L.*

Proof Let $G = (N, T, P, S)$ be the grammar which generates language L. Define NFA $\mathrm{A} = (Q, T, E, \{S\}, F)$:

- $Q = N \cup \{Z\}$, where $Z \notin N \cup T$ (i.e. Z is a new symbol),
- For every production $A \to aB$, define transition (A, a, B) in E.
- For every production $A \to a$, define transition (A, a, Z) in E.
- $F = \begin{cases} \{Z\} & \text{if production } S \to \varepsilon \text{ does not occur in } G, \\ \{Z, S\} & \text{if production } S \to \varepsilon \text{ occurs in } G. \end{cases}$

Prove that $L(G) = L(A)$.
Let $u = a_1a_2 \dots a_n \in L(G)$, $u \neq \varepsilon$. Then there is in G a derivation of word u:

$$S \Longrightarrow a_1A_1 \Longrightarrow a_1a_2A_2 \Longrightarrow \dots \Longrightarrow a_1a_2 \dots a_{n-1}A_{n-1} \Longrightarrow a_1a_2 \dots a_n.$$

This derivation is based on productions

$$S \to a_1A_1, \;\; A_1 \to a_2A_2, \;\; \dots, \;\; A_{n-2} \to a_{n-1}A_{n-1}, \;\; A_{n-1} \to a_n.$$

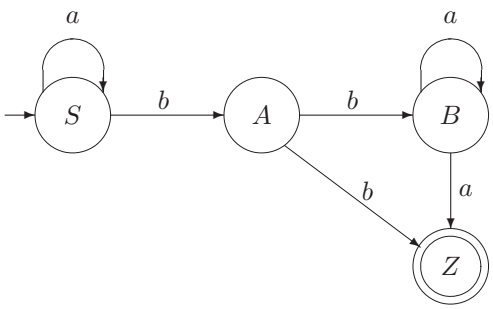


Figure 1.9. NFA associated to grammar in Example 1.14.

Then, by the definition of the transitions of NFA A there exists a walk

$$S \xrightarrow{a_1} A_1 \xrightarrow{a_2} A_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} A_{n-1} \xrightarrow{a_n} Z, \quad Z \in F.$$

Thus, $u \in L(\mathrm{A})$. If $\varepsilon \in L(G)$, there is production $S \to \varepsilon$, but in this case the initial state is also a final one, so $\varepsilon \in L(A)$. Therefore, $L(G) \subseteq L(A)$.

Let now $u = a_1 a_2 \dots a_n \in L(\mathrm{A})$. Then there exists a walk

$$S \xrightarrow{a_1} A_1 \xrightarrow{a_2} A_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} A_{n-1} \xrightarrow{a_n} Z, \quad Z \in F.$$

If u is the empty word, then instead of Z we have in the above formula S, which also is a final state. In other cases only Z can be as last symbol. Thus, in G there exist the productions

$$S \to a_1 A_1, \quad A_1 \to a_2 A_2, \quad \dots, \quad A_{n-2} \to a_{n-1} A_{n-1}, \quad A_{n-1} \to a_n \ ,$$

and there is the derivation

$$S \Longrightarrow a_1 A_1 \Longrightarrow a_1 a_2 A_2 \Longrightarrow \dots \Longrightarrow a_1 a_2 \dots a_{n-1} A_{n-1} \Longrightarrow a_1 a_2 \dots a_n,$$

thus, $u \in L(G)$ and therefore $L(A) \subseteq L(G)$. ∎

Example 1.14 Let $G = (\{S, A, B\}, \{a, b\}, \{S \to aS, S \to bA, A \to bB, A \to b, B \to aB, B \to a\}, S)$ be a regular grammar. The NFA associated is $A = (\{S, A, B, Z\}, \{a, b\}, E, S, \{Z\})$, where $E = (S, a, S), (S, b, A), (A, b, B), (A, b, Z), (B, a, B), (B, a, Z)$. The corresponding transition table is

δ	a	b
S	$\{S\}$	$\{A\}$
A	$\emptyset$	$\{B, Z\}$
B	$\{B, Z\}$	$\emptyset$
E	$\emptyset$	$\emptyset$

The transition graph is in Fig. 1.9. This NFA can be simplified, states B and Z can be contracted in one final state.

Using the above theorem we define an algorithm which associate an NFA A = $(Q, T, E, \{S\}, F)$ to a regular grammar $G = (N, T, P, S)$.

NFA-FROM-REGULAR-GRAMMAR(A)

```
1  E ← ∅
2  Q ← N ∪ {Z}
3  for all A ∈ N
4      do for all a ∈ T
5             do if (A → a) ∈ P
6                    then E ← E ∪ {(A, a, Z)}
7                for all B ∈ N
8                    do if (A → aB) ∈ P
9                           then E ← E ∪ {(A, a, B)}
10 if (S → ε) ∉ P
11    then F ← {Z}
12    else F ← {Z, S}
13 return A
```

As in the case of algorithm REGULAR-GRAMMAR-FROM-DFA, the running time is $\Theta(n^2m)$, where n is number of nonterminals and m the number of terminals. Loops in lines 3, 4 and 7 can be replaced by only one, which uses productions. The running time in this case is better and is equal to $\Theta(p)$, if p is the number of productions. This algorithm is:

NFA-FROM-REGULAR-GRAMMAR'(A)

```
1  E ← ∅
2  Q ← N ∪ {Z}
3  for all (A → u) ∈ P
4      do if u = a
5             then E ← E ∪ {(A, a, Z)}
6          if u = aB
7             then E ← E ∪ {(A, a, B)}
8  if (S → ε) ∉ P
9     then F ← {Z}
10    else F ← {Z, S}
11 return A
```

From theorems 1.10, 1.11 and 1.12 results that the class of regular languages coincides with the class of languages accepted by NFA's and also with class of languages accepted by DFA's. The result of these three theorems is illustrated in Fig. 1.10 and can be summarised also in the following theorem.

Theorem 1.13 *The following three class of languages are the same:*

- *the class of regular languages,*
- *the class of languages accepted by DFA's,*
- *the class of languages accepted by NFA's.*

Operation on regular languages It is known (see Theorem 1.8) that the set $\mathcal{L}_3$ of regular languages is closed under the regular operations, that is if L_1, L_2 are

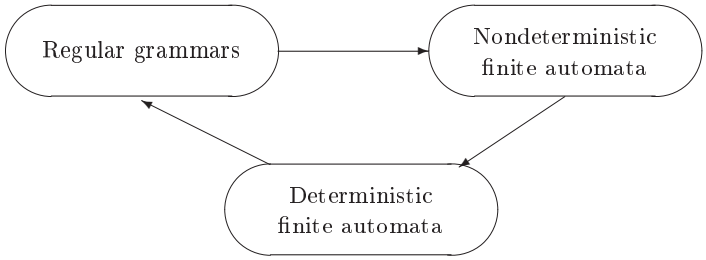


Figure 1.10. Relations between regular grammars and finite automata. To any regular grammar one may construct an NFA which accepts the language generated by that grammar. Any NFA can be transformed in an equivalent DFA. To any DFA one may construct a regular grammar which generates the language accepted by that DFA.

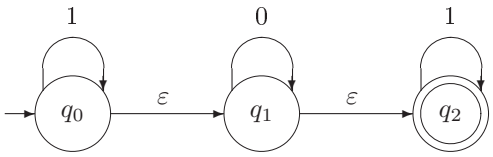


Figure 1.11. Finite automata ϵ-moves.

regular languages, then languages $L_1 \cup L_2$, $L_1 L_2$ and L_1^* are also regular. For regular languages are true also the following statements.

The complement of a regular language is also regular. This is easy to prove using automata. Let L be a regular language and let $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ be a DFA which accepts language L. It is easy to see that the DFA $\overline{\mathrm{A}} = (Q, \Sigma, E, \{q_0\}, Q \setminus F)$ accepts language $\overline{L}$. So, $\overline{L}$ is also regular.

The intersection of two regular languages is also regular. Since $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, the intersection is also regular.

The difference of two regular languages is also regular. Since $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$, the difference is also regular.

1.2.4. Finite automata with empty input

A finite automaton with ε-moves (FA with ε-moves) extends NFA in such way that it may have transitions on the empty input ε, i.e. it may change a state without reading any input symbol. In the case of a FA with ε-moves $\mathrm{A} = (Q, \Sigma, E, I, F)$ for the set of transitions it is true that $E \subseteq Q \times \big(\Sigma \cup \{\varepsilon\}\big) \times Q$.

The transition function of a FA with ε-moves is:

$$\delta : Q \times \big(\Sigma \cup \{\varepsilon\}\big) \to \mathcal{P}(Q), \quad \delta(p, a) = \{q \in Q \mid (p, a, q) \in E\} \ .$$

The FA with ε-moves in Fig. 1.11 accepts words of form uvw, where $u \in \{1\}^*$, $v \in \{0\}^*$ and $w \in \{1\}^*$.

Theorem 1.14 *To any FA with ε-moves one may construct an equivalent NFA (without ε-moves).*

Let $\mathrm{A} = (Q, \Sigma, E, I, F)$ be an FA with ε-moves and we construct an equivalent NFA $\overline{\mathrm{A}} = (Q, \Sigma, \overline{E}, I, \overline{F})$. The following algorithm determines sets $\overline{F}$ and $\overline{E}$.

For a state q denote by $\Lambda(q)$ the set of states (including even q) in which one may go from q using ε-moves only. This may be extended also to sets

$$\Lambda(S) = \bigcup_{q \in S} \Lambda(q), \quad \forall S \subseteq Q \ .$$

Clearly, for all $q \in Q$ and $S \subseteq Q$ both $\Lambda(q)$ and $\Lambda(S)$ may be computed. Suppose in the sequel that these are given.

The following algorithm determine the transitions using the transition function $\overline{\delta}$, which is defined in line 5.

If $|Q| = n$ and $|\Sigma| = m$,, then lines 2–6 show that the running time in worst case is $O(n^2 m)$.

ELIMINATE-EPSILON-MOVES(A)

1 $\overline{F} \leftarrow F \cup \{q \in I \mid \Lambda(q) \cap F \neq \emptyset\}$
2 **for** all $q \in Q$
3 **do for** all $a \in \Sigma$
4 **do** $\Delta \leftarrow \bigcup_{p \in \Lambda(q)} \delta(p, a)$
5 $\overline{\delta}(q, a) \leftarrow \Delta \cup \left(\bigcup_{p \in \Delta} \Lambda(p) \right)$
6 $\overline{E} \leftarrow \big\{(p, a, q), \mid p, q \in Q,\ a \in \Sigma, q \in \overline{\delta}(p, a)\big\}$
7 **return** $\overline{\mathrm{A}}$

Example 1.15 Consider the FA with ε-moves in Fig. 1.11. The corresponding transition table is:

δ	0	1	ε
q_0	$\emptyset$	$\{q_0\}$	$\{q_1\}$
q_1	$\{q_1\}$	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\{q_2\}$	$\emptyset$

Apply algorithm ELIMINATE-EPSILON-MOVES.
$\Lambda(q_0) = \{q_0, q_1, q_2\}$, $\Lambda(q_1) = \{q_1, q_2\}$, $\Lambda(q_2) = \{q_2\}$
$\Lambda(I) = \Lambda(q_0)$, and its intersection with F is not empty, thus $\overline{F} = F \cup \{q_0\} = \{q_0, q_2\}$.
$(q_0, 0)$:
$\Delta = \delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0) = \{q_1\}$, $\{q_1\} \cup \Lambda(q_1) = \{q_1, q_2\}$
$\overline{\delta}(q_0, 0) = \{q_1, q_2\}$.
$(q_0, 1)$:
$\Delta = \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1) = \{q_0, q_2\}$, $\{q_0, q_2\} \cup (\Lambda(q_0) \cup \Lambda(q_2)) = \{q_0, q_1, q_2\}$
$\overline{\delta}(q_0, 1) = \{q_0, q_1, q_2\}$
$(q_1, 0)$:
$\Delta = \delta(q_1, 0) \cup \delta(q_2, 0) = \{q_1\}$, $\{q_1\} \cup \Lambda(q_1) = \{q_1, q_2\}$

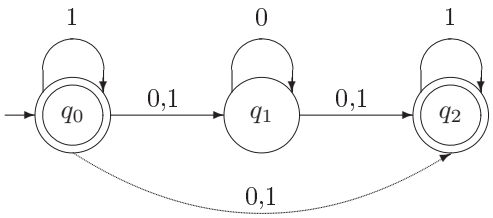


Figure 1.12. NFA equivalent to FA with ε-moves given in Fig. 1.11.

$\overline{\delta}(q_1, 0) = \{q_1, q_2\}$
$(q_1, 1)$:
$\Delta = \delta(q_1, 1) \cup \delta(q_2, 1) = \{q_2\}, \quad \{q_2\} \cup \Lambda(q_2) = \{q_2\}$
$\overline{\delta}(q_1, 1) = \{q_2\}$
$(q_1, 1) : \Delta = \delta(q_2, 0) = \emptyset$
$\overline{\delta}(q_2, 0) = \emptyset$
$(q_2, 1)$:
$\Delta = \delta(q_2, 1) = \{q_2\}, \quad \{q_2\} \cup \Lambda(q_2) = \{q_2\}$
$\overline{\delta}(q_2, 1) = \{q_2\}$.

The transition table of NFA $\overline{\mathrm{A}}$ is:

$\overline{\delta}$	0	1
q_0	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
q_1	$\{q_1, q_2\}$	$\{q_2\}$
q_2	$\emptyset$	$\{q_2\}$

and the transition graph is in Fig. 1.12.

Define regular operations on NFA: union, product and iteration. The result will be an FA with ε-moves.

Operation will be given also by diagrams. An NFA is given as in Fig. 1.13(a). Initial states are represented by a circle with an arrow, final states by a double circle.

Let $\mathrm{A}_1 = (Q_1, \Sigma_1, E_1, I_1, F_1)$ and $\mathrm{A}_2 = (Q_2, \Sigma_2, E_2, I_2, F_2)$ be NFA. The result of any operation is a FA with ε-moves $\mathrm{A} = (Q, \Sigma, E, I, F)$. Suppose that $Q_1 \cap Q_2 = \emptyset$ always. If not, we can rename the elements of any set of states.

Union. $\mathrm{A} = \mathrm{A}_1 \cup \mathrm{A}_2$, where

$$Q = Q_1 \cup Q_2 \cup \{q_0\},$$
$$\Sigma = \Sigma_1 \cup \Sigma_2,$$
$$I = \{q_0\},$$
$$F = F_1 \cup F_2,$$
$$E = E_1 \cup E_2 \cup \bigcup_{q \in I_1 \cup I_2} \{(q_0, \varepsilon, q)\}.$$

For the result of the union see Fig. 1.13(b). The result is the same if instead of a single initial state we choose as set of initial states the union $I_1 \cup I_2$. In this case the result automaton will be without ε-moves. By the definition it is easy to see that $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$.

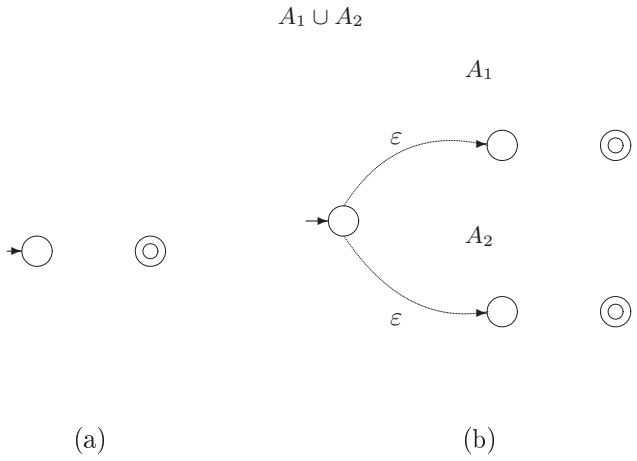


Figure 1.13. (a) Representation of an NFA. Initial states are represented by a circle with an arrow, final states by a double circle. **(b)** Union of two NFA's.

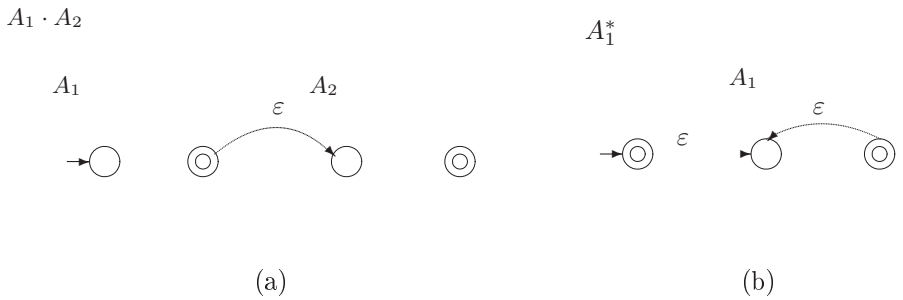


Figure 1.14. (a) Product of two FA. **(b)** Iteration of an FA.

Product. $\mathrm{A} = \mathrm{A}_1 \cdot \mathrm{A}_2$, where

$$\begin{aligned} Q &= Q_1 \cup Q_2, \\ \Sigma &= \Sigma_1 \cup \Sigma_2, \\ F &= F_2, \\ I &= I_1, \\ E &= E_1 \cup E_2 \cup \bigcup_{\substack{p \in F_1 \\ q \in I_2}} \{(p, \varepsilon, q)\} \end{aligned}$$

For the result automaton see Fig. 1.14(a). Here also $L(A_1 \cdot A_2) = L(A_1)L(A_2)$.

Iteration. $\mathrm{A} = \mathrm{A}_1{}^*$, where

$$\begin{aligned} Q &= Q_1 \cup \{q_0\}, \\ \Sigma &= \Sigma_1, \\ F &= F_1 \cup \{q_0\}, \\ I &= \{q_0\} \end{aligned}$$

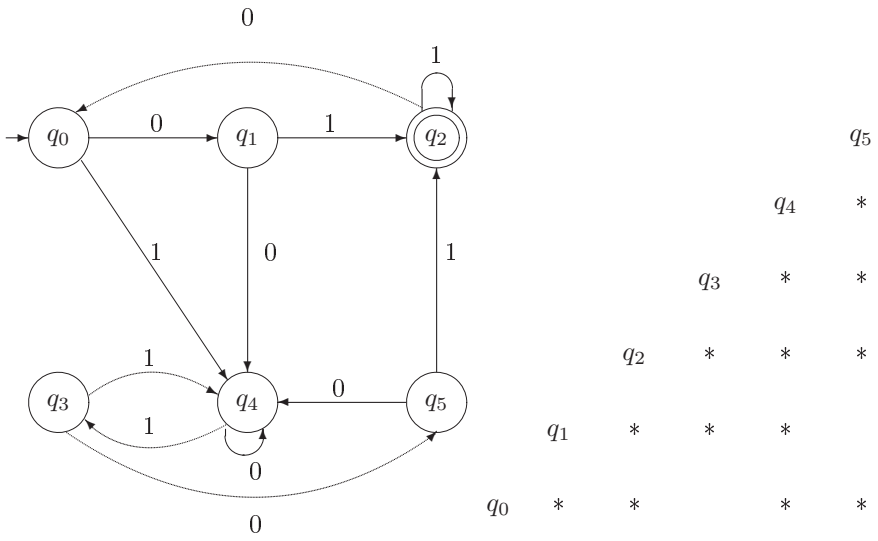


Figure 1.15. Minimization of DFA.

$$E = E_1 \cup \bigcup_{p \in I_1} \{(q_0, \varepsilon, p)\} \cup \bigcup_{\substack{q \in F_1 \\ p \in I_1}} \{(q, \varepsilon, p)\} .$$

The iteration of an FA can be seen in Fig. 1.14(b). For this operation it is also true that $L(A_1^*) = \big(L(A_1)\big)^*$.

The definition of these tree operations proves again that regular languages are closed under the regular operations.

1.2.5. Minimization of finite automata

A DFA $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ is called ***minimum state automaton*** if for any equivalent complete DFA $\mathrm{A}' = (Q', \Sigma, E', \{q_0'\}, F')$ it is true that $|Q| \leq |Q'|$. We give an algorithm which builds for any complete DFA an equivalent minimum state automaton.

States p and q of an DFA $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ are ***equivalent*** if for arbitrary word u we reach from both either final or nonfinal states, that is

$$p \equiv q \text{ if for any word } u \in \Sigma^* \begin{cases} p \xrightarrow{u} r,\ r \in F \text{ and } q \xrightarrow{u} s,\ s \in F \text{ or} \\ p \xrightarrow{u} r,\ r \notin F \text{ and } q \xrightarrow{u} s,\ s \notin F . \end{cases}$$

If two states are not equivalent, then they are distinguishable. In the following algorithm the distinguishable states will be marked by a star, and equivalent states will be merged. The algorithm will associate list of pair of states with some pair of states expecting a later marking by a star, that is if we mark a pair of states by a star, then all pairs on the associated list will be also marked by a star. The algorithm is given for DFA without inaccessible states. The used DFA is complete, so $\delta(p, a)$ contains exact one element, function *elem* defined on page 34, which gives the unique element of the set, will be also used here.

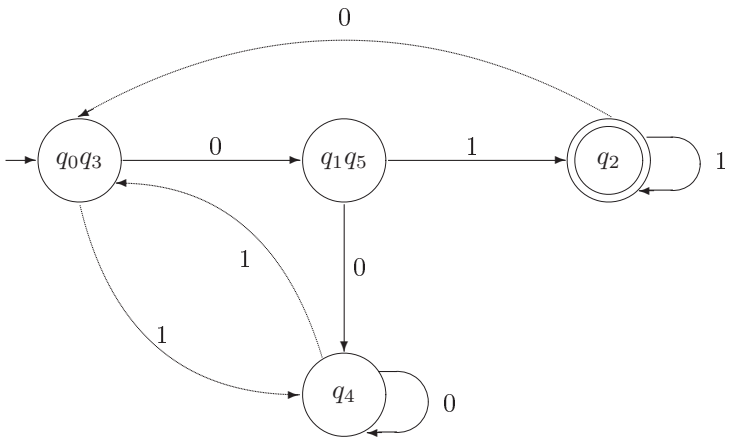


Figure 1.16. Minimum automaton equivalent with DFA in Fig. 1.15.

AUTOMATON-MINIMIZATION(A)

1 mark with a star all pairs of states $\{p, q\}$ for which
$p \in F$ and $q \notin F$ or $p \notin F$ and $q \in F$
2 associate an empty list with each unmarked pair $\{p, q\}$
3 **for** all unmarked pair of states $\{p, q\}$ and for all symbol $a \in \Sigma$
examine pairs of states $\{elem(\delta(p, a)), elem(\delta(q, a))\}$
if any of these pairs is marked,
then mark also pair $\{p, q\}$ with all the elements on the list before
associated with pair $\{p, q\}$
else if all the above pairs are unmarked
then put pair $\{p, q\}$ on each list associated with pairs
$\{elem(\delta(p, a)), elem(\delta(q, a))\}$, unless $\delta(p, a) = \delta(q, a)$
4 merge all unmarked (equivalent) pairs

After finishing the algorithm, if a cell of the table does not contain a star, then the states corresponding to its row and column index, are equivalent and may be merged. Merging states is continued until it is possible. We can say that the equivalence relation decomposes the set of states in equivalence classes, and the states in such a class may be all merged.

Remark. The above algorithm can be used also in the case of an DFA which is not complete, that is there are states for which does not exist transition. Then a pair $\{\emptyset, \{q\}\}$ may occur, and if q is a final state, consider this pair marked.

Example 1.16 Let be the DFA in Fig. 1.15. We will use a table for marking pairs with a star. Marking pair $\{p, q\}$ means putting a star in the cell corresponding to row p and column q (or row q and column p).

First we mark pairs $\{q_2, q_0\}$, $\{q_2, q_1\}$, $\{q_2, q_3\}$, $\{q_2, q_4\}$ and $\{q_2, q_5\}$ (because q_2 is the single final state). Then consider all unmarked pairs and examine them as the algorithm requires. Let us begin with pair $\{q_0, q_1\}$. Associate with it pairs $\{elem\ \delta(q_0, 0)\ , elem\ \delta(q_1, 0)\ \}, \{elem\ \delta(q_0, 1)\ , elem\ \delta(q_1, 1)\ \}$, that is $\{q_1, q_4\}$, $\{q_4, q_2\}$. Because pair $\{q_4, q_2\}$ is already marked, mark also pair $\{q_0, q_1\}$.

In the case of pair $\{q_0, q_3\}$ the new pairs are $\{q_1, q_5\}$ and $\{q_4, q_4\}$. With pair $\{q_1, q_5\}$ associate pair $\{q_0, q_3\}$ on a list, that is

$$\{q_1, q_5\} \longrightarrow \{q_0, q_3\}\ .$$

Now continuing with $\{q_1, q_5\}$ one obtain pairs $\{q_4, q_4\}$ and $\{q_2, q_2\}$, with which nothing are associated by algorithm.

Continue with pair $\{q_0, q_4\}$. The associated pairs are $\{q_1, q_4\}$ and $\{q_4, q_3\}$. None of them are marked, so associate with them on a list pair $\{q_0, q_4\}$, that is

$$\{q_1, q_4\} \longrightarrow \{q_0, q_4\}, \qquad \{q_4, q_3\} \longrightarrow \{q_0, q_4\}\ .$$

Now continuing with $\{q_1, q_4\}$ we get the pairs $\{q_4, q_4\}$ and $\{q_2, q_3\}$, and because this latter is marked we mark pair $\{q_1, q_4\}$ and also pair $\{q_0, q_4\}$ associated to it on a list. Continuing we will get the table in Fig. 1.15, that is we get that $q_0 \equiv q_3$ and $q_1 \equiv q_5$. After merging them we get an equivalent minimum state automaton (see Fig. 1.16).

1.2.6. Pumping lemma for regular languages

The following theorem, called *pumping lemma* for historical reasons, may be efficiently used to prove that a language is not regular. It is a sufficient condition for a regular language.

Theorem 1.15 (pumping lemma). *For any regular language L there exists a natural number $n \geq 1$ (depending only on L), such that any word u of L with length at least n may be written as $u = xyz$ such that*

(1) $|xy| \leq n$,

(2) $|y| \geq 1$,

(3) $xy^i z \in L$ *for all* $i = 0, 1, 2, \ldots$.

Proof If L is a regular language, then there is such an DFA which accepts L (by Theorems 1.12 and 1.10). Let $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ be this DFA, so $L = L(\mathrm{A})$. Let n be the number of its states, that is $|Q| = n$. Let $u = a_1 a_2 \ldots a_m \in L$ and $m \geq n$. Then, because the automaton accepts word u, there are states $q_0, q_1, \ldots, q_m$ and walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{m-1}} q_{m-1} \xrightarrow{a_m} q_m,\ q_m \in F.$$

Because the number of states is n and $m \geq n$, by the pigeonhole principle[3] states $q_0, q_1, \ldots, q_m$ can not all be distinct (see Fig. 1.17), there are at least two of them which are equal. Let $q_j = q_k$, where $j < k$ and k is the least such index. Then $j < k \leq n$. Decompose word u as:

$x = a_1 a_2 \ldots a_j$

[3] *Pigeonhole principle:* If we have to put more than k objects into k boxes, then at least one box will contain at least two objects.

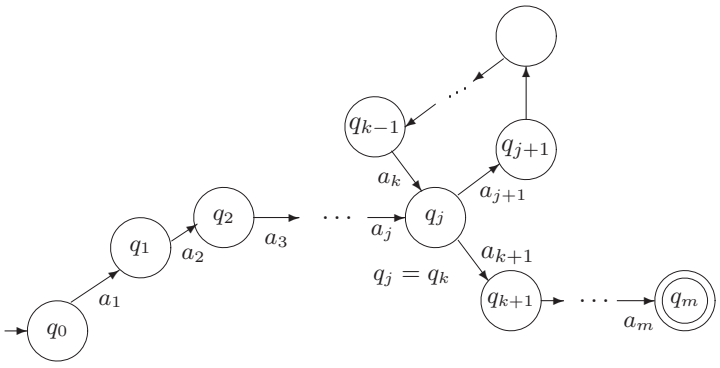


Figure 1.17. Sketch of DFA used in the proof of the pumping lemma.

$y = a_{j+1}a_{j+2}\dots a_k$
$z = a_{k+1}a_{k+2}\dots a_m$.

This decomposition immediately yields to $|xy| \leq n$ and $|y| \geq 1$. We will prove that $xy^iz \in L$ for any i.

Because $u = xyz \in L$, there exists an walk

$$q_0 \xrightarrow{x} q_j \xrightarrow{y} q_k \xrightarrow{z} q_m,\ q_m \in F,$$

and because of $q_j = q_k$, this may be written also as

$$q_0 \xrightarrow{x} q_j \xrightarrow{y} q_j \xrightarrow{z} q_m,\ q_m \in F\ .$$

From this walk $q_j \xrightarrow{y} q_j$ can be omitted or can be inserted many times. So, there are the following walks:

$$q_0 \xrightarrow{x} q_j \xrightarrow{z} q_m,\ q_m \in F\ ,$$
$$q_0 \xrightarrow{x} q_j \xrightarrow{y} q_j \xrightarrow{y} \dots \xrightarrow{y} q_j \xrightarrow{z} q_m,\ q_m \in F\ .$$

Therefore $xy^iz \in L$ for all i, and this proves the theorem. ∎

Example 1.17 We use the pumping lemma to show that $L_1 = \{a^kb^k \mid k \geq 1\}$ is not regular. Assume that L_1 is regular, and let n be the corresponding natural number given by the pumping lemma. Because the length of the word $u = a^nb^n$ is $2n$, this word can be written as in the lemma. We prove that this leads to a contradiction. Let $u = xyz$ be the decomposition as in the lemma. Then $|xy| \leq n$, so x and y can contain no other letters than a, and because we must have $|y| \geq 1$, word y contains at least one a. Then xy^iz for $i \neq 1$ will contain a different number of a's and b's, therefore $xy^iz \notin L_1$ for any $i \neq 1$. This is a contradiction with the third assertion of the lemma, this is why that assumption that L_1 is regular, is false. Therefore $L_1 \notin \mathcal{L}_3$.

Because the context-free grammar $G_1 = (\{S\}, \{a, b\}, \{S \to ab, S \to aSb\}, S)$ generates language L_1, we have $L_1 \in \mathcal{L}_2$. From these two follow that $\mathcal{L}_3 \subset \mathcal{L}_2$.

Example 1.18 We show that $L_2 = \{u \in \{0,1\}^* \mid n_0(u) = n_1(u)\}$ is not regular. ($n_0(u)$ is the number of 0's in u, while $n_1(u)$ the number of 1's).

We proceed as in the previous example using here word $u = 0^n 1^n$, where n is the natural number associated by lemma to language L_2.

Example 1.19 We prove, using the pumping lemma, that $L_3 = \{uu \mid u \in \{a,b\}^*\}$ is not a regular language. Let $w = a^n b a^n b = xyz$ be, where n here is also the natural number associated to L_3 by the pumping lemma. From $|xy| \leq n$ we have that y contains no other letters than a, but it contains at least one. By lemma we have $xz \in L_3$, that is not possible. Therefore L_3 is not regular.

Pumping lemma has several interesting consequences.

Corollary 1.16 *Regular language L is not empty if and only if there exists a word $u \in L$, $|u| < n$, where n is the natural number associated to L by the pumping lemma.*

Proof The assertion in a direction is obvious: if there exists a word shorter than n in L, then $L \neq \emptyset$. Conversely, let $L \neq \emptyset$ and let u be the shortest word in L. We show that $|u| < n$. If $|u| \geq n$, then we apply the pumping lemma, and give the decomposition $u = xyz$, $|y| > 1$ and $xz \in L$. This is a contradiction, because $|xz| < |u|$ and u is the shortest word in L. Therefore $|u| < n$. ■

Corollary 1.17 *There exists an algorithm that can decide if a regular language is or not empty.*

Proof Assume that $L = L(\mathrm{A})$, where $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ is a DFA. By consequence 1.16 and theorem 1.15 language L is not empty if and only if it contains a word shorter than n, where n is the number of states of automaton A. By this it is enough to decide that there is a word shorter than n which is accepted by automaton A. Because the number of words shorter than n is finite, the problem can be decided. ■

When we had given an algorithm for inaccessible states of a DFA, we remarked that the procedure can be used also to decide if the language accepted by that automaton is or not empty. Because finite automata accept regular languages, we can consider to have already two procedures to decide if a regular languages is or not empty. Moreover, we have a third procedure, if we take into account that the algorithm for finding productive states also can be used to decide on a regular language when it is empty.

Corollary 1.18 *A regular language L is infinite if and only if there exists a word $u \in L$ such that $n \leq |u| < 2n$, where n is the natural number associated to language L, given by the pumping lemma.*

Proof If L is infinite, then it contains words longer than $2n$, and let u be the shortest word longer than $2n$ in L. Because L is regular we can use the pumping lemma, so $u = xyz$, where $|xy| \leq n$, thus $|y| \leq n$ is also true. By the lemma $u' = xz \in L$. But because $|u'| < |u|$ and the shortest word in L longer than $2n$ is u, we get $|u'| < 2n$. From $|y| \leq n$ we get also $|u'| \geq n$.

Conversely, if there exists a word $u \in L$ such that $n \leq |u| < 2n$, then using the pumping lemma, we obtain that $u = xyz$, $|y| \geq 1$ and $xy^i z \in L$ for any i, therefore L is infinite. ■

Now, the question is: how can we apply the pumping lemma for a finite regular language, since by pumping words we get an infinite number of words? The number of states of a DFA accepting language L is greater than the length of the longest word in L. So, in L there is no word with length at least n, when n is the natural number associated to L by the pumping lemma. Therefore, no word in L can be decomposed in the form xyz, where $|xyz| \geq n$, $|xy| \leq n$, $|y| \geq 1$, and this is why we can not obtain an infinite number of words in L.

1.2.7. Regular expressions

In this subsection we introduce for any alphabet Σ the notion of regular expressions over Σ and the corresponding representing languages. A regular expression is a formula, and the corresponding language is a language over Σ. For example, if $\Sigma = \{a, b\}$, then a^*, b^*, $a^* + b^*$ are regular expressions over Σ which represent respectively languages $\{a\}^*$, $\{b\}^*$, $\{a\}^* \cup \{b\}^*$. The exact definition is the following.

Definition 1.19 *Define recursively a regular expression over Σ and the language it represent.*

- *$\emptyset$ is a regular expression representing the empty language.*
- *ε is a regular expression representing language $\{\varepsilon\}$.*
- *If $a \in \Sigma$, then a is a regular expression representing language $\{a\}$.*
- *If x, y are regular expressions representing languages X and Y respectively, then $(x + y)$, (xy), (x^*) are regular expressions representing languages $X \cup Y$, XY and X^* respectively.*

Regular expression over Σ can be obtained only by using the above rules a finite number of times.

Some brackets can be omitted in the regular expressions if taking into account the priority of operations (iteration, product, union) the corresponding languages are not affected. For example instead of $((x^*)(x + y))$ we can consider $x^*(x + y)$.

Two regular expressions are ***equivalent*** if they represent the same language, that is $x \equiv y$ if $X = Y$, where X and Y are the languages represented by regular expressions x and y respectively. Figure 1.18 shows some equivalent expressions.

We show that to any finite language L can be associated a regular expression x which represent language L. If $L = \emptyset$, then $x = \emptyset$. If $L = \{w_1, w_2, \ldots, w_n\}$, then $x = x_1 + x_2 + \ldots + x_n$, where for any $i = 1, 2, \ldots, n$ expression x_i is a regular expression representing language $\{w_i\}$. This latter can be done by the following rule. If $w_i = \varepsilon$, then $x_i = \varepsilon$, else if $w_i = a_1 a_2 \ldots a_m$, where $m \geq 1$ depends on i, then $x_i = a_1 a_2 \ldots a_m$, where the brackets are omitted.

We prove the theorem of Kleene which refers to the relationship between regular languages and regular expression.

Theorem 1.20 (Kleene's theorem). *Language $L \subseteq \Sigma^*$ is regular if and only if there exists a regular expression over Σ representing language L.*

Proof First we prove that if x is a regular expression, then language L which represents x is also regular. The proof will be done by induction on the construction

$$
\begin{aligned}
x+y &\equiv y+x \\
(x+y)+z &\equiv x+(y+z) \\
(xy)z &\equiv x(yz) \\
(x+y)z &\equiv xz+yz \\
x(y+z) &\equiv xy+xz \\
(x+y)^* \equiv (x^*+y)^* &\equiv (x+y^*)^* \equiv (x^*+y^*)^* \\
(x+y)^* &\equiv (x^*y^*)^* \\
(x^*)^* &\equiv x^* \\
x^*x &\equiv xx^* \\
xx^*+\varepsilon &\equiv x^*
\end{aligned}
$$

Figure 1.18. Properties of regular expressions.

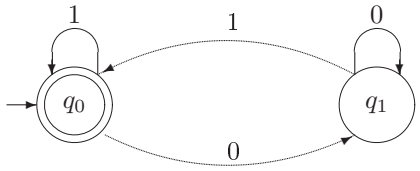


Figure 1.19. DFA from Example 1.20, to which regular expression is associated by Method 1.

of expression.

If $x = \emptyset$, $x = \varepsilon$, $x = a, \forall a \in \Sigma$, then $L = \emptyset$, $L = \{\varepsilon\}$, $L = \{a\}$ respectively. Since L is finite in all three cases, it is also regular.

If $x = (x_1 + x_2)$, then $L = L_1 \cup L_2$, where L_1 and L_2 are the languages which represent the regular expressions x_1 and x_2 respectively. By the induction hypothesis languages L_1 and L_2 are regular, so L is also regular because regular languages are closed on union. Cases $x = (x_1x_2)$ and $x = (x_1^*)$ can be proved by similar way.

Conversely, we prove that if L is a regular language, then a regular expression x can be associated to it, which represent exactly the language L. If L is regular, then there exists a DFA $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ for which $L = L(\mathrm{A})$. Let $q_0, q_1, \ldots, q_n$ the states of the automaton A. Define languages R_{ij}^k for all $-1 \leq k \leq n$ and $0 \leq i, j \leq n$. R_{ij}^k is the set of words, for which automaton A goes from state q_i to state q_j without using any state with index greater than k. Using transition graph we can say: a word is in R_{ij}^k, if from state q_i we arrive to state q_j following the edges of the graph, and concatenating the corresponding labels on edges we get exactly that word, not using any state $q_{k+1}, \ldots q_n$. Sets R_{ij}^k can be done also formally:

$R_{ij}^{-1} = \{a \in \Sigma \mid (q_i, a, q_j) \in E\}$, if $i \neq j$,

$R_{ii}^{-1} = \{a \in \Sigma \mid (q_i, a, q_i) \in E\} \cup \{\varepsilon\}$,

$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} \left(R_{kk}^{k-1}\right)^* R_{kj}^{k-1}$ for all $i, j, k \in \{0, 1, \ldots, n\}$.

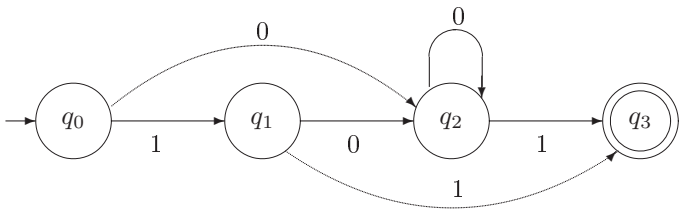


Figure 1.20. DFA in Example 1.21 to which a regular expression is associated by Method 1. The computation are in Figure 1.21.

We can prove by induction that sets R_{ij}^k can be described by regular expressions. Indeed, if $k = -1$, then for all i and j languages R_{ij}^k are finite, so they can be expressed by regular expressions representing exactly these languages. Moreover, if for all i and j language R_{ij}^{k-1} can be expressed by regular expression, then language R_{ij}^k can be expressed also by regular expression, which can be corresponding constructed from regular expressions representing languages R_{ij}^{k-1}, R_{ik}^{k-1}, R_{kk}^{k-1} and R_{kj}^{k-1} respectively, using the above formula for R_{ij}^k.

Finally, if $F = \{q_{i_1}, q_{i_2}, \ldots, q_{i_p}\}$ is the set of final states of the DFA A, then $L = L(\mathrm{A}) = R_{0i_1}^n \cup R_{0i_2}^n \cup \ldots \cup R_{0i_p}^n$ can be expressed by a regular expression obtained from expressions representing languages $R_{0i_1}^n, R_{0i_2}^n, \ldots, R_{0i_p}^n$ using operation $+$. ■

Further on we give some procedures which associate DFA to regular expressions and conversely regular expression to DFA.

Associating regular expressions to finite automata We present here three methods, each of which associate to a DFA the corresponding regular expression.

Method 1. Using the result of the theorem of Kleene, we will construct the sets R_{ij}^k, and write a regular expression which represent the language $L = R_{0i_1}^n \cup R_{0i_2}^n \cup \ldots \cup R_{0i_p}^n$, where $F = \{q_{i_1}, q_{i_2}, \ldots, q_{i_p}\}$ is the set of final states of the automaton.

Example 1.20 Consider the DFA in Fig. 1.19.

$L(\mathrm{A}) = R_{00}^1 = R_{00}^0 \cup R_{01}^0 \ R_{11}^0 \ ^* \ R_{10}^0$

R_{00}^0 : $1^* + \varepsilon \equiv 1^*$

R_{01}^0 : 1^*0

R_{11}^0 : $11^*0 + \varepsilon + 0 \equiv (11^* + \varepsilon)0 + \varepsilon \equiv 1^*0 + \varepsilon$

R_{10}^0 : 11^*

Then the regular expression corresponding to $L(\mathrm{A})$ is $1^* + 1^*0(1^*0 + \varepsilon)^*11^* \equiv 1^* + 1^*0(1^*0)^*11^*$.

Example 1.21 Find a regular expression associated to DFA in Fig. 1.20. The computations are in Figure 1.21. The regular expression corresponding to R_{03}^3 is $11 + (0 + 10)0^*1$.

	$k=-1$	$k=0$	$k=1$	$k=2$	$k=3$
R_{00}^k	ε	ε	ε	ε	
R_{01}^k	1	1	1	1	
R_{02}^k	0	0	$0+10$	$(0+10)0^*$	
R_{03}^k	$\emptyset$	$\emptyset$	11	$11+(0+10)0^*1$	$11+(0+10)0^*1$
R_{11}^k	ε	ε	ε	ε	
R_{12}^k	0	0	0	00^*	
R_{13}^k	1	1	1	$1+00^*1$	
R_{22}^k	$0+\varepsilon$	$0+\varepsilon$	$0+\varepsilon$	0^*	
R_{23}^k	1	1	1	0^*1	
R_{33}^k	ε	ε	ε	ε	

Figure 1.21. Determining a regular expression associated to DFA in Figure 1.20 using sets R_{ij}^k.

Method 2. Now we generalize the notion of finite automaton, considering words instead of letters as labels of edges. In such an automaton each walk determine a regular expression, which determine a regular language. The regular language accepted by a generalized finite automaton is the union of regular languages determined by the productive walks. It is easy to see that the generalized finite automata accept regular languages.

The advantage of generalized finite automata is that the number of its edges can be diminuted by equivalent transformations, which do not change the accepted language, and leads to a graph with only one edge which label is exactly the accepted language.

The possible equivalent transformations can be seen in Fig. 1.22. If some of the

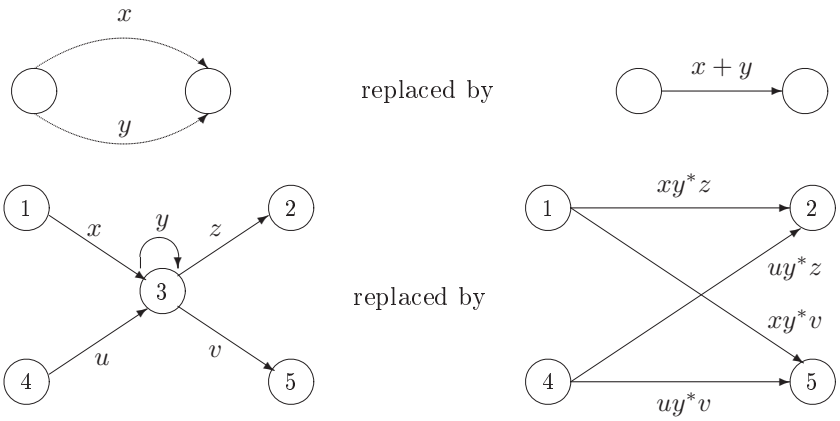


Figure 1.22. Possible equivalent transformations for finding regular expression associated to an automaton.

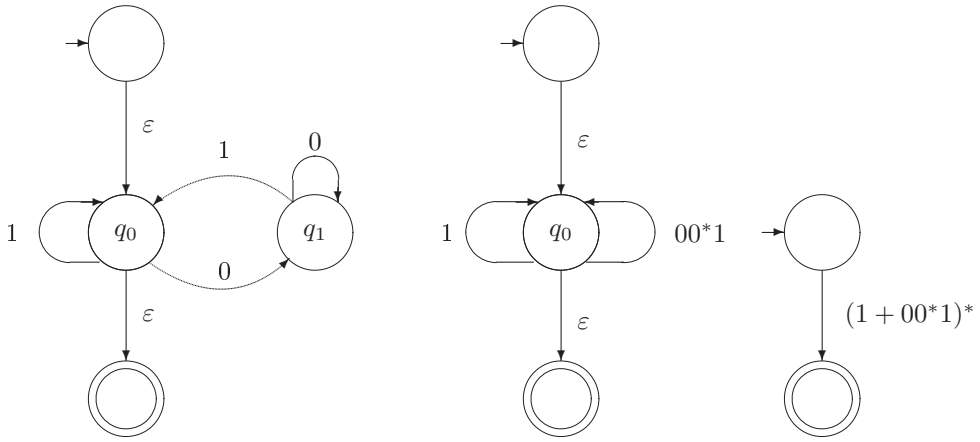


Figure 1.23. Transformation of the finite automaton in Fig. 1.19.

vertices 1, 2, 4, 5 on the figure coincide, in the result they are merged, and a loop will arrive.

First, the automaton is transformed by corresponding ε-moves to have only one initial and one final state. Then, applying the equivalent transformations until the graph will have only one edge, we will obtain as the label of this edge the regular expression associated to the automaton.

Example 1.22 In the case of Fig. 1.19 the result is obtained by steps illustrated in Fig. 1.23. This result is $(1+00^*1)^*$, which represents the same language as obtained by Method 1 (See example 1.20).

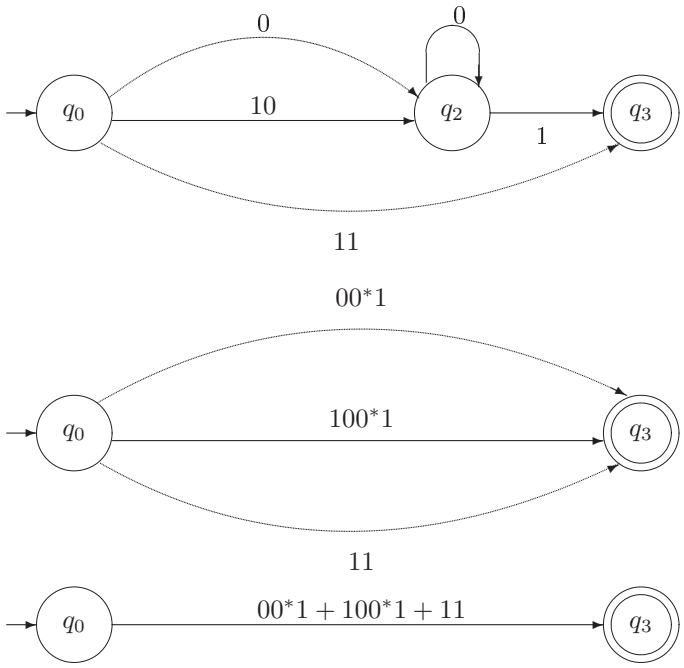


Figure 1.24. Steps of Example 1.23.

Example 1.23 In the case of Fig. 1.20 is not necessary to introduce new initial and final state. The steps of transformations can be seen in Fig. 1.24. The resulted regular expression can be written also as $(0 + 10)0^*1 + 11$, which is the same as obtained by the previous method.

Method 3. The third method for writing regular expressions associated to finite automata uses formal equations. A variable X is associated to each state of the automaton (to different states different variables). Associate to each state an equation which left side contains X, its right side contains sum of terms of form Ya or ε, where Y is a variable associated to a state, and a is its corresponding input symbol. If there is no incoming edge in the state corresponding to X then the right side of the equation with left side X contains ε, otherwise is the sum of all terms of the form Ya for which there is a transition labelled with letter a from state corresponding to Y to the state corresponding to X. If the state corresponding to X is also an initial and a final state, then on right side of the equation with the left side X will be also a term equal to ε. For example in the case of Fig. 1.20 let these variable X, Y, Z, U corresponding to the states q_0, q_1, q_2, q_3. The corresponding equation are

$X = \varepsilon$
$Y = X1$
$Z = X0 + Y0 + Z0$
$U = Y1 + Z1.$

If an equation is of the form $X = X\alpha + \beta$, where α, β are arbitrary words not containing X, then it is easy to see by a simple substitution that $X = \beta\alpha^*$ is a solution of the equation.

Because these equations are linear, all of them can be written in the form $X = X\alpha + \beta$ or $X = X\alpha$, where α do not contain any variable. Substituting this in the other equations the number of remaining equations will be diminuted by one. In such a way the system of equation can be solved for each variable.

The solution will be given by variables corresponding to final states summing the corresponding regular expressions.

In our example from the first equation we get $Y = 1$. From here $Z = 0 + 10 + Z0$, or $Z = Z0 + (0 + 10)$, and solving this we get $Z = (0 + 10)0^*$. Variable U can be obtained immediately and we obtain $U = 11 + (0 + 10)0^*1$.

Using this method in the case of Fig. 1.19, the following equations will be obtained

$X = \varepsilon + X1 + Y1$

$Y = X0 + Y0$

Therefore

$X = \varepsilon + (X + Y)1$

$Y = (X + Y)0$.

Adding the two equations we will obtain

$X + Y = \varepsilon + (X + Y)(0 + 1)$, from where (considering ε as β and $(0 + 1)$ as α) we get the result

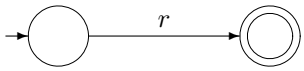
$X + Y = (0 + 1)^*$.

From here the value of X after the substitution is

$X = \varepsilon + (0 + 1)^*1$,

which is equivalent to the expression obtained using the other methods.

Associating finite automata to regular expressions Associate to the regular expression r a generalized finite automaton:



After this, use the transformations in Fig. 1.25 step by step, until an automaton with labels equal to letters from Σ or ε will be obtained.

Example 1.24 Get started from regular expression $\varepsilon + (0 + 1)^*1$. The steps of transformations are in Fig. 1.26(a)-(e). The last finite automaton (see Fig. 1.26(e)) can be done in a simpler form as can be seen in Fig. 1.26(f). After eliminating the ε-moves and transforming in a deterministic finite automaton the DFA in Fig. 1.27 will be obtained, which is equivalent to DFA in Fig. 1.19.

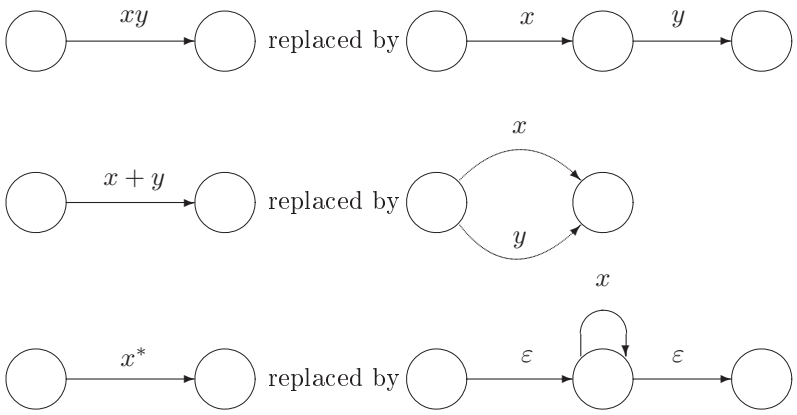


Figure 1.25. Possible transformations to obtain finite automaton associated to a regular expression.

Exercises

1.2-1 Give a DFA which accepts natural numbers divisible by 9.

1.2-2 Give a DFA which accepts the language containing all words formed by

a. an even number of 0's and an even number of 1's,

b. an even number of 0's and an odd number of 1's,

c. an odd number of 0's and an even number of 1's,

d. an odd number of 0's and an odd number of 1's.

1.2-3 Give a DFA to accept respectively the following languages:

$L_1 = \{a^n b^m \mid n \geq 1, m \geq 0\}$, $L_2 = \{a^n b^m \mid n \geq 1, m \geq 1\}$,
$L_3 = \{a^n b^m \mid n \geq 0, m \geq 0\}$, $L_4 = \{a^n b^m \mid n \geq 0, m \geq 1\}$.

1.2-4 Give an NFA which accepts words containing at least two 0's and any number of 1's. Give an equivalent DFA.

1.2-5 Minimize the DFA's in Fig. 1.28.

1.2-6 Show that the DFA in 1.29.(a) is a minimum state automaton.

1.2-7 Transform NFA in Fig. 1.29.(b) in a DFA, and after this minimize it.

1.2-8 Define finite automaton A_1 which accepts all words of the form $0(10)^n$ $(n \geq 0)$, and finite automaton A_2 which accepts all words of the form $1(01)^n$ $(n \geq 0)$. Define the union automaton $\mathrm{A}_1 \cup \mathrm{A}_2$, and then eliminate the ε-moves.

1.2-9 Associate to DFA in Fig. 1.30 a regular expression.

1.2-10 Associate to regular expression $ab^*ba^* + b + ba^*a$ a DFA.

1.2-11 Prove, using the pumping lemma, that none of the following languages are regular:

$L_1 = \{a^n c b^n \mid n \geq 0\}$, $L_2 = \{a^n b^n a^n \mid n \geq 0\}$, $L_3 = \{a^p \mid p \text{ prím}\}$.

1.2-12 Prove that if L is a regular language, then $\{u^{-1} \mid u \in L\}$ is also regular.

1.2-13 Prove that if $L \subseteq \Sigma^*$ is a regular language, then the following languages are also regular.

$\mathrm{pre}(L) = \{w \in \Sigma^* \mid \exists u \in \Sigma^*, wu \in L\}$, $\mathrm{suf}(L) = \{w \in \Sigma^* \mid \exists u \in \Sigma^*, uw \in L\}$.

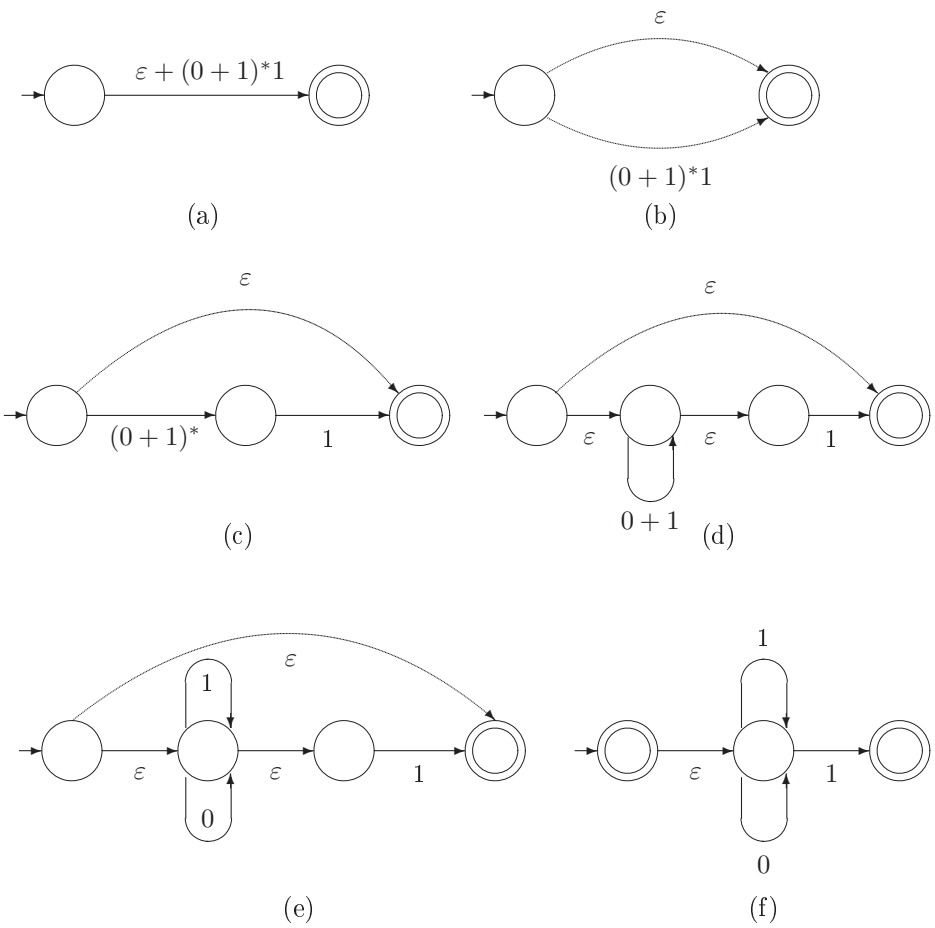


Figure 1.26. Associating finite automaton to regular expression $\varepsilon + (0+1)^*1$.

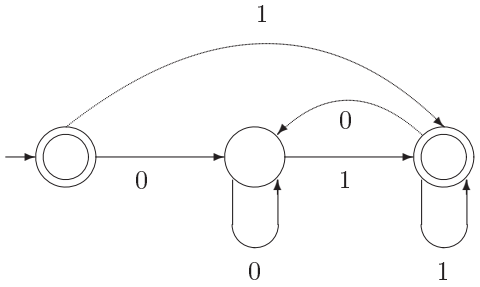


Figure 1.27. Finite automaton associated to regular expression $\varepsilon + (0+1)^*1$.

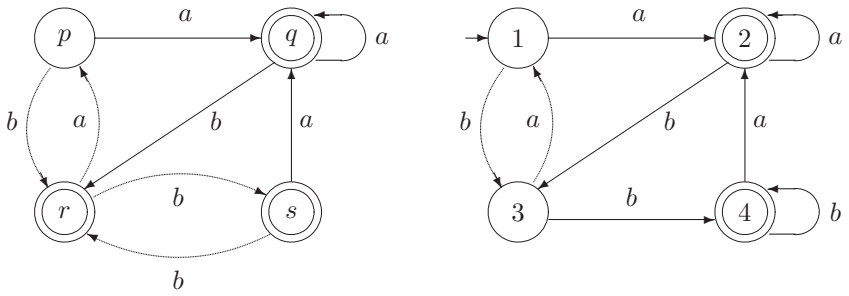


Figure 1.28. DFA's to minimize for Exercise 1.2-5

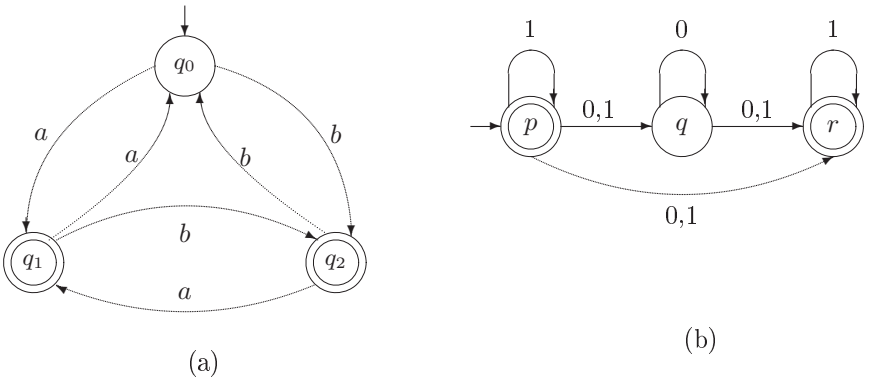


Figure 1.29. Finite automata for Exercises 1.2-6 and 1.2-7

1.2-14 Show that the following languages are all regular.

$L_1 = \{ab^n cd^m \mid n > 0, m > 0\}$,
$L_2 = \{(ab)^n \mid n \geq 0\}$,
$L_3 = \{a^{kn} \mid n \geq 0, \ \ k \text{ constant}\}$.

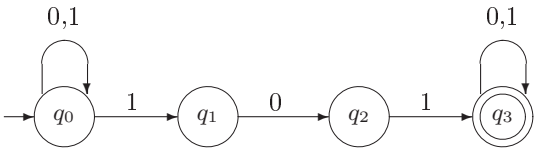


Figure 1.30. DFA for Exercise 1.2-9

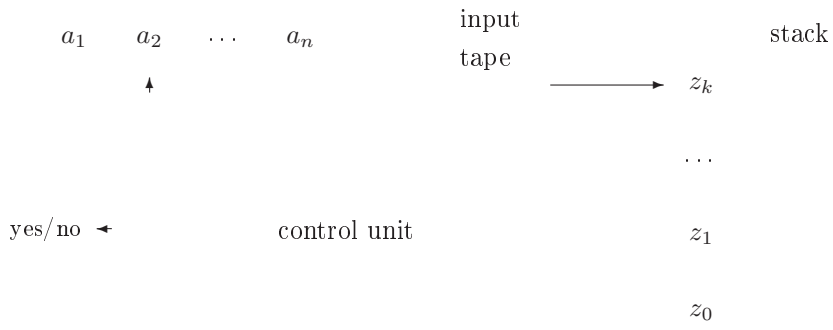


Figure 1.31. Pushdown automaton.

1.3. Pushdown automata and context-free languages

In this section we deal with the pushdown automata and the class of languages — the context-free languages — accepted by them.

As we have been seen in Section 1.1, a context-free grammar $G = (N, T, P, S)$ is one with the productions of the form $A \to \beta$, $A \in N$, $\beta \in (N \cup T)^+$. The production $S \to \varepsilon$ is also permitted if S does not appear in right hand side of any productions. Language $L(G) = \{u \in T \mid S \overset{*}{\underset{G}{\Longrightarrow}} u\}$ is the context-free language generated by grammar G.

1.3.1. Pushdown automata

We have been seen that finite automata accept the class of regular languages. Now we get to know a new kind of automata, the so-called ***pushdown automata***, which accept context-free languages. The pushdown automata differ from finite automata mainly in that to have the possibility to change states without reading any input symbol (i.e. to read the empty symbol) and possess a stack memory, which uses the so-called stack symbols (See Fig. 1.31).

The pushdown automaton get a word as input, start to function from an initial state having in the stack a special symbol, the initial stack symbol. While working, the pushdown automaton change its state based on current state, next input symbol (or empty word) and stack top symbol and replace the top symbol in the stack with a (possibly empty) word.

There are two type of acceptances. The pushdown automaton accepts a word by final state when after reading it the automaton enter a final state. The pushdown automaton accepts a word by empty stack when after reading it the automaton empties its stack. We show that these two acceptances are equivalent.

Definition 1.21 *A **nondeterministic pushdown automaton** is a system*

$$\mathrm{V} = (Q, \Sigma, W, E, q_0, z_0, F),$$

where

- Q *is the finite, non-empty set of* ***states***
- Σ *is the* ***input alphabet,***
- W *is the* ***stack alphabet,***
- $E \subseteq Q \times \big(\Sigma \cup \{\varepsilon\}\big) \times W \times W^* \times Q$ *is the set of* ***transitions*** *or* ***edges,***
- $q_0 \in Q$ *is the* ***initial state,***
- $z_0 \in W$ *is the* ***start symbol of stack,***
- $F \subseteq Q$ *is the set of* ***final states****.*

A transition (p, a, z, w, q) means that if pushdown automaton V is in state p, reads from the input tape letter a (instead of input letter we can also consider the empty word ε), and the top symbol in the stack is z, then the pushdown automaton enters state q and replaces in the stack z by word w. Writing word w in the stack is made by natural order (letters of word w will be put in the stack letter by letter from left to right). Instead of writing transition (p, a, z, w, q) we will use a more suggestive notation $\big(p, (a, z/w), q\big)$.

Here, as in the case of finite automata, we can define a transition function

$$\delta \ : \ Q \times (\Sigma \cup \{\varepsilon\}) \times W \to \mathcal{P}(W^* \times Q) \ ,$$

which associate to current state, input letter and top letter in stack pairs of the form (w, q), where $w \in W^*$ is the word written in stack and $q \in Q$ the new state.

Because the pushdown automaton is nondeterministic, we will have for the transition function

$\delta(q, a, z) = \{(w_1, p_1), \ldots, (w_k, p_k)\}$ (if the pushdown automaton reads an input letter and moves to right), or

$\delta(q, \varepsilon, z) = \{(w_1, p_1), \ldots, (w_k, p_k)\}$ (without move on the input tape).

A pushdown automaton is ***deterministic,*** if for any $q \in Q$ and $z \in W$ we have

- $|\delta(q, a, z)| \leq 1$, $\forall a \in \Sigma \cup \{\varepsilon\}$ and
- if $\delta(q, \varepsilon, z) \neq \emptyset$, then $\delta(q, a, z) = \emptyset$, $\forall a \in \Sigma$.

We can associate to any pushdown automaton a transition table, exactly as in the case of finite automata. The rows of this table are indexed by elements of Q, the columns by elements from $\Sigma \cup \{\varepsilon\}$ and W (to each $a \in \Sigma \cup \{\varepsilon\}$ and $z \in W$ will correspond a column). At intersection of row corresponding to state $q \in Q$ and column corresponding to $a \in \Sigma \cup \{\varepsilon\}$ and $z \in W$ we will have pairs $(w_1, p_1), \ldots, (w_k, p_k)$ if $\delta(q, a, z) = \{(w_1, p_1), \ldots, (w_k, p_k)\}$.

The transition graph, in which the label of edge (p, q) will be $(a, z/w)$ corresponding to transition $\big(p, (a, z/w), q\big)$, can be also defined.

Example 1.25 $\mathrm{V}_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \{z_0, z_1\}, E, q_0, z_0, \{q_0\})$. Elements of E are:

$q_0, (a, z_0/z_0z_1), q_1$
$q_1, (a, z_1/z_1z_1), q_1$ $\qquad$ $q_1, (b, z_1/\varepsilon), q_2$
$q_2, (b, z_1/\varepsilon), q_2$ $\qquad$ $q_2, (\varepsilon, z_0/\varepsilon), q_0$.

The transition function:

$\delta(q_0, a, z_0) = \{(z_0z_1, q_1)\}$

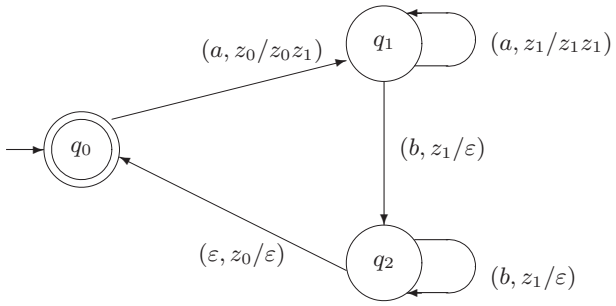


Figure 1.32. Example of pushdown automaton.

$$\delta(q_1, a, z_1) = \{(z_1z_1, q_1)\} \qquad \delta(q_1, b, z_1) = \{(\varepsilon, q_2)\}$$
$$\delta(q_2, b, z_1) = \{(\varepsilon, q_2)\} \qquad \delta(q_2, \varepsilon, z_0) = \{(\varepsilon, q_0)\}\ .$$

The transition table:

$\Sigma \cup \{\varepsilon\}$	a		b	ε
W	z_0	z_1	z_1	z_0
q_0	(z_0z_1, q_1)			
q_1		(z_1z_1, q_1)	(ε, q_2)	
q_2			(ε, q_2)	(ε, q_0)

Because for the transition function every set which is not empty contains only one element (e.g. $\delta(q_0, a, z_0) = \{(z_0z_1, q_1)\}$), in the above table each cell contains only one element, And the set notation is not used. Generally, if a set has more than one element, then its elements are written one under other. The transition graph of this pushdown automaton is in Fig. 1.32.

The current state, the unread part of the input word and the content of stack constitutes a ***configuration*** of the pushdown automaton, i.e. for each $q \in Q$, $u \in \Sigma^*$ and $v \in W^*$ the triplet (q, u, v) can be a configuration.

If $u = a_1a_2 \dots a_k$ and $v = x_1x_2 \dots x_m$, then the pushdown automaton can change its configuration in two ways:

- $(q,\ a_1a_2 \dots a_k,\ x_1x_2 \dots x_{m-1}x_m) \Longrightarrow (p,\ a_2a_3 \dots a_k,\ x_1, x_2 \dots x_{m-1}w)$,
  if $\big(q, (a_1, x_m/w), p\big) \in E$
- $(q,\ a_1a_2 \dots a_k,\ x_1x_2 \dots x_m) \Longrightarrow (p,\ a_1a_2 \dots a_k,\ x_1, x_2 \dots x_{m-1}w)$,
  if $\big(q, (\varepsilon, x_m/w), p\big) \in E$.

The reflexive and transitive closure of the relation $\Longrightarrow$ will be denoted by $\overset{*}{\Longrightarrow}$.

Instead of using $\Longrightarrow$, sometimes $\vdash$ is considered.

How does work such a pushdown automaton? Getting started with the initial configuration $(q_0, a_1a_2 \dots a_n, z_0)$ we will consider all possible next configurations, and after this the next configurations to these next configurations, and so on, until it is possible.

Definition 1.22 *Pushdown automaton* V *accepts (recognizes) word* u ***by final state*** *if there exist a sequence of configurations of* V *for which the following are true:*

- *the first element of the sequence is* (q_0, u, z_0),
- *there is a going from each element of the sequence to the next element, excepting the case when the sequence has only one element,*
- *the last element of the sequence is* (p, ε, w), *where* $p \in F$ *and* $w \in W^*$.

Therefore pushdown automaton V accepts word u by final state, if and only if $(q_0, u, z_0) \overset{*}{\Longrightarrow} (p, \varepsilon, w)$ for some $w \in W^*$ and $p \in F$. The set of words accepted by final state by pushdown automaton V will be called the language accepted by V by final state and will be denoted by $L(\mathrm{V})$.

Definition 1.23 *Pushdown automaton* V *accepts (recognizes) word* u ***by empty stack*** *if there exist a sequence of configurations of* V *for which the following are true:*

- *the first element of the sequence is* (q_0, u, z_0),
- *there is a going from each element of the sequence to the next element,*
- *the last element of the sequence is* $(p, \varepsilon, \varepsilon)$ *and* p *is an arbitrary state.*

Therefore pushdown automaton V accepts a word u by empty stack if $(q_0, u, z_0) \overset{*}{\Longrightarrow} (p, \varepsilon, \varepsilon)$ for some $p \in Q$. The set of words accepted by empty stack by pushdown automaton V will be called the language accepted by empty stack by V and will be denoted by $L_\varepsilon(\mathrm{V})$.

Example 1.26 Pushdown automaton V_1 of Example 1.25 accepts the language $\{a^n b^n \mid n \geq 0\}$ by final state. Consider the derivation for words *aaabbb* and *abab*.

Word a^3b^3 is accepted by the considered pushdown automaton because
$(q_0, aaabbb, z_0) \Longrightarrow (q_1, aabbb, z_0z_1) \Longrightarrow (q_1, abbb, z_0z_1z_1) \Longrightarrow (q_1, bbb, z_0z_1z_1z_1)$
$\Longrightarrow (q_2, bb, z_0z_1z_1) \Longrightarrow (q_2, b, z_0z_1) \Longrightarrow (q_2, \varepsilon, z_0) \Longrightarrow (q_0, \varepsilon, \varepsilon)$ and because q_0 is a final state the pushdown automaton accepts this word. But the stack being empty, it accepts this word also by empty stack.

Because the initial state is also a final state, the empty word is accepted by final state, but not by empty stack.

To show that word *abab* is not accepted, we need to study all possibilities. It is easy to see that in our case there is only a single possibility:

$(q_0, abab, z_0) \Longrightarrow (q_1, bab, z_0z_1) \Longrightarrow (q_2, ab, z_0) \Longrightarrow (q_0, ab, \varepsilon)$, but there is no further going, so word *abab* is not accepted.

Example 1.27 The transition table of the pushdown automaton $\mathrm{V}_2 = (\{q_0, q_1\}, \{0, 1\}, \{z_0, z_1, z_2\}, E, q_0, z_0, \emptyset)$ is:

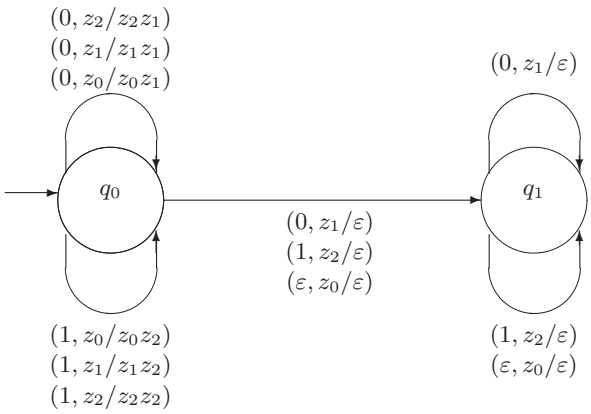


Figure 1.33. Transition graph of the Example 1.27

$\Sigma \cup \{\varepsilon\}$	0			1			ε
W	z_0	z_1	z_2	z_0	z_1	z_2	z_0
q_0	(z_0z_1, q_0)	(z_1z_1, q_0) (ε, q_1)	(z_2z_1, q_0)	(z_0z_2, q_0)	(z_1z_2, q_0)	(z_2z_2, q_0) (ε, q_1)	(ε, q_1)
q_1		(ε, q_1)				(ε, q_1)	(ε, q_1)

The corresponding transition graph can be seen in Fig. 1.33. Pushdown automaton V_2 accepts the language $uu^{-1} \mid u \in \{0,1\}^*$. Because V_2 is nemdeterministic, all the configurations obtained from the initial configuration (q_0, u, z_0) can be illustrated by a ***computation tree***. For example the computation tree associated to the initial configuration $(q_0, 1001, z_0)$ can be seen in Fig. 1.34. From this computation tree we can observe that, because $(q_1, \varepsilon, \varepsilon)$ is a leaf of the tree, pushdown automaton V_2 accepts word 1001 by empty stack. The computation tree in Fig. 1.35 shows that pushdown automaton V_2 does not accept word 101, because the configurations in leaves can not be continued and none of them has the form $(q, \varepsilon, \varepsilon)$.

Theorem 1.24 *A language L is accepted by a nondeterministic pushdown automaton V_1 by empty stack if and only if it can be accepted by a nondeterministic pushdown automaton V_2 by final state.*

Proof *a)* Let $V_1 = (Q, \Sigma, W, E, q_0, z_0, \emptyset)$ be the pushdown automaton which accepts by empty stack language L. Define pushdown automaton $V_2 = (Q \cup \{p_0, p\}, \Sigma, W \cup \{x\}, E', p_0, x, \{p\})$, where $p, p_0 \notin Q$, , $x \notin W$ and

$$E' = E \cup \Big\{\big(p_0, (\varepsilon, x/xz_0), q_0\big)\Big\} \cup \Big\{\big(q, (\varepsilon, x/\varepsilon), p\big) \mid q \in Q\Big\}.$$

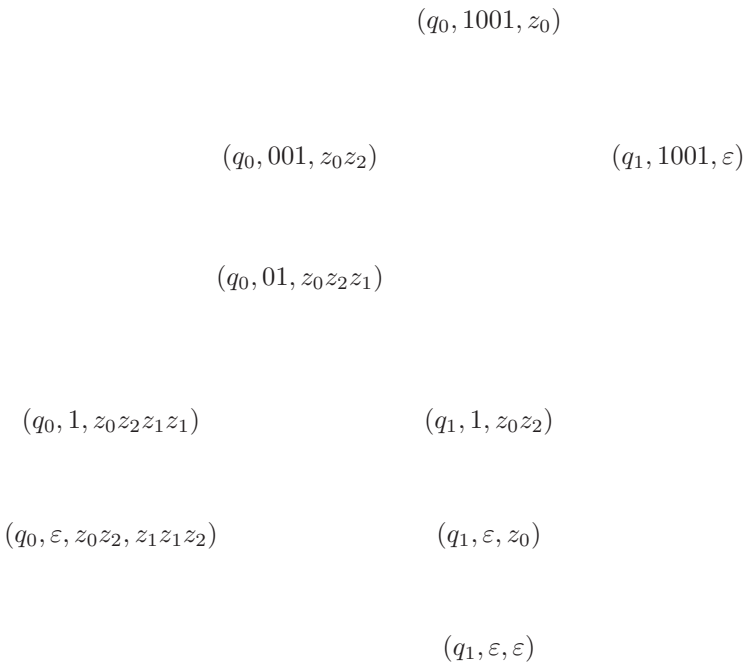


Figure 1.34. Computation tree to show acceptance of the word 1001 (see Example 1.27).

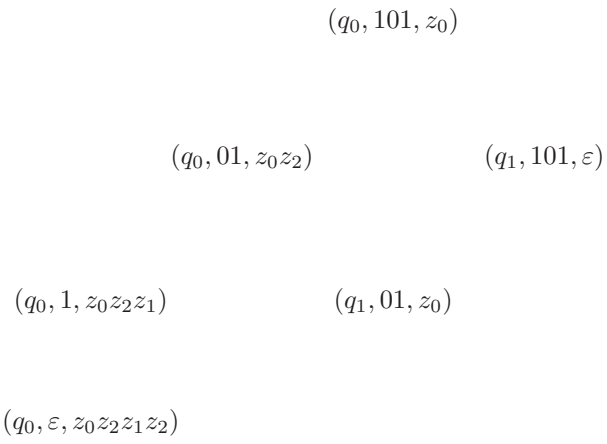


Figure 1.35. Computation tree to show that the pushdown automaton in Example 1.27 does not accept word 101.

Working of V_2: Pushdown automaton V_2 with an ε-move first goes in the initial state of V_1, writing z_0 (the initial stack symbol of V_1) in the stack (beside x). After this it is working as V_1. If V_1 for a given word empties its stack, then V_2 still has x in the stack, which can be deleted by V_2 using an ε-move, while a final state will be

reached. V_2 can reach a final state only if V_1 has emptied the stack.

b) Let $V_2 = (Q, \Sigma, W, E, q_0, z_0, F)$ be a pushdown automaton, which accepts language L by final state. Define pushdown automaton $V_1 = (Q \cup \{p_0, p\}, \Sigma, W \cup \{x\}, E', p_0, x, \emptyset)$, where $p_0, p \notin Q$, $x \notin W$ and

$$E' = E \cup \Big\{\big(p_0, (\varepsilon, x/xz_0), q_0\big)\Big\} \cup \Big\{\big(q, (\varepsilon, z/\varepsilon), p\big) \mid q \in F, p \in Q, z \in W\Big\}$$
$$\cup \Big\{\big(p, (\varepsilon, z/\varepsilon), p\big) \mid p \in Q, z \in W \cup \{x\}\Big\}$$

Working V_1: Pushdown automaton V_1 with an ε-move writes in the stack beside x the initial stack symbol z_0 of V_2, then works as V_2, i.e reaches a final state for each accepted word. After this V_1 empties the stack by an ε-move. V_1 can empty the stack only if V_2 goes in a final state. ■

The next two theorems prove that the class of languages accepted by nondeterministic pushdown automata is just the set of context-free languages.

Theorem 1.25 *If G is a context-free grammar, then there exists such a nondeterministic pushdown automaton V which accepts $L(G)$ by empty stack, i.e. $L_\varepsilon(V) = L(G)$.*

We outline the proof only. Let $G = (N, T, P, S)$ be a context-free grammar. Define pushdown automaton $V = (\{q\}, T, N \cup T, E, q, S, \emptyset)$, where $q \notin N \cup T$, and the set E of transitions is:

• If there is in the set of productions of G a production of type $A \to \alpha$, then let put in E the transition $\big(q, (\varepsilon, A/\alpha^{-1}), q\big)$,

• For any letter $a \in T$ let put in E the transition $\big(q, (a, a/\varepsilon), q\big)$.

If there is a production $S \to \alpha$ in G, the pushdown automaton put in the stack the mirror of α with an ε-move. If the input letter coincides with that in the top of the stack, then the automaton deletes it from the stack. If in the top of the stack there is a nonterminal A, then the mirror of right-hand side of a production which has A in its left-hand side will be put in the stack. If after reading all letters of the input word, the stack will be empty, then the pushdown automaton recognized the input word.

The following algorithm builds for a context-free grammar $G = (N, T, P, S)$ the pushdown automaton $V = (\{q\}, T, N \cup T, E, q, S, \emptyset)$, which accepts by empty stack the language generated by G.

FROM-CFG-TO-PUSHDOWN-AUTOMATON(G)

1 **for** all production $A \to \alpha$
  do put in E the transition $\big(q, (\varepsilon, A/\alpha^{-1}), q\big)$
2 **for** all terminal $a \in T$
  do put in E the transition $\big(q, (a, a/\varepsilon), q\big)$
3 **return** V

If G has n productions and m terminals, then the number of step of the algorithm is $\Theta(n + m)$.

Example 1.28 Let $G = (\{S, A\}, \{a, b\}, \{S \to \varepsilon,\ S \to ab,\ S \to aAb,\ A \to aAb,\ A \to ab\}, S)$. Then $V = (\{q\}, \{a, b\}, \{a, b, A, S\}, E, q, S, \emptyset)$, with the following transition table.

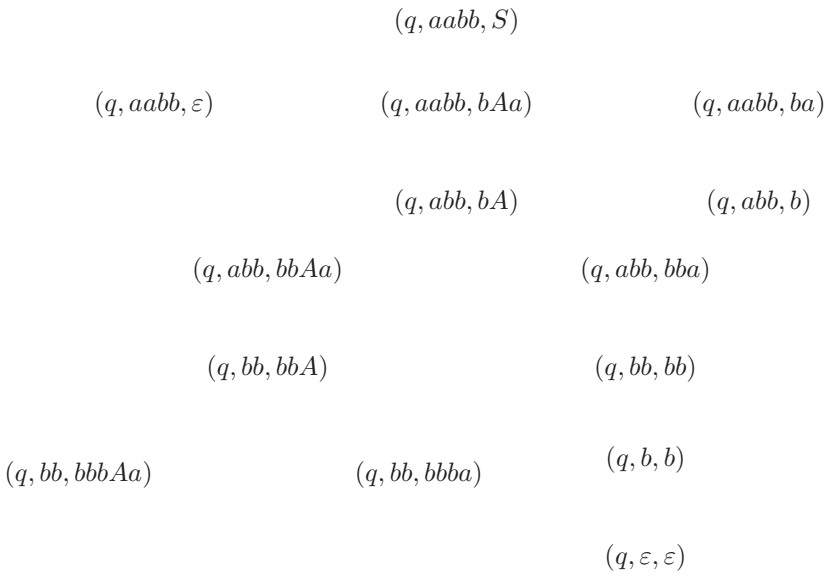


Figure 1.36. Recognising a word by empty stack (see Example 1.28).

$\Sigma \cup \{\varepsilon\}$	a	b	ε	
W	a	b	S	A
q	(ε, q)	(ε, q)	(ε, q) (ba, q) (bAa, q)	(bAa, q) (ba, q)

Let us see how pushdown automaton V accepts word $aabb$, which in grammar G can be derived in the following way:

$$S \Longrightarrow aAb \Longrightarrow aabb,$$

where productions $S \to aAb$ and $A \to ab$ were used. Word is accepted by empty stack (see Fig. 1.36).

Theorem 1.26 *For a nondeterministic pushdown automaton* V *there exists always a context-free grammar* G *such that* V *accepts language* $L(G)$ *by empty stack, i.e.* $L_\varepsilon(\mathrm{V}) = L(G)$.

Instead of a proof we will give a method to obtain grammar G. Let $\mathrm{V} = (Q, \Sigma, W, E, q_0, z_0, \emptyset)$ be the nondeterministic pushdown automaton in question.

Then $G = (N, T, P, S)$, where
$N = \{S\} \cup \{S_{p,z,q} \mid p, q \in Q, z \in W\}$ and $T = \Sigma$.
Productions in P will be obtained as follows.
• For all state q put in P production $S \to S_{q_0,z_0,q}$.

• If $\big(q,(a,z/z_k\dots z_2z_1),p\big)\in E$, where $q\in Q$, $z,z_1,z_2,\dots z_k\in W$ $(k\geq 1)$ and $a\in\Sigma\cup\{\varepsilon\}$, put in P for all possible states $p_1,p_2,\dots,p_k$ productions

$S_{q,z,p_k}\to aS_{p,z_1,p_1}S_{p_1,z_2,p_2}\dots S_{p_{k-1},z_k,p_k}$.

• If $\big(q,(a,z/\varepsilon),p\big)\in E$, where $p,q\in Q, z\in W$, and $a\in\Sigma\cup\{\varepsilon\}$, put in P production

$S_{q,z,p}\to a$.

The context-free grammar defined by this is an extended one, to which an equivalent context-free language can be associated. The proof of the theorem is based on the fact that to every sequence of configurations, by which the pushdown automaton V accepts a word, we can associate a derivation in grammar G. This derivation generates just the word in question, because of productions of the form $S_{q,z,p_k}\to aS_{p,z_1,p_1}S_{p_1,z_2,p_2}\dots S_{p_{k-1},z_k,p_k}$, which were defined for all possible states $p_1,p_2,\dots,p_k$. In Example 1.27 we show how can be associated a derivation to a sequence of configurations. The pushdown automaton defined in the example recognizes word 00 by the sequence of configurations

$(q_0,00,z_0)\Longrightarrow(q_0,0,z_0z_1)\Longrightarrow(q_1,\varepsilon,z_0)\Longrightarrow(q_1,\varepsilon,\varepsilon)$,

which sequence is based on the transitions

$\big(q_0,(0,z_0/z_0z_1),q_0\big)$,
$\big(q_0,(0,z_1/\varepsilon),q_1\big)$,
$\big(q_1,(\varepsilon,z_1/\varepsilon),q_1\big)$.

To these transitions, by the definition of grammar G, the following productions can be associated

(1) $S_{q_0,z_0,p_2}\longrightarrow 0S_{q_0,z_1,p_1}S_{p_1,z_0,p_2}$ for all states $p_1,p_2\in Q$,
(2) $S_{q_0,z_1,q_1}\longrightarrow 0$,
(3) $S_{q_1,z_0,q_1}\longrightarrow\varepsilon$.

Furthermore, for each state q productions $S\longrightarrow S_{q_0,z_0,q}$ were defined.

By the existence of production $S\longrightarrow S_{q_0,z_0,q}$ there exists the derivation $S\Longrightarrow S_{q_0,z_0,q}$, where q can be chosen arbitrarily. Let choose in above production (1) state q to be equal to p_2. Then there exists also the derivation

$S\Longrightarrow S_{q_0,z_0,q}\Longrightarrow 0S_{q_0,z_1,p_1}S_{p_1,z_0,q}$,

where $p_1\in Q$ can be chosen arbitrarily. If $p_1=q_1$, then the derivation

$S\Longrightarrow S_{q_0,z_0,q}\Longrightarrow 0S_{q_0,z_1,q_1}S_{q_1,z_0,q}\Longrightarrow 00S_{q_1,z_0,q}$

will result. Now let q equal to q_1, then

$S\Longrightarrow S_{q_0,z_0,q_1}\Longrightarrow 0S_{q_0,z_1,q_1}S_{q_1,z_0,q_1}\Longrightarrow 00S_{q_1,z_0,q_1}\Longrightarrow 00$,

which proves that word 00 can be derived used the above grammar.

The next algorithm builds for a pushdown automaton $\mathrm{V}=(Q,\Sigma,W,E,q_0,z_0,\emptyset)$ a context-free grammar $G=(N,T,P,S)$, which generates the language accepted by pushdown automaton V by empty stack.

FROM-PUSHDOWN-AUTOMATON-TO-CF-GRAMMAR(V, G)

1 **for** all $q \in Q$
2 **do** put in P production $S \to S_{q_0,z_0,q}$
3 **for** all $\big(q,(a,z/z_k \dots z_2 z_1),p\big) \in E \rhd q \in Q,\ \ z,z_1,z_2,\dots z_k \in W\ (k \geq 1),\ a \in \Sigma \cup \{\varepsilon\}$
4 **do for** all states $p_1, p_2, \dots, p_k$
5 **do** put in P productions $S_{q,z,p_k} \to a S_{p,z_1,p_1} S_{p_1,z_2,p_2} \dots S_{p_{k-1},z_k,p_k}$
6 **for** All $\big(q(a,z/\varepsilon),p\big) \in E$ $\rhd\ p,q \in Q, z \in W,\ a \in \Sigma \cup \{\varepsilon\}$
7 **do** put in P production $S_{q,z,p} \to a$

If the automaton has n states and m productions, then the above algorithm executes at most $n + mn + m$ steps, so in worst case the number of steps is $O(nm)$.

Finally, without proof, we mention that the class of languages accepted by deterministic pushdown automata is a proper subset of the class of languages accepted by nondeterministic pushdown automata. This points to the fact that pushdown automata behave differently as finite automata.

Example 1.29 As an example, consider pushdown automaton V from the Example 1.28: $\mathrm{V} = (\{q\}, \{a,b\}, \{a,b,A,S\}, E, q, S, \emptyset)$. Grammar G is:

$$G = (\{S, S_a, S_b, S_S, S_A, \}, \{a,b\}, P, S)\ ,$$

where for all $z \in \{a,b,S,A\}$ instead of $S_{q,z,q}$ we shortly used S_z. The transitions:

$$\begin{array}{lll} q,(a,a/\varepsilon),q\ , & q,(b,b/\varepsilon),q\ , & \\ q,(\varepsilon,S/\varepsilon),q\ , & q,(\varepsilon,S/ba),q\ , & q,(\varepsilon,S/bAa),q\ , \\ q,(\varepsilon,A/ba),q\ , & q,(\varepsilon,A/bAa),q\ . & \end{array}$$

Based on these, the following productions are defined:

$S \ \to S_S$
$S_a \to a$
$S_b \to b$
$S_S \to \varepsilon \mid S_a S_b \mid S_a S_A S_b$
$S_A \to S_a S_A S_b \mid S_a S_b$.

It is easy to see that S_S can be eliminated, and the productions will be:

$S \ \to \varepsilon \mid S_a S_b \mid S_a S_A S_b$,
$S_A \to S_a S_A S_b \mid S_a S_b$,
$S_a \to a,\ \ S_b \to b$,

and these productions can be replaced:

$S \to \varepsilon \mid ab \mid aAb$,
$A \to aAb \mid ab$.

1.3.2. Context-free languages

Consider context-free grammar $G = (N, T, P, S)$. A ***derivation tree*** of G is a finite, ordered, labelled tree, which root is labelled by the the start symbol S, every interior vertex is labelled by a nonterminal and every leaf by a terminal. If an interior vertex labelled by a nonterminal A has k descendents, then in P there exists a production $A \to a_1 a_2 \dots a_k$ such that the descendents are labelled by letters a_1, a_2, $\dots a_k$. The

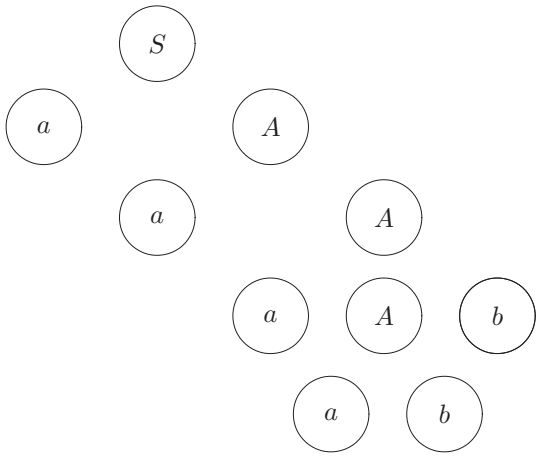


Figure 1.37. Derivation (or syntax) tree of word *aaaabb*.

result of a derivation tree is a word over T, which can be obtained by reading the labels of the leaves from left to right. Derivation tree is also called ***syntax tree***.

Consider the context-free grammar $G = (\{S, A\}, \{a, b\}, \{S \to aA,\ S \to a,\ S \to \varepsilon,\ A \to aA,\ A \to aAb,\ A \to ab,\ A \to b\}, S)$. It generates language $L(G) = \{a^n b^m \mid n \geq m \geq 0\}$. Derivation of word $a^4b^2 \in L(G)$ is:

$$S \Longrightarrow aA \Longrightarrow aaA \Longrightarrow aaaAb \Longrightarrow aaaabb.$$

In Fig. 1.37 this derivation can be seen, which result is *aaaabb*.

To every derivation we can associate a syntax tree. Conversely, to any syntax tree more than one derivation can be associated. For example to syntax tree in Fig. 1.37 the derivation

$$S \Longrightarrow aA \Longrightarrow aaAb \Longrightarrow aaaAb \Longrightarrow aaaabb$$

also can be associated.

Definition 1.27 *Derivation* $\alpha_0 \Longrightarrow \alpha_1 \Longrightarrow \ldots \Longrightarrow \alpha_n$ *is a* ***leftmost derivation****, if for all* $i = 1, 2, \ldots, n-1$ *there exist words* $u_i \in T^*$, $\beta_i \in (N \cup T)^*$ *and productions* $(A_i \to \gamma_i) \in P$, *for which we have*

$$\alpha_i = u_i A_i \beta_i \quad \text{and} \quad \alpha_{i+1} = u_i \gamma_i \beta_i.$$

Consider grammar:

$G = (\{S, A\}, \{a, b, c\}, \{S \to bA,\ S \to bAS,\ S \to a,\ A \to cS,\ A \to a\}, S)$.

In this grammar word *bcbaa* has two different leftmost derivations:

$S \Longrightarrow bA \Longrightarrow bcS \Longrightarrow bcbAS \Longrightarrow bcbaS \Longrightarrow bcbaa$,

$S \Longrightarrow bAS \Longrightarrow bcSS \Longrightarrow bcbAS \Longrightarrow bcbaS \Longrightarrow bcbaa$.

Definition 1.28 *A context-free grammar* G *is* ***ambiguous*** *if in* $L(G)$ *there exists a word with more than one leftmost derivation. Otherwise* G *is* ***unambiguous***.

The above grammar G is ambiguous, because word *bcbaa* has two different leftmost derivations. A language can be generated by more than one grammar, and

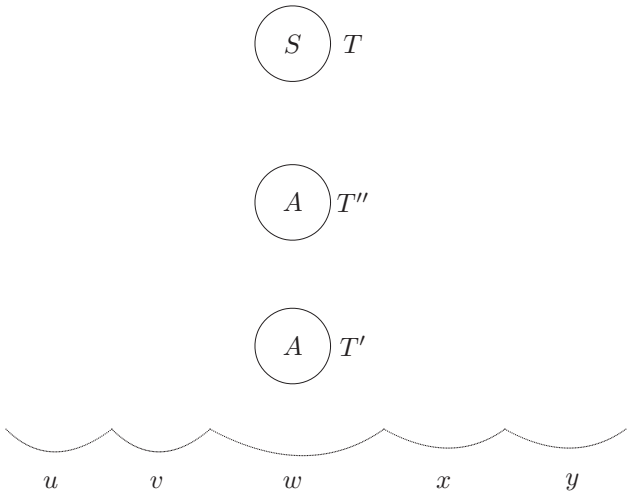


Figure 1.38. Decomposition of tree in the proof of pumping lemma.

between them can exist ambiguous and unambiguous too. A context-free language is ***inherently ambiguous,*** if there is no unambiguous grammar which generates it.

Example 1.30 Examine the following two grammars.
Grammar $G_1 = (\{S\}, \{a, +, *\}, \{S \to S + S,\ S \to S * S,\ S \to a\}, S)$ is ambiguous because
$S \Longrightarrow S + S \Longrightarrow a + S \Longrightarrow a + S * S \Longrightarrow a + a * S \Longrightarrow a + a * S + S \Longrightarrow a + a * a + S \Longrightarrow a + a * a + a$ and
$S \Longrightarrow S * S \Longrightarrow S + S * S \Longrightarrow a + S * S \Longrightarrow a + a * S \Longrightarrow a + a * S + S \Longrightarrow a + a * a + S \Longrightarrow a + a * a + a.$
Grammar $G_2 = (\{S, A\}, \{a, *, +\}, \{S \to A + S \mid A,\ A \to A * A \mid a\}, S)$ is unambiguous.
Can be proved that $L(G_1) = L(G_2)$.

1.3.3. Pumping lemma for context-free languages

Like for regular languages there exists a pumping lemma also for context-free languages.

Theorem 1.29 (pumping lemma). *For any context-free language L there exists a natural number n (which depends only on L), such that every word z of the language longer than n can be written in the form $uvwxy$ and the following are true:*
(1) $|w| \geq 1$,
(2) $|vx| \geq 1$,
(3) $|vwx| \leq n$,
(4) uv^iwx^iy *is also in L for all* $i \geq 0$.

Proof Let $G = (N, T, P, S)$ be a grammar without unit productions, which generates language L. Let $m = |N|$ be the number of nonterminals, and let ℓ be the maximum of lengths of right-hand sides of productions, i.e. $\ell = \max\{|\alpha| \mid \exists A \in N : (A \to \alpha) \in$

$P\}$. Let $n = \ell^{m+1}$ and $z \in L(G)$, such that $|z| > n$. Then there exists a derivation tree T with the result z. Let h be the height of T (the maximum of path lengths from root to leaves). Because in T all interior vertices have at most ℓ descendents, T has at most ℓ^h leaves, i.e. $|z| \leq \ell^h$. On the other hand, because of $|z| > \ell^{m+1}$, we get that $h > m+1$. From this follows that in derivation tree T there is a path from root to a leave in which there are more than $(m+1)$ vertices. Consider such a path. Because in G the number of nonterminals is m and on this path vertices different from the leaf are labelled with nonterminals, by the pigeonhole principle, it must be a nonterminal on this path which occurs at least twice.

Let us denote by A the nonterminal being the first on this path from root to the leaf which firstly repeat. Denote by T' the subtree, which root is this occurrence of A. Similarly, denote by T'' the subtree, which root is the second occurrence of A on this path. Let w be the result of the tree T'. Then the result of T'' is in form vwx, while of T in $uvwxy$. Derivation tree T with this decomposition of z can be seen in Fig. 1.38. We show that this decomposition of z satisfies conditions (1)–(4) of lemma.

Because in P there are no ε-productions (except maybe the case $S \to \varepsilon$), we have $|w| \geq 1$. Furthermore, because each interior vertex of the derivation tree has at least two descendents (namely there are no unit productions), also the root of T'' has, hence $|vx| \geq 1$. Because A is the first repeated nonterminal on this path, the height of T'' is at most $m+1$, and from this $|vwx| \leq \ell^{m+1} = n$ results.

After eliminating from T all vertices of T'' excepting the root, the result of obtained tree is uAy, i.e. $S \overset{*}{\underset{G}{\Longrightarrow}} uAy$.

Similarly, after eliminating T' we get $A \overset{*}{\underset{G}{\Longrightarrow}} vAx$, and finally because of the definition of T' we get $A \overset{*}{\underset{G}{\Longrightarrow}} w$. Then $S \overset{*}{\underset{G}{\Longrightarrow}} uAy$, $A \overset{*}{\underset{G}{\Longrightarrow}} vAx$ and $A \overset{*}{\underset{G}{\Longrightarrow}} w$. Therefore $S \overset{*}{\underset{G}{\Longrightarrow}} uAy \overset{*}{\underset{G}{\Longrightarrow}} uwy$ and $S \overset{*}{\underset{G}{\Longrightarrow}} uAy \overset{*}{\underset{G}{\Longrightarrow}} uvAxy \overset{*}{\underset{G}{\Longrightarrow}} \ldots \overset{*}{\underset{G}{\Longrightarrow}} uv^iAx^iy \overset{*}{\underset{G}{\Longrightarrow}} uv^iwx^iy$ for all $i \geq 1$. Therefore, for all $i \geq 0$ we have $S \overset{*}{\Longrightarrow} uv^iwx^iy$, i.e. for all $i \geq 0$ $uv^iwx^iy \in L(G)$. ■

Now we present two consequences of the lemma.

Corollary 1.30 $\mathcal{L}_2 \subset \mathcal{L}_1$.

Proof This consequence states that there exists a context-sensitive language which is not context-free. To prove this it is sufficient to find a context-sensitive language for which the lemma is not true. Let this language be $L = \{a^m b^m c^m \mid m \geq 1\}$.

To show that this language is context-sensitive it is enough to give a convenient grammar. In Example 1.2 both grammars are extended context-sensitive, and we know that to each extended grammar of type i an equivalent grammar of the same type can be associated.

Let n be the natural number associated to L by lemma, and consider the word $z = a^n b^n c^n$. Because of $|z| = 3n > n$, if L is context-free z can be decomposed in $z = uvwxy$ such that conditions (1)–(4) are true. We show that this leads us to a contradiction.

Firstly, we will show that word v and x can contain only one type of letters. Indeed if either v or x contain more than one type of letters, then in word $uvvwxxy$

the order of the letters will be not the order a, b, c, so $uvvwxxy \notin L(G)$, which contradicts condition (4) of lemma.

If both v and x contain at most one type of letters, then in word uwy the number of different letters will be not the same, so $uwy \notin L(G)$. This also contradicts condition (4) in lemma. Therefore L is not context-free. ■

Corollary 1.31 *The class of context-free languages is not closed under the intersection.*

Proof We give two context-free languages which intersection is not context-free. Let $N = \{S, A, B\}$, $T = \{a, b, c\}$ and

$G_1 = (N, T, P_1, S)$ where P_1 :
$S \to AB$,
$A \to aAb \mid ab$,
$B \to cB \mid c$,

and $G_2 = (N, T, P_2, S)$, where P_2 :
$S \to AB$,
$A \to Aa \mid a$,
$B \to bBc \mid bc$.

Languages $L(G_1) = \{a^n b^n c^m \mid n \geq 1, m \geq 1\}$ and $L(G_2) = \{a^n b^m c^m \mid n \geq 1, m \geq 1\}$ are context-free. But

$$L(G_1) \cap L(G_2) = \{a^n b^n c^n \mid n \geq 1\}$$

is not context-free (see the proof of the Consequence 1.30). ■

1.3.4. Normal forms of the context-free languages

In the case of arbitrary grammars the normal form was defined (see page 20) as grammars with no terminals in the left-hand side of productions. The normal form in the case of the context-free languages will contains some restrictions on the right-hand sides of productions. Two normal forms (Chomsky and Greibach) will be discussed.

Chomsky normal form

Definition 1.32 *A context-free grammar $G = (N, T, P, S)$ is in Chomsky normal form, if all productions have form $A \to a$ or $A \to BC$, where $A, B, C \in N$, $a \in T$.*

Example 1.31 Grammar $G = (\{S, A, B, C\}, \{a, b\}, \{S \to AB,\ S \to CB,\ C \to AS,\ A \to a,\ B \to b\}, S)$ is in Chomsky normal form and $L(G) = \{a^n b^n \mid n \geq 1\}$.

To each ε-free context-free language can be associated an equivalent grammar is Chomsky normal form. The next algorithm transforms an ε-free context-free grammar $G = (N, T, P, S)$ in grammar $G' = (N', T, P', S)$ which is in Chomsky normal form.

CHOMSKY-NORMAL-FORM(G)

1 $N' \leftarrow N$
2 eliminate unit productions, and let P' the new set of productions
(see algorithm ELIMINATE-UNIT-PRODUCTIONS on page 20)
3 in P' replace in each production with at least two letters in right-hand side
all terminals a by a new nonterminal A, and add this nonterminal to N'
and add production $A \to a$ to P'
4 replace all productions $B \to A_1 A_2 \dots A_k$, where $k \geq 3$ and $A_1, A_2, \dots, A_k \in N$,
by the following:

$$\begin{aligned} B &\to A_1 C_1, \\ C_1 &\to A_2 C_2, \\ &\dots \\ C_{k-3} &\to A_{k-2} C_{k-2}, \\ C_{k-2} &\to A_{k-1} A_k, \end{aligned}$$

where $C_1, C_2, \dots, C_{k-2}$ are new nonterminals, and add them to N'
5 **return** G'

Example 1.32 Let $G = (\{S, D\}, \{a, b, c\}, \{S \to aSc,\ S \to D,\ D \to bD,\ D \to b\}, S)$. It is easy to see that $L(G) = \{a^n b^m c^n \mid n \geq 0, m \geq 1\}$. Steps of transformation to Chomsky normal form are the following:

Step 1: $N' = \{S, D\}$
Step 2: After eliminating the unit production $S \to D$ the productions are:
$S \to aSc \mid bD \mid b$,
$D \to bD \mid b$.
Step 3: We introduce three new nonterminals because of the three terminals in productions. Let these be A, B, C. Then the production are:
$S \to ASC \mid BD \mid b$,
$D \to BD \mid b$,
$A \to a$,
$B \to b$,
$C \to c$.
Step 4: Only one new nonterminal (let this E) must be introduced because of a single production with three letters in the right-hand side. Therefore $N' = \{S, A, B, C, D, E\}$, and the productions in P' are:
$S \to AE \mid BD \mid b$,
$D \to BD \mid b$,
$A \to a$,
$B \to b$,
$C \to c$,
$E \to SC$.
All these productions are in required form.

Greibach normal form 1mm

Definition 1.33 *A context-free grammar $G = (N, T, P, S)$ is in **Greibach normal form** if all production are in the form $A \to aw$, where $A \in N$, $a \in T$, $w \in N^*$.*

Example 1.33 Grammar $G = (\{S, B\}, \{a, b\}, \{S \to aB,\ S \to aSB,\ B \to b\}, S)$ is in Greibach normal form and $L(G) = \{a^n b^n \mid n \geq 1\}$.

To each ε-free context-free grammar an equivalent grammar in Greibach normal form can be given. We give and algorithm which transforms a context-free grammar $G = (N, T, P, S)$ in Chomsky normal form in a grammar $G' = (N', T, P', S)$ in Greibach normal form.

First, we give an order of the nonterminals: $A_1, A_2, \ldots, A_n$, where A_1 is the start symbol. The algorithm will use the notations $x \in N'^+$, $\alpha \in TN'^* \cup N'^+$.

GREIBACH-NORMAL-FORM(G)

1 $N' \leftarrow N$
2 $P' \leftarrow P$
3 **for** $i \leftarrow 2$ **to** n ▷ Case $A_i \to A_j x,\ j < i$
4 **do for** $j \leftarrow 1$ **to** $i - 1$
5 **do** for all productions $A_i \to A_j x$ and $A_j \to \alpha$ (where α has no A_j as first letter) in P' productions $A_i \to \alpha x$, delete from P' productions $A_i \to A_j x$
6 **if** there is a production $A_i \to A_i x$ ▷ Case $A_i \to A_i x$
7 **then** put in N' the new nonterminal B_i, for all productions $A_i \to A_i x$ put in P' productions $B_i \to xB_i$ and $B_i \to x$,delete from P' production $A_i \to A_i x$, for all production $A_i \to \alpha$ (where A_i is not the first letter of α) put in P' production $A_i \to \alpha B_i$
8 **for** $i \leftarrow n - 1$ **downto** 1 ▷ Case $A_i \to A_j x,\ j > i$
9 **do for** $j \leftarrow i + 1$ **to** n
10 **do** for all productions $A_i \to A_j x$ and $A_j \to \alpha$ put in P' production $A_i \to \alpha x$ and delete from P' productions $A_i \to A_j x$,
11 **for** $i \leftarrow 1$ **to** n ▷ Case $B_i \to A_j x$
12 **do for** $j \leftarrow 1$ **to** n
13 **do** for all productions $B_i \to A_j x$ and $A_j \to \alpha$ put in P' production $B_i \to \alpha x$ and delete from P' productions $B_i \to A_j x$
14 **return** G'

The algorithm first transform productions of the form $A_i \to A_j x,\ j < i$ such that $A_i \to A_j x,\ j \geq i$ or $A_i \to \alpha$, where this latter is in Greibach normal form. After this, introducing a new nonterminal, eliminate productions $A_i \to A_i x$, and using substitutions all production of the form $A_i \to A_j x,\ j > i$ and $B_i \to A_j x$ will be transformed in Greibach normal form.

Example 1.34 Transform productions in Chomsky normal form

$A_1 \to A_2A_3 \mid A_2A_4$
$A_2 \to A_2A_3 \mid a$
$A_3 \to A_2A_4 \mid b$
$A_4 \to c$

in Greibach normal form.

Steps of the algorithm:

3–5: Production $A_3 \to A_2A_4$ must be transformed. For this production $A_2 \to a$ is appropriate. Put $A_3 \to aA_4$ in the set of productions and eliminate $A_3 \to A_2A_4$.

The productions will be:

$A_1 \to A_2A_3 \mid A_2A_4$
$A_2 \to A_2A_3 \mid a$
$A_3 \to aA_4 \mid b$
$A_4 \to c$

6-7: Elimination of production $A_2 \to A_2A_3$ will be made using productions:

$B_2 \to A_3B_2$
$B_2 \to A_3$
$A_2 \to aB_2$

Then, after steps 6–7. the productions will be:

$A_1 \to A_2A_3 \mid A_2A_4$
$A_2 \to aB_2 \mid a$
$A_3 \to aA_4 \mid b$
$A_4 \to c$
$B_2 \to A_3B_2 \mid A_3$

8–10: We make substitutions in productions with A_1 in left-hand side. The results is:

$A_1 \to aA_3 \mid aB_2A_3 \mid aA_4 \mid aB_2A_4$

11–13: Similarly with productions with B_2 in left-hand side:

$B_2 \to aA_4B_2 \mid aA_3A_4B_2 \mid aA_4 \mid aA_3A_4$

After the elimination in steps 8–13 of productions in which substitutions were made, the following productions, which are now in Greibach normal form, result:

$A_1 \to aA_3 \mid aB_2A_3 \mid aA_4 \mid aB_2A_4$
$A_2 \to aB_2 \mid a$
$A_3 \to aA_4 \mid b$
$A_4 \to c$
$B_2 \to aA_4B_2 \mid aA_3A_4B_2 \mid aA_4 \mid aA_3A_4$

Example 1.35 Language

$$L = \ a^n b^k c^{n+k} \mid n \geq 0, k \geq 0, n+k > 0$$

can be generated by grammar

$$G = \ \{S, R\}, \{a, b, c\}, \{S \to aSc,\ S \to ac,\ S \to R, R \to bRc,\ R \to bc\}, S$$

First, will eliminate the single unit production, and after this we will give an equivalent grammar in Chomsky normal form, which will be transformed in Greibach normal form.

Productions after the elimination of production $S \to R$:

$S \to aSc \mid ac \mid bRc \mid bc$
$R \to bRc \mid bc.$

We introduce productions $A \to a, B \to b, C \to c$, and replace terminals by the corresponding nonterminals:

$S \to ASC \mid AC \mid BRC \mid BC$,
$R \to BRC \mid BC$,
$A \to a,\ B \to b,\ C \to c$.

After introducing two new nonterminals (D, E):

$S \to AD \mid AC \mid BE \mid BC$,
$D \to SC$,
$E \to RC$,
$R \to BE \mid BC$,
$A \to a,\ B \to b,\ C \to c$.

This is now in Chomsky normal form. Replace the nonterminals to be letters A_i as in the algorithm. Then, after applying the replacements

S replaced by A_1, A replaced by A_2, B replaced by A_3, C replaced by A_4, D replaced by A_5,
E replaced by A_6, R replaced by A_7,

our grammar will have the productions:

$A_1 \to A_2A_5 \mid A_2A_4 \mid A_3A_6 \mid A_3A_4$,
$A_2 \to a,\ A_3 \to b,\ A_4 \to c$,
$A_5 \to A_1A_4$,
$A_6 \to A_7A_4$,
$A_7 \to A_3A_6 \mid A_3A_4$.

In steps 3–5 of the algorithm the new productions will occur:

$A_5 \to A_2A_5A_4 \mid A_2A_4A_4 \mid A_3A_6A_4 \mid A_3A_4A_4$ then
$A_5 \to aA_5A_4 \mid aA_4A_4 \mid bA_6A_4 \mid bA_4A_4$
$A_7 \to A_3A_6 \mid A_3A_4$, then
$A_7 \to bA_6 \mid bA_4$.

Therefore

$A_1 \to A_2A_5 \mid A_2A_4 \mid A_3A_6 \mid A_3A_4$,
$A_2 \to a,\ A_3 \to b,\ A_4 \to c$,
$A_5 \to aA_5A_4 \mid aA_4A_4 \mid bA_6A_4 \mid bA_4A_4$
$A_6 \to A_7A_4$,
$A_7 \to bA_6 \mid bA_4$.

Steps 6–7 will be skipped, because we have no left-recursive productions. In steps 8–10 after the appropriate substitutions we have:

$A_1 \to aA_5 \mid aA_4 \mid bA_6 \mid bA_4$,
$A_2 \to a$,
$A_3 \to b$,
$A_4 \to c$,
$A_5 \to aA_5A_4 \mid aA_4A_4 \mid bA_6A_4 \mid bA_4A_4$
$A_6 \to bA_6A_4 \mid bA_4A_4$,
$A_7 \to bA_6 \mid bA_4$.

Exercises

1.3-1 Give pushdown automata to accept the following languages:

$L_1 = \{a^n c b^n \mid n \geq 0\}$,
$L_2 = \{a^n b^{2n} \mid n \geq 1\}$,
$L_3 = \{a^{2n} b^n \mid n \geq 0\} \cup \{a^n b^{2n} \mid n \geq 0\}$,

1.3-2 Give a context-free grammar to generate language $L = \{a^n b^n c^m \mid n \geq 0, m \geq 0\}$, and transform it in Chomsky and Greibach normal forms. Give a pushdown automaton which accepts L.

1.3-3 What languages are generated by the following context-free grammars?
$G_1 = \big(\{S\}, \{a, b\}, \{S \to SSa, \ \to b\}, S\big)$, $G_2 = \big(\{S\}, \{a, b\}, \{S \to SaS, \ \to b\}, S\big)$.

1.3-4 Give a context-free grammar to generate words with an equal number of letters a and b.

1.3-5 Prove, using the pumping lemma, that a language whose words contains an equal number of letters a, b and c can not be context-free.

1.3-6 Let the grammar $G = (V, T, P, S)$, where
$V = \{S\}$,
$T = \{\textit{if, then, else, a, c}\}$,
$P = \{S \to \textit{if a then } S, \quad S \to \textit{if a then } S \textit{ else } S, \quad S \to c\}$,
Show that word *if a then if a then c else c* has two different leftmost derivations.

1.3-7 Prove that if L is context-free, then $L^{-1} = \{u^{-1} \mid u \in L\}$ is also context-free.

Problems

1-1 Linear grammars

A grammar $G = (N, T, P, S)$ which has productions only in the form $A \to u_1 B u_2$ or $A \to u$, where $A, B \in N$, $u, u_1, u_2 \in T^*$, is called a ***linear grammar***. If in a linear grammar all production are of the form $A \to Bu$ or $A \to v$, then it is called a left-linear grammar. Prove that the language generated by a left-linear grammar is regular.

1-2 Operator grammars

An ε-free context-free grammar is called ***operator grammar*** if in the right-hand side of productions there are no two successive nonterminals. Show that, for all ε-free context-free grammar an equivalent operator grammar can be built.

1-3 Complement of context-free languages

Prove that the class of context-free languages is not closed on complement.

Chapter notes

In the definition of finite automata instead of transition function we have used the transition graph, which in many cases help us to give simpler proofs.

There exist a lot of classical books on automata and formal languages. We mention from these the following: two books of Aho and Ullman [5, 6] in 1972 and 1973, book of Gécseg and Peák [78] in 1972, two books of Salomaa [207, 208] in 1969 and 1973, a book of Hopcroft and Ullman [112] in 1979, a book of Harrison [103] in 1978, a book of Manna [160], which in 1981 was published also in Hungarian. We notice also a book of Sipser [228] in 1997 and a monograph of Rozenberg and Salomaa [206]. In a book of Lothaire (common name of French authors) [153] on combinatorics of words we can read on other types of automata. Paper of Giammarresi and Montalbano [83] generalise the notion of finite automata. A new monograph is of

Hopcroft, Motwani and Ullman [111]. In German we recommend the student book of Asteroth and Baier [13]. The concise description of the transformation in Greibach normal form is based on this book.

Other books in English: [30, 37, 63, 132, 139, 147, 152, 163, 171, 225, 226, 236, 237].

At the end of the next chapter on compilers another books on the subject are mentioned.

2. Compilers

When a programmer writes down a solution of her problems, she writes a program on a special programming language. These programming languages are very different from the proper languages of computers, from the ***machine languages***. Therefore we have to produce the executable forms of programs created by the programmer. We need a software or hardware tool, that translates the ***source language program*** – written on a high level programming language – to the ***target language program***, a lower level programming language, mostly to a machine code program.

There are two fundamental methods to execute a program written on higher level language. The first is using an ***interpreter***. In this case, the generated machine code is not saved but executed immediately. The interpreter is considered as a special computer, whose machine code is the high level language. Essentially, when we use an interpreter, then we create a two-level machine; its lower level is the real computer, on which the higher level, the interpreter, is built. The higher level is usually realized by a computer program, but, for some programming languages, there are special hardware interpreter machines.

The second method is using a ***compiler*** program. The difference of this method from the first is that here the result of translation is not executed, but it is saved in an intermediate file called ***target program***.

The target program may be executed later, and the result of the program is received only then. In this case, in contrast with interpreters, the times of translation and execution are distinguishable.

In the respect of translation, the two translational methods are identical, since the interpreter and the compiler both generate target programs. For this reason we speak about compilers only. We will deal the these translator programs, called compilers (Figure 2.1).

Figure 2.1. The translator.

Our task is to study the algorithms of compilers. This chapter will care for the translators of high level imperative programming languages; the translational methods of logical or functional languages will not be investigated.

First the structure of compilers will be given. Then we will deal with scanners, that is, lexical analysers. In the topic of parsers – syntactic analysers –, the two most successful methods will be studied: the $LL(1)$ and the $LALR(1)$ parsing methods. The advanced methods of semantic analysis use *O-ATG* grammars, and the task of code generation is also written by this type of grammars. In this book these topics are not considered, nor we will study such important and interesting problems as symbol table handling, error repairing or code optimising. The reader can find very new, modern and efficient methods for these methods in the bibliography.

2.1. The structure of compilers

A compiler translates the source language program (in short, source program) into a target language program (in short, target program). Moreover, it creates a list by which the programmer can check her private program. This list contains the detected errors, too.

Using the notation *program (input)(output)* the compiler can be written by

compiler (source program)(target program, list) .

In the next, the structure of compilers are studied, and the tasks of program elements are described, using the previous notation.

The first program of a compiler transforms the source language program into character stream that is easy to handle. This program is the ***source handler***.

source handler (source program)(character stream).

The form of the source program depends from the operating system. The source handler reads the file of source program using a system, called operating system, and omits the characters signed the end of lines, since these characters have no importance in the next steps of compilation. This modified, "poured" character stream will be the input data of the next steps.

The list created by the compiler has to contain the original source language program written by the programmer, instead of this modified character stream. Hence we define a ***list handler*** program,

list handler (source program, errors)(list) ,

which creates the list according to the file form of the operating system, and puts this list on a secondary memory.

It is practical to join the source handler and the list handler programs, since they have same input files. This program is the ***source handler***.

source handler (source program, errors)(character stream, list) .

The target program is created by the compiler from the generated target code. It is

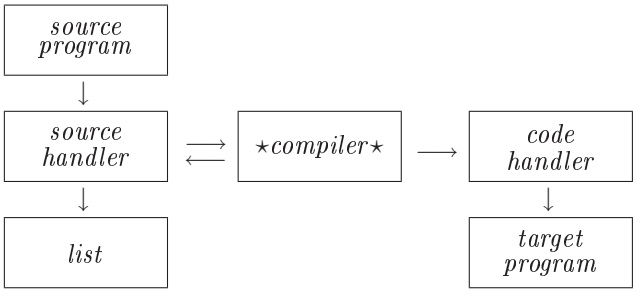


Figure 2.2. The structure of compilers.

located on a secondary memory, too, usually in a transferable binary form. Of course this form depends on the operating system. This task is done by the ***code handler*** program.

code handler (target code)(target program) .

Using the above programs, the structure of a compiler is the following (Figure 2.2):

source handler (source program, errors) (character string, list),
⋆compiler⋆ (character stream)(target code, errors),
code handler (target code)(target program) .

This decomposition is not a sequence: the three program elements are executed not sequentially. The decomposition consists of three independent working units. Their connections are indicated by their inputs and outputs.

In the next we do not deal with the handlers because of their dependentness on computers, operating system and peripherals – although the outer form, the connection with the user and the availability of the compiler are determined mainly by these programs.

The task of the program *⋆compiler⋆* is the translation. It consists of two main subtasks: analysing the input character stream, and to synthetizing the target code.

The first problem of the *analysis* is to determine the connected characters in the character stream. These are the symbolic items, e.g., the constants, names of variables, keywords, operators. This is done by the ***lexical analyser***, in short, ***scanner***. >From the character stream the scanner makes a ***series of symbols*** and during this task it detects ***lexical errors***.

scanner (character stream)(series of symbols, lexical errors) .

This series of symbols is the input of the ***syntactic analyser***, in short, ***parser***. Its task is to check the syntactic structure of the program. This process is near to the checking the verb, the subject, predicates and attributes of a sentence by a language teacher in a language lesson. The errors detected during this analysis are the ***syntactic errors***. The result of the syntactic analysis is the syntax tree of the program, or some similar equivalent structure.

parser (series of symbols)(syntactically analysed program, syntactic errors) .

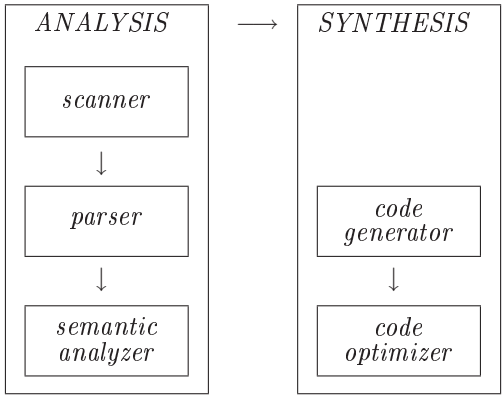


Figure 2.3. The programs of the analysis and the synthesis.

The third program of the analysis is the ***semantic analyser***. Its task is to check the static semantics. For example, when the semantic analyser checks declarations and the types of variables in the expression `a + b`, it verifies whether the variables `a` and `b` are declared, do they are of the same type, do they have values? The errors detected by this program are the ***semantic errors***.

semantic analyser (syntactically analysed program)(analysed program, semantic errors) .

The output of the semantic analyser is the input of the programs of ***synthesis***. The first step of the synthesis is the code generation, that is made by the ***code generator***:

code generator (analysed program)(target code).

The target code usually depends on the computer and the operating system. It is usually an assembly language program or machine code. The next step of synthesis is the ***code optimisation***:

code optimiser (target code)(target code).

The code optimiser transforms the target code on such a way that the new code is better in many respect, for example running time or size.

As it follows from the considerations above, a compiler consists of the next components (the structure of the ⋆compiler⋆ program is in the Figure 2.3):

source handler (source program, errors)(character stream, list),
scanner (character stream)(series of symbols, lexical errors),
parser (series of symbols)(syntactically analysed program, syntactic errors),
semantic analyser (syntactically analysed program)(analysed program,

semantic errors),
code generator (analysed program)(target code),
code optimiser (target code)(target code),
code handler(target code)(target program).

The algorithm of the part of the compiler, that performs analysis and synthesis, is the next:

⋆COMPILER⋆

1 determine the symbolic items in the text of source program
2 check the syntactic correctness of the series of symbols
3 check the semantic correctness of the series of symbols
4 generate the target code
5 optimise the target code

The objects written in the first two points will be analysed in the next sections.

Exercises

2.1-1 Using the above notations, give the structure of interpreters.
2.1-2 Take a programming language, and write program details in which there are lexical, syntactic and semantic errors.
2.1-3 Give respects in which the code optimiser can create better target code than the original.

2.2. Lexical analysis

The *source-handler* transforms the source program into a character stream. The main task of lexical analyser (scanner) is recognising the ***symbolic units*** in this character stream. These symbolic units are named ***symbols***.

Unfortunately, in different programming languages the same symbolic units consist of different character streams, and different symbolic units consist of the same character streams. For example, there is a programming language in which the `1.` and `.10` characters mean real numbers. If we concatenate these symbols, then the result is the `1..10` character stream. The fact, that a sign of an algebraic function is missing between the two numbers, will be detected by the next analyser, doing syntactic analysis. However, there are programming languages in which this character stream is decomposited into three components: 1 and 10 are the lower and upper limits of an interval type variable.

The lexical analyser determines not only the characters of a symbol, but the attributes derived from the surrounded text. Such attributes are, e.g., the type and value of a symbol.

The scanner assigns codes to the symbols, same codes to the same sort of symbols. For example the code of all integer numbers is the same; another unique code is assigned to variables.

The lexical analyser transforms the character stream into the series of symbol

codes and the attributes of a symbols are written in this series, immediately after the code of the symbol concerned.

The output information of the lexical analyser is not „readable": it is usually a series of binary codes. We note that, in the viewpoint of the compiler, from this step of the compilation it is no matter from which characters were made the symbol, i.e. the code of the *if* symbol was made form English *if* or Hungarian *ha* or German *wenn* characters. Therefore, for a program language using English keywords, it is easy to construct another program language using keywords of another language. In the compiler of this new program language the lexical analysis would be modified only, the other parts of the compiler are unchanged.

2.2.1. The automaton of the scanner

The exact definition of symbolic units would be given by *regular grammar*, *regular expressions* or *deterministic finite automaton*. The theories of regular grammars, regular expressions and deterministic finite automata were studied in previous chapters.

Practically the lexical analyser may be a part of the syntactic analysis. The main reason to distinguish these analysers is that a lexical analyser made from regular grammar is much more simpler than a lexical analyser made from a context-free grammar. Context-free grammars are used to create syntactic analysers.

One of the most popular methods to create the lexical analyser is the following:

1. describe symbolic units in the language of regular expressions, and from this information construct the deterministic finite automaton which is equivalent to these regular expressions,

2. implement this deterministic finite automaton.

We note that, in writing of symbols regular expressions are used, because they are more comfortable and readable then regular grammars. There are standard programs as the `lex` of UNIX systems, that generate a complete syntactical analyser from regular expressions. Moreover, there are generator programs that give the automaton of scanner, too.

A very trivial implementation of the deterministic finite automaton uses multidirectional **case** instructions. The conditions of the branches are the characters of state transitions, and the instructions of a branch represent the new state the automaton reaches when it carries out the given state transition.

The main principle of the lexical analyser is building a symbol from the *longest series of symbols*. For example the string `ABC` is a three-letters symbol, rather than three one-letter symbols. This means that the alternative instructions of the **case** branch read characters as long as they are parts of a constructed symbol.

Functions can belong to the final states of the automaton. For example, the function converts constant symbols into an inner binary forms of constants, or the function writes identifiers to the symbol table.

The input stream of the lexical analyser contains tabulators and space characters, since the source-handler expunges the carriage return and line feed characters only.

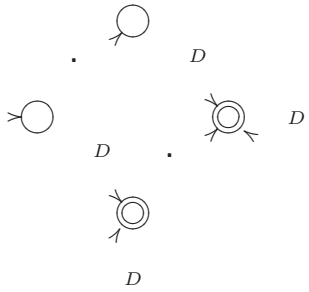


Figure 2.4. The positive integer and real number.

In most programming languages it is possible to write a lot of spaces or tabulators between symbols. In the point of view of compilers these symbols have no importance after their recognition, hence they have the name ***white spaces***.

Expunging white spaces is the task of the lexical analyser. The description of the white space is the following regular expression:

$$(space \mid tab)^* \ ,$$

where *space* and the *tab* tabulator are the characters which build the white space symbols and | is the symbol for the *or* function. No actions have to make with this white space symbols, the scanner does not pass these symbols to the syntactic analyser.

Some examples for regular expression:

Example 2.1 Introduce the following notations: Let D be an arbitrary digit, and let L be an arbitrary letter,

$$D \in \{0, 1, \ldots, 9\}, \text{ and } L \in \{a, b, \ldots, z, A, B, \ldots, Z\} \ ,$$

the not-visible characters are denoted by their short names, and let ε be the name of the empty character stream. $Not(a)$ denotes a character distinct from a. The regular expressions are:

1. real number: $(+ \mid - \mid \varepsilon)D^+.D^+(\mathsf{e}(+ \mid - \mid \varepsilon)D^+ \mid \varepsilon)$,
2. positive integer and real number: $(D^+(\varepsilon \mid .)) \mid (D^*.D^+)$,
3. identifier: $(L \mid _)(L \mid D \mid _)^*$,
4. comment: $\texttt{--}(Not(eol))^* eol$,
5. comment terminated by $\#\#$: $\#\#((\# \mid \varepsilon)Not(\#))^*\#\#$,
6. string of characters: $"(Not(") \mid "")^* "$.

Deterministic finite automata constructed from regular expressions 2 and 3 are in Figures 2.4 and 2.5.

The task of lexical analyser is to determine the text of symbols, but not all the characters of a regular expression belong to the symbol. As is in the 6th example, the first and the last " characters do not belong to the symbol. To unravel this problem,

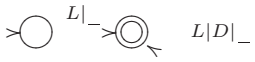


Figure 2.5. The identifier.

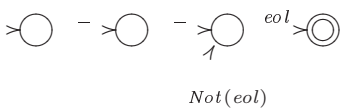


Figure 2.6. A comment.

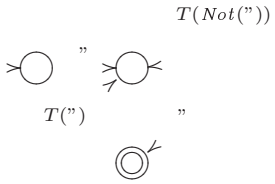


Figure 2.7. The character string.

a buffer is created for the scanner. After recognising of a symbol, the characters of these symbols will be in the buffer. Now the deterministic finite automaton is supplemented by a T transfer function, where $T(a)$ means that the character a is inserted into the buffer.

Example 2.2 The 4th and 6th regular expressions of the example 2.1 are supplemented by the T function, automata for these expressions are in Figures 2.6 and 2.7. The automaton of the 4th regular expression has none T function, since it recognises comments. The automaton of the 6th regular expression recognises `This is a "string"` from the character string `"This is a ""string"""`.

Now we write the algorithm of the lexical analyser given by deterministic finite automaton. (The state of the set of one element will be denoted by the only element of the set).

Let $A = (Q, \Sigma, \delta, q_0, F)$ be the deterministic finite automaton, which is the scanner. We augment the alphabet Σ with a new notion: let *others* be all the characters not in Σ. Accordingly, we modify the transition function δ:

$$\delta'(q,a) = \begin{cases} \delta(q,a), & \text{if } a \neq \mathit{others}\ , \\ \emptyset, & \text{otherwise}\ . \end{cases}$$

The algorithm of parsing, using the augmented automaton A', follows:

LEX-ANALYSE($x\#, A'$)

```
1  q ← q_0, a ← first character of x
2  s' ← analyzing
3  while a ≠ # and s' = analyzing
4        do if δ'(q, a) ≠ ∅
5              then q ← δ'(q, a)
6                   a ← next character of x
7              else  s' ← error
8  if s' = analyzing and q ∈ F
9    then s' ← O.K.
10   else  s' ← ERROR
11 return s', a
```

The algorithm has two parameters: the first one is the input character string terminated by #, the second one is the automaton of the scanner. In the line 1 the state of the scanner is set to q_0, to the start state of the automaton, and the first character of the input string is determined. The variable s' indicates that the algorithm is analysing the input string, the text *analysing* is set in this variable in the line 2. In the line 5 a state-transition is executed. It can be seen that the above augmentation is needed to terminate in case of unexpected, invalid character. In line 8–10 the *O.K.* means that the analysed character string is correct, and the *ERROR* signs that a lexical error was detected. In the case of successful termination the variable a contains the # character, at erroneous termination it contains the invalid character.

We note that the algorithm LEX-ANALYSE recognise one symbol only, and then it is terminated. The program written in a programming language consists of a lot of symbols, hence after recognising a symbol, the algorithm have to be continued by detecting the next symbol. The work of the analyser is restarted at the state of the automaton. We propose the full algorithm of the lexical analyser as an exercise (see Exercise *2-1*).

Example 2.3 The automaton of the identifier in the point 3 of example 2.1 is in Figure 2.5. The start state is 0, and the final state is 1. The transition function of the automaton follows:

δ	L	_	D
0	1	1	$\emptyset$
1	1	1	1

The augmented transition function of the automaton:

δ'	L	_	D	*others*
0	1	1	$\emptyset$	$\emptyset$
1	1	1	1	$\emptyset$

The algorithm LEX-ANALYSE gives the series of states 0111111 and sign *O.K.* to the input string `abc123#`, it gives sign *ERROR* to the input sting `9abc#`, and the series 0111 and sign *ERROR* to the input string `abc`χ`123`.

2.2.2. Special problems

In this subsection we investigate the problems emerged during running of lexical analyser, and supply solutions for these problems.

Keywords, standard words All of programming languages allows identifiers having special names and predefined meanings. They are the ***keywords***. Keywords are used only in their original notions. However there are identifiers which also have predefined meaning but they are alterable in the programs. These words are called ***standard words***.

The number of keywords and standard words in programming languages are vary. For example, there is a program language, in which three keywords are used for the zero value: `zero, zeros` és `zeroes.`

Now we investigate how does the lexical analyser recognise keywords and standard words, and how does it distinguish them from identifiers created by the programmers.

The usage of a standard word distinctly from its original meaning renders extra difficulty, not only to the compilation process but also to the readability of the program, such as in the next example:

```
if if then else = then;
```

or if we declare procedures which have names `begin` and `end`:

```
begin
  begin; begin end; end; begin end;
end;
```

Recognition of keywords and standard words is a simple task if they are written using special type characters (for example bold characters), or they are between special prefix and postfix characters (for example between apostrophes).

We give two methods to analyse keywords.

1. All keywords is written as a regular expression, and the implementation of the automaton created to this expression is prepared. The disadvantage of this method is the size of the analyser program. It will be large even if the description of keywords, whose first letter are the same, are contracted.

2. Keywords are stored in a special keyword-table. The words can be determined in the character stream by a general identifier- recogniser. Then, by a simple search algorithm, we check whether this word is in the keyword- table. If this word is in the table then it is a keyword. Otherwise it is an identifier defined by the user. This method is very simple, but the efficiency of search depends on

the structure of keyword-table and on the algorithm of search. A well-selected mapping function and an adequate keyword-table should be very effective.

If it is possible to write standard words in the programming language, then the lexical analyser recognises the standard words using one of the above methods. But the meaning of this standard word depends of its context. To decide, whether it has its original meaning or it was overdefined by the programmer, is the task of syntactic analyser.

Look ahead Since the lexical analyser creates a symbol from the longest character stream, the lexical analyser has to look ahead one or more characters for the allocation of the right-end of a symbol There is a classical example for this problem, the next two FORTRAN statements:

```
DO 10 I = 1.1000
DO 10 I = 1,1000
```

In the FORTRAN programming language space-characters are not important characters, they do not play an important part, hence the character between `1` and `1000` decides that the statement is a `DO` cycle statement or it is an assignment statement for the `DO10I` identifier.

To sign the right end of the symbol, we introduce the symbol / into the description of regular expressions. Its name is ***lookahead operator.*** Using this symbol the description of the above `DO` keyword is the next:

$$\texttt{DO} \; / \; (\mathit{letter} \mid \mathit{digit})^* = (\mathit{letter} \mid \mathit{digit})^* \texttt{,}$$

This definition means that the lexical analyser says that the first two `D` and `O` letters are the `DO` keyword, if looking ahead, after the `O` letter, there are letters or digits, then there is an equal sign, and after this sign there are letters or digits again, and finally, there is a „ , ” character. The lookahead operator implies that the lexical analyser has to look ahead after the `DO` characters. We remark that using this lookahead method the lexical analyser recognises the `DO` keyword even if there is an error in the character stream, such as in the `DO2A=3B,` character stream, but in a correct assignment statement it does not detect the `DO` keyword.

In the next example we concern for positive integers. The definition of integer numbers is a prefix of the definition of the real numbers, and the definition of real numbers is a prefix of the definition of real numbers containing explicit power-part.

pozitív egész :	D^+
pozitív valós :	$D^+.D^+$
	és $D^+.D^+\texttt{e}(+ \mid - \mid \varepsilon)D^+$

The automaton for all of these three expressions is the automaton of the longest character stream, the real number containing explicit power-part.

The problem of the lookahead symbols is resolved using the following algorithm. Put the character into a buffer, and put an auxiliary information aside this character. This information is „it is invalid”. if the character string, using this red character, is

not correct; otherwise we put the type of the symbol into here. If the automaton is in a final-state, then the automaton recognises a real number with explicit power-part. If the automaton is in an internal state, and there is no possibility to read a next character, then the longest character stream which has valid information is the recognised symbol.

Example 2.4 Consider the `12.3e+f#` character stream, where the character `#` is the endsign of the analysed text. If in this character stream there was a positive integer number in the place of character `f`, then this character stream should be a real number. The content of the puffer of lexical analyser:

`1`	*integer number*
`12`	*integer number*
`12.`	*invalid*
`12.3`	*real number*
`12.3e`	*invalid*
`12.3e+`	*invalid*
`12.3e+f`	*invalid*
`12.3e+f#`	

The recognised symbol is the `12.3` *real number*. The lexical analysing is continued at the text `e+f`.

The number of lookahead-characters may be determined from the definition of the program language. In the modern languages this number is at most two.

The symbol table There are programming languages, for example C, in which small letters and capital letters are different. In this case the lexical analyser uses characters of all symbols without modification. Otherwise the lexical analyser converts all characters to their small letter form or all characters to capital letter form. It is proposed to execute this transformation in the source handler program.

At the case of simpler programming languages the lexical analyser writes the characters of the detected symbol into the symbol table, if this symbol is not there. After writing up, or if this symbol has been in the symbol table already, the lexical analyser returns the table address of this symbol, and writes this information into its output. These data will be important at semantic analysis and code generation.

Directives In programming languages the ***directives*** serve to control the compiler. The lexical analyser identifies directives and recognises their operands, and usually there are further tasks with these directives.

If the directive is the `if` of the conditional compilation, then the lexical analyser has to detect all of parameters of this condition, and it has to evaluate the value of the branch. If this value is `false`, then it has to omit the next lines until the `else` or `endif` directive. It means that the lexical analyser performs syntactic and semantic checking, and creates code-style information. This task is more complicate if the programming language gives possibility to write nested conditions.

Other types of directives are the substitution of macros and including files into the source text. These tasks are far away from the original task of the lexical analyser.

The usual way to solve these problems is the following. The compiler executes

a pre-processing program, and this program performs all of the tasks written by directives.

Exercises

2.2-1 Give a regular expression to the comments of a programming language. In this language the delimiters of comments are $/*$ and $*/$, and inside of a comment may occurs / and $*$ characters, but $*/$ is forbidden.

2.2-2 Modify the result of the previous question if it is supposed that the programming language has possibility to write nested comments.

2.2-3 Give a regular expression for positive integer numbers, if the pre- and post-zero characters are prohibited. Give a deterministic finite automaton for this regular expression.

2.2-4 Write a program, which re-creates the original source program from the output of lexical analyser. Pay attention for nice an correct positions of the re-created character streams.

2.3. Syntactic analysis

The perfect definition of a programming language includes the definition of its *syntax* and *semantics*.

The syntax of the programming languages cannot be written by context free grammars. It is possible by using context dependent grammars, two-level grammars or attribute grammars. For these grammars there are not efficient parsing methods, hence the description of a language consists of two parts. The main part of the syntax is given using context free grammars, and for the remaining part a context dependent or an attribute grammar is applied. For example, the description of the program structure or the description of the statement structure belongs to the first part, and the type checking, the scope of variables or the correspondence of formal and actual parameters belong to the second part.

The checking of properties written by context free grammars is called *syntactic analysis* or *parsing*. Properties that cannot be written by context free grammars are called form the *static semantics*. These properties are checked by the *semantic analyser*.

The conventional semantics has the name *run-time semantics* or *dynamic semantics*. The dynamic semantics can be given by verbal methods or some interpreter methods, where the operation of the program is given by the series of state-alterations of the interpreter and its environment.

We deal with *context free grammars*, and in this section we will use *extended grammars* for the syntactic analysis. We investigate on methods of checking of properties which are written by context free grammars. First we give basic notions of the syntactic analysis, then the parsing algorithms will be studied.

Definition 2.1 *Let $G = (N, T, P, S)$ be a grammar. If $S \overset{*}{\Longrightarrow} \alpha$ and $\alpha \in (N \cup T)^*$ then α is a **sentential form**. If $S \overset{*}{\Longrightarrow} x$ and $x \in T^*$ then x is a **sentence** of the language defined by the grammar.*

The sentence has an important role in parsing. The program written by a programmer is a series of terminal symbols, and this series is a sentence if it is correct, that is, it has not syntactic errors.

Definition 2.2 *Let $G = (N, T, P, S)$ be a grammar and $\alpha = \alpha_1\beta\alpha_2$ is a sentential form ($\alpha, \alpha_1, \alpha_2, \beta \in (N \cup T)^*$). We say that β is a* ***phrase*** *of α, if there is a symbol $A \in N$, which $S \overset{*}{\Longrightarrow} \alpha_1 A \alpha_2$ and $A \overset{*}{\Longrightarrow} \beta$. We say that α is a* ***simple phrase*** *of β, if $A \to \beta \in P$.*

We note that every sentence is phrase. The leftmost simple phrase has an important role in parsing; it has its own name.

Definition 2.3 *The leftmost simple phase of a sentence is the* ***handle****.*

The leaves of the *syntax tree* of a sentence are terminal symbols, other points of the tree are nonterminal symbols, and the root symbol of the tree is the start symbol of the grammar.

In an ambiguous grammar there is at least one sentence, which has several syntax trees. It means that this sentence has more than one analysis, and therefore there are several target programs for this sentence. This ambiguity raises a lot of problems, therefore the compilers translate languages generated by unambiguous grammars only.

We suppose that the grammar G has properties as follows:

1. the grammar is *cycle free,* that is, it has not series of derivations rules $A \overset{+}{\Longrightarrow} A$ ($A \in N$),
2. the grammar is *reduced*, that is, there are not „unused symbols" in the grammar, all of nonterminals happen in a derivation, and from all nonterminals we can derive a part of a sentence. This last property means that for all $A \in N$ it is true that $S \overset{*}{\Longrightarrow} \alpha A \beta \overset{*}{\Longrightarrow} \alpha y \beta \overset{*}{\Longrightarrow} xyz$, where $A \overset{*}{\Longrightarrow} y$ and $|y| > 0$ ($\alpha, \beta \in (N \cup T)^*,\ x, y, z \in T^*$).

As it has shown, the lexical analyser translates the program written by a programmer into series of terminal symbols, and this series is the input of syntactic analyser. The task of syntactic analyser is to decide if this series is a sentence of the grammar or it is not. To achieve this goal, the parser creates the syntax tree of the series of symbols. From the known start symbol and the leaves of the syntax tree the parser creates all vertices and edges of the tree, that is, it creates a derivation of the program.

If this is possible, then we say that the program is an element of the language. It means that the program is syntactically correct.

Hence forward we will deal with *left to right* parsing methods. These methods read the symbols of the programs left to right. All of the real compilers use this method.

To create the inner part of the syntax tree there are several methods. One of these methods builds the syntax tree from its start symbol S. This method is called *top-down* method. If the parser goes from the leaves to the symbol S, then it uses

the *bottom-up* parsing method.

We deal with top-down parsing methods in Subsection 2.3.1. We investigate bottom-up parsers in Subsection 2.3.2; now these methods are used in real compilers.

2.3.1. *LL*(1) parser

If we analyse from top to down then we start with the start symbol. This symbol is the root of syntax tree; we attempt to construct the syntax tree. Our goal is that the leaves of tree are the terminal symbols.

First we review the notions that are necessary in the top-down parsing. Then the $LL(1)$ table methods and the recursive descent method will be analysed.

***LL(k)* grammars** Our methods build the syntax tree top-down and read symbols of the program left to right. For this end we try to create terminals on the left side of sentential forms.

Definition 2.4 *If $A \to \alpha \in P$ then the **leftmost direct derivation** of the sentential form $xA\beta$ ($x \in T^*,\ \alpha, \beta \in (N \cup T)^*$) is $x\alpha\beta$, and*

$$xA\beta \underset{leftmost}{\Longrightarrow} x\alpha\beta\ .$$

Definition 2.5 *If all of direct derivations in $S \stackrel{*}{\Longrightarrow} x$ ($x \in T^*$) are leftmost, then this derivation is said to be **leftmost derivation**, and*

$$S \underset{leftmost}{\overset{*}{\Longrightarrow}} x\ .$$

In a leftmost derivation terminal symbols appear at the left side of the sentential forms. Therefore we use leftmost derivations in all of top-down parsing methods. Hence if we deal with top-down methods, we do not write the text "leftmost" at the arrows.

One might as well say that we create all possible syntax trees. Reading leaves from left to right, we take sentences of the language. Then we compare these sentences with the parseable text and if a sentence is same as the parseable text, then we can read the steps of parsing from the syntax tree which is belongs to this sentence. But this method is not practical; generally it is even impossible to apply.

A good idea is the following. We start at the start symbol of the grammar, and using leftmost derivations we try to create the text of the program. If we use a not suitable derivation at one of steps of parsing, then we find that, at the next step, we can not apply a proper derivation. At this case such terminal symbols are at the left side of the sentential form, that are not same as in our parseable text.

For the leftmost terminal symbols we state the theorem as follows.

Theorem 2.6 *If $S \stackrel{*}{\Longrightarrow} x\alpha \stackrel{*}{\Longrightarrow} yz$ ($\alpha \in (N \cup T)^*,\ x, y, z \in T^*$) és $|x| = |y|$, then $x = y$.*

The proof of this theorem is trivial. It is not possible to change the leftmost terminal symbols x of sentential forms using derivation rules of a context free grammar.

This theorem is used during the building of syntax tree, to check that the leftmost terminals of the tree are same as the leftmost symbols of the parseable text. If they are different then we created wrong directions with this syntax tree. At this case we have to make a backtrack, and we have to apply an other derivation rule. If it is impossible (since for example there are no more derivation rules) then we have to apply a backtrack once again.

General top-down methods are realized by using backtrack algorithms, but these backtrack steps make the parser very slow. Therefore we will deal only with grammars such that have parsing methods without backtracks.

The main properties of $LL(k)$ grammars are the following. If, by creating the leftmost derivation $S \overset{*}{\Longrightarrow} wx$ $(w, x \in T^*)$, we obtain the sentential form $S \overset{*}{\Longrightarrow} wA\beta$ $(A \in N,\ \beta \in (N \cup T)^*)$ at some step of this derivation, and our goal is to achieve $A\beta \overset{*}{\Longrightarrow} x$, then the next step of the derivation for nonterminal A is determinable unambiguously from the first k symbols of x.

To look ahead k symbols we define the function $First_k$.

Definition 2.7 *Let* $\boldsymbol{First_k(\alpha)}$ $(k \geq 0,\ \alpha \in (N \cup T)^*)$ *be the set as follows.*

$$First_k(\alpha) = \{x \mid \alpha \overset{*}{\Longrightarrow} x\beta \text{ and } |x| = k\} \cup \{x \mid \alpha \overset{*}{\Longrightarrow} x \text{ and } |x| < k\} \ \ (x \in T^*,\ \beta \in (N \cup T)^*)\ .$$

The set $First_k(x)$ consists of the first k symbols of x; for $|x| < k$, it consists the full x. If $\alpha \overset{*}{\Longrightarrow} \varepsilon$, then $\varepsilon \in First_k(\alpha)$.

Definition 2.8 *The grammar G is a* $\boldsymbol{LL(k)}$ ***grammar*** $(k \geq 0)$*, if for derivations*

$$S \overset{*}{\Longrightarrow} wA\beta \Longrightarrow w\alpha_1\beta \overset{*}{\Longrightarrow} wx\ ,$$
$$S \overset{*}{\Longrightarrow} wA\beta \Longrightarrow w\alpha_2\beta \overset{*}{\Longrightarrow} wy$$

$(A \in N,\ x, y, w \in T^*,\ \alpha_1, \alpha_2, \beta \in (N \cup T)^*)$ the equality

$$First_k(x) = First_k(y)$$

implies

$$\alpha_1 = \alpha_2\ .$$

Using this definition, if a grammar is a $LL(k)$ grammar then the k symbol after the parsed x determine the next derivation rule unambiguously (Figure 2.8).

One can see from this definition that if a grammar is an $LL(k_0)$ grammar then for all $k > k_0$ it is also an $LL(k)$ grammar. If we speak about $LL(k)$ grammar then we also mean that k is the least number such that the properties of the definition are true.

Example 2.5 The next grammar is a $LL(1)$ grammar. Let $G = (\{A, S\}, \{a, b\}, P, S)$ be a grammar whose derivation rules are:

$S \to AS \mid \varepsilon$

$A \to aA \mid b$

We have to use the derivation $S \to AS$ for the start symbol S if the next symbol of the parseable text is a or b. We use the derivation $S \to \varepsilon$ if the next symbol is the mark $\#$.

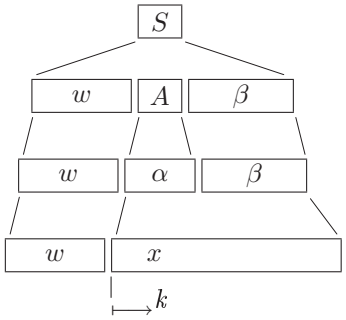


Figure 2.8. $LL(k)$ grammar.

Example 2.6 The next grammar is a $LL(2)$ grammar. Let $G = (\{A, S\}, \{a, b\}, P, S)$ be a grammar whose the derivation rules are:

$S \to abA \mid \varepsilon$

$A \to Saa \mid b$

One can see that at the last step of derivations

$$S \Longrightarrow abA \Longrightarrow abSaa \overset{S \to abA}{\Longrightarrow} ababAaa$$

and

$$S \Longrightarrow abA \Longrightarrow abSaa \overset{S \to \varepsilon}{\Longrightarrow} abaa$$

if we look ahead one symbol, then in both derivations we obtain the symbol a. The proper rule for symbol S is determined to look ahead two symbols (ab or aa).

There are context free grammars such that are not $LL(k)$ grammars. For example the next grammar is not $LL(k)$ grammar for any k.

Example 2.7 Let $G = (\{A, B, S\}, \{a, b, c\}, P, S)$ be a grammar whose the derivation rules are:

$S \to A \mid B$

$A \to aAb \mid ab$

$B \to aBc \mid ac$

$L(G)$ consists of sentences $a^i b^i$ és $a^i c^i$ $(i \geq 1)$. If we analyse the sentence $a^{k+1}b^{k+1}$, then at the first step we can not decide by looking ahead k symbols whether we have to use the derivation $S \to A$ or $S \to B$, since for all k $First_k(a^k b^k) = First_k(a^k c^k) = a^k$.

By the definition of the $LL(k)$ grammar, if we get the sentential form $wA\beta$ using leftmost derivations, then the next k symbol determines the next rule for symbol A. This is stated in the next theorem.

Theorem 2.9 *Grammar G is a $LL(k)$ grammar iff*

$$S \overset{*}{\Longrightarrow} wA\beta, \text{ és } A \to \gamma \mid \delta \quad (\gamma \neq \delta,\ w \in T^*,\ A \in N,\ \beta, \gamma, \delta \in (N \cup T)^*)$$

implies

$$First_k(\gamma\beta) \cap First_k(\delta\beta) = \emptyset\ .$$

If there is a $A \to \varepsilon$ rule in the grammar, then the set $First_k$ consists the k length prefixes of terminal series generated from β. It implies that, for deciding the property $LL(k)$, we have to check not only the derivation rules, but also the infinite derivations.

We can give good methods, that are used in the practice, for $LL(1)$ grammars only. We define the follower-series, which follow a symbol or series of symbols.

Definition 2.10 $\boldsymbol{Follow_k(\beta)} = \{x \mid S \overset{*}{\Longrightarrow} \alpha\beta\gamma \text{ and } x \in First_k(\gamma)\}$, *and if* $\varepsilon \in Follow_k(\beta)$, *then* $Follow_k(\beta) = Follow_k(\beta) \setminus \{\varepsilon\} \cup \{\#\}$ $(\alpha, \beta, \gamma \in (N \cup T)^*,\ x \in T^*)$.

The second part of the definition is necessary because if there are no symbols after the β in the derivation $\alpha\beta\gamma$, that is $\gamma = \varepsilon$, then the next symbol after β is the mark $\#$ only.

$Follow_1(A)$ $(A \in N)$ consists of terminal symbols that can be immediately after the symbol A in the derivation

$$S \overset{*}{\Longrightarrow} \alpha A\gamma \overset{*}{\Longrightarrow} \alpha Aw \ (\alpha, \gamma \in (N \cup T)^*,\ w \in T^*).$$

Theorem 2.11 *The grammar G is a $LL(1)$ grammar iff, for all nonterminal A and for all derivation rules $A \to \gamma \mid \delta$,*

$$First_1(\gamma Follow_1(A)) \cap First_1(\delta Follow_1(A)) = \emptyset \ .$$

In this theorem the expression $First_1(\gamma Follow_1(A))$ means that we have to concatenate to γ the elements of set $Follow_1(A)$ separately, and for all elements of this new set we have to apply the function $First_1$.

It is evident that Theorem 2.11 is suitable to decide whether a grammar is $LL(1)$ or it is not.

Hence forward we deal with $LL(1)$ languages determined by $LL(1)$ grammars, and we investigate the parsing methods of $LL(1)$ languages. For the sake of simplicity, we omit indexes from the names of functions $First_1$ és $Follow_1$.

The elements of the set $First(\alpha)$ are determined using the next algorithm.

FIRST(α)

```
1  if α = ε
2     then F ← {ε}
3  if α = a, where a ∈ T
4     then F ← {a}
```

```
 5  if α = A, where A ∈ N
 6     then if A → ε ∈ P
 7             then F ← {ε}
 8             else  F ← ∅
 9          for all A → Y1Y2...Ym ∈ P  (m ≥ 1)
10             do F ← F ∪ (First(Y1) \ {ε})
11                for k ← 1 to m − 1
12                   do if Y1Y2...Yk ⟹* ε
13                         then F ← F ∪ (First(Yk+1) \ {ε})
14                if Y1Y2...Ym ⟹* ε
15                   then F ← F ∪ {ε}
16  if α = Y1Y2...Ym (m ≥ 2)
17     then F ← (First(Y1) \ {ε})
18          for k ← 1 to m − 1
19             do if Y1Y2...Yk ⟹* ε
20                   then F ← F ∪ (First(Yk+1) \ {ε})
21          if Y1Y2...Ym ⟹* ε
22             then F ← F ∪ {ε}
23  return F
```

In lines 1–4 the set is given for ε and a terminal symbol a. In lines 5–15 we construct the elements of this set for a nonterminal A. If ε is derivated from A then we put symbol ε into the set in lines 6–7 and 14–15. If the argument is a symbol stream then the elements of the set are constructed in lines 16–22. We notice that we can terminate the **for** cycle in lines 11 and 18 if $Y_k \in T$, since in this case it is not possible to derive symbol ε from $Y_1Y_2\ldots Y_k$.

In Theorem 2.11 and hereafter, it is necessary to know the elements of the set *Follow*(A). The next algorithm constructs this set.

Follow(A)

```
 1  if A = S
 2     then F ← {#}
 3     else  F ← ∅
 4  for all rules B → αAβ ∈ P
 5     do if |β| > 0
 6           then F ← F ∪ (First(β) \ {ε})
 7                if β ⟹* ε
 8                   then F ← F ∪ Follow(B)
 9           else  F ← F ∪ Follow(B)
10  return F
```

The elements of the *Follow*(A) set get into the set F. In lines 4–9 we check that, if the argumentum is at the right side of a derivation rule, what symbols may stand immediately after him. It is obvious that no ε is in this set, and the symbol # is in

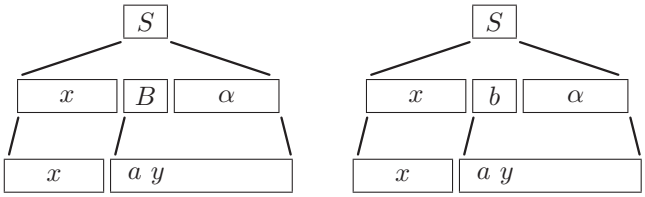


Figure 2.9. The sentential form and the analysed text.

the set only if the argumentum is the rightmost symbol of a sentential form.

Parsing with table Suppose that we analyse a series of terminal symbols xay and the part x has already been analysed without errors. We analyse the text with a top-down method, so we use leftmost derivations. Suppose that our sentential form is $xY\alpha$, that is, it has form $xB\alpha$ or $xb\alpha$ ($Y \in (N \cup T)$, $B \in N$, $a, b \in T$, $x, y \in T^*$, $\alpha \in (N \cup T)^*$) (Figure 2.9).

In the first case the next step is the substitution of symbol B. We know the next element of the input series, this is the terminal a, therefore we can determine the correct substitution of symbol B. This substitution is the rule $B \rightarrow \beta$ for which $a \in \mathit{First}(\beta \mathit{Follow}(B))$. If there is such a rule then, according to the definition of $LL(1)$ grammar, there is exactly one. If there is not such a rule, then a *syntactic error* was found.

In the second case the next symbol of the sentential form is the terminal symbol b, thus we look out for the symbol b as the next symbol of the analysed text. If this comes true, that is, $a = b$, then the symbol a is a correct symbol and we can go further. We put the symbol a into the already analysed text. If $a \neq b$, then here is a *syntactic error*. We can see that the position of the error is known, and the erroneous symbol is the terminal symbol a.

The action of the parser is the following. Let $\#$ be the sign of the right end of the analysed text, that is, the mark $\#$ is the last symbol of the text. We use a stack through the analysing, the bottom of the stack is signed by mark $\#$, too. We give serial numbers to derivation rules and through the analysing we write the number of the applied rule into a list. At the end of parsing we can write the syntax tree from this list (Figure 2.10).

We sign the *state of the parser* using triples $(ay\#, X\alpha\#, v)$. The symbol $ay\#$ is the text not analysed yet. $X\alpha\#$ is the part of the sentential form corresponding to the not analysed text; this information is in the stack, the symbol X is at the top of the stack. v is the list of the serial numbers of production rules.

If we analyse the text then we observe the symbol X at the top of the stack, and the symbol a that is the first symbol of the not analysed text. The name of the symbol a is *actual symbol*. There are pointers to the top of the stack and to the actual symbol.

We use a top down parser, therefore the initial content of the stack is $S\#$. If the initial analysed text is xay, then the initial state of the parsing process is the triple

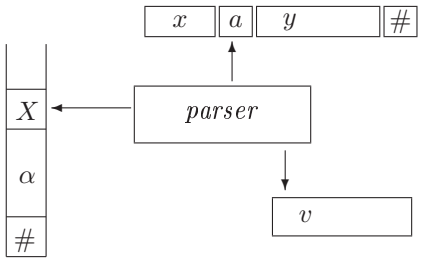


Figure 2.10. The structure of the $LL(1)$ parser.

$(xay\#, S\#, \varepsilon)$, where ε is the sign of the empty list.

We analyse the text, the series of symbols using a *parsing table* The rows of this table sign the symbols at the top of the stack, the columns of the table sign the next input symbols, and we write mark # to the last row and the last column of the table. Hence the number of rows of the table is greater by one than the number of symbols of the grammar, and the number of columns is greater by one than the number of terminal symbols of the grammar.

The element $T[X, a]$ of the table is as follows.

$$T[X,a] = \begin{cases} (\beta, i), & \text{ha } X \to \beta \text{ az } i\text{-th derivation rule ,} \\ & a \in \mathit{First}(\beta) \text{ or} \\ & (\varepsilon \in \mathit{First}(\beta) \text{ and } a \in \mathit{Follow}(X)) \text{ ,} \\ \mathit{pop}, & \text{if } X = a \text{ ,} \\ \mathit{accept}, & \text{if } X = \# \text{ and } a = \# \text{ ,} \\ \mathit{error} & \text{otherwise .} \end{cases}$$

We fill in the parsing table using the following algorithm.

LL(1)-Table-Fill-in(G)

```
 1  for all A ∈ N
 2      do if A → α ∈ P the i-th rule
 3            then for all a ∈ First(α)- ra
 4                    do T[A, a] ← (α, i)
 5                    if ε ∈ First(α)
 6                       then for all a ∈ Follow(A)
 7                               do T[A, a] ← (α, i)
 8  for all a ∈ T
 9      do T[a, a] ← pop
10  T[#, #] ← accept
11  for all X ∈ (N ∪ T ∪ {#}) and all a ∈ (T ∪ {#})
12      do if T[X, a] = „empty"
13            then T[X, a] ← error
14  return T
```

At the line 10 we write the text *accept* into the right lower corner of the table. At

the lines 8–9 we write the text *pop* into the main diagonal of the square labelled by terminal symbols. The program in lines 1–7 writes a tuple in which the first element is the right part of a derivation rule and the second element is the serial number of this rule. In lines 12–13 we write *error* texts into the empty positions.

The actions of the parser are written by state-transitions. The initial state is $(x\#, S\#, \varepsilon)$, where the initial text is x, and the parsing process will be finished if the parser goes into the state $(\#, \#, w)$, this state is the *final state* If the text is $ay\#$ in an intermediate step, and the symbol X is at the top of stack, then the possible state-transitions are as follows.

$$(ay\#, X\alpha\#, v) \;\rightarrow\; \begin{cases} (ay\#, \beta\alpha\#, vi), & \text{ha } T[X,a] = (\beta, i)\ , \\ (y\#, \alpha\#, v), & \text{ha } T[X,a] = \mathit{pop}\ , \\ \mathit{O.K.}, & \text{ha } T[X,a] = \mathit{accept}\ , \\ \mathit{ERROR}, & \text{ha } T[X,a] = \mathit{error}\ . \end{cases}$$

The letters *O.K.* mean that the analysed text is syntactically correct; the text *ERROR* means that a syntactic error is detected.

The actions of this parser are written by the next algorithm.

LL(1)-Parser$(xay\#, T)$

```
 1  s ← (xay#, S#, ε), s' ← analyze
 2  repeat
 3        if s = (ay#, Aα#, v) és T[A, a] = (β, i)
 4           then s ← (ay#, βα#, vi)
 5           else if s = (ay#, aα#, v)
 6                    then s ← (y#, α#, v)            ▷ Then T[a, a] = pop.
 7                    else  if s = (#, #, v)
 8                             then s' ← O.K.         ▷ Then T[#, #] = accept.
 9                             else  s' ← ERROR       ▷ Then T[A, a] = error.
10  until s' = O.K. or s' = ERROR
11  return s', s
```

The input parameters of this algorithm are the text xay and the parsing table T. The variable s' describes the state of the parser: its value is *analyse,* during the analysis, and it is either *O.K.* or *ERROR.* at the end. The parser determines his action by the actual symbol a and by the symbol at the top of the stack, using the parsing table T. In the line 3–4 the parser builds the syntax tree using the derivation rule $A \to \beta$. In lines 5–6 the parser executes a shift action, since there is a symbol a at the top of the stack. At lines 8–9 the algorithm finishes his work if the stack is empty and it is at the end of the text, otherwise a syntactic error was detected. At the end of this work the result is *O.K.* or *ERROR* in the variable s', and, as a result, there is the triple s at the output of this algorithm. If the text was correct, then we can create the syntax tree of the analysed text from the third element of the triple. If there was an error, then the first element of the triple points to the position of the erroneous symbol.

Example 2.8 Let G be a grammar $G = (\{E, E', T, T', F\}, \{+, *, (,), i\}, P, E)$, where the

set P of derivation rules:

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid i$

>From these rules we can determine the $Follow(A)$ sets. To fill in the parsing table, the following sets are required:

$First(TE') = \{(, i\}$,
$First(+TE') = \{+\}$,
$First(FT') = \{(, i\}$,
$First(*FT') = \{*\}$,
$First((E)) = \{(\}$,
$First(i) = \{i\}$,
$Follow(E') = \{), \#\}$,
$Follow(T') = \{+,), \#\}$.

The parsing table is as follows. The empty positions in the table mean *errors*

	+	*	(	)	i	#
E			$(TE', 1)$		$(TE', 1)$	
E'	$(+TE', 2)$			$(\varepsilon, 3)$		$(\varepsilon, 3)$
T			$(FT', 4)$		$(FT', 4)$	
T'	$(\varepsilon, 6)$	$(*FT', 5)$		$(\varepsilon, 6)$		$(\varepsilon, 6)$
F			$((E), 7)$		$(i, 8)$	
+	*pop*					
*		*pop*				
(			*pop*			
)				*pop*		
i					*pop*	
#						*accept*

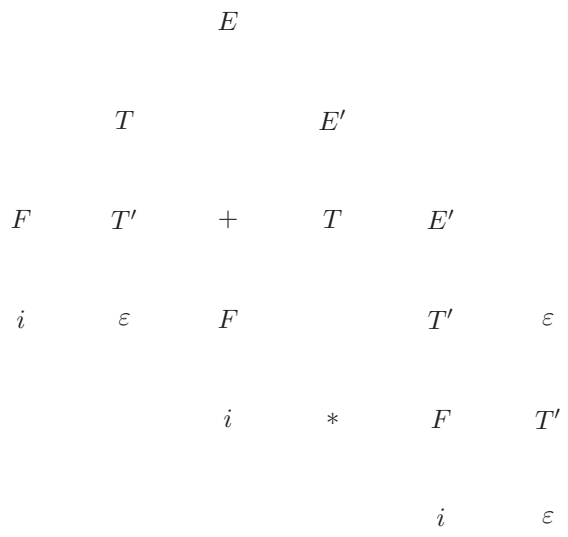


Figure 2.11. The syntax tree of the sentence $i+i*i$.

Example 2.9 Using the parsing table of the previous example, analyse the text $i+i*i$.

$(i+i*i\#,\ S\#,\ \varepsilon)$	$\xrightarrow{(TE',1)}$	(	$i+i*i\#,$	$TE'\#,$	1	)
	$\xrightarrow{(FT',4)}$	(	$i+i*i\#,$	$FT'E'\#,$	14	)
	$\xrightarrow{(i,8)}$	(	$i+i*i\#,$	$iT'E'\#,$	148	)
	$\xrightarrow{pop}$	(	$+i*i\#,$	$T'E'\#,$	148	)
	$\xrightarrow{(\varepsilon,6)}$	(	$+i*i\#,$	$E'\#,$	1486	)
	$\xrightarrow{(+TE',2)}$	(	$+i*i\#,$	$+TE'\#,$	14862	)
	$\xrightarrow{pop}$	(	$i*i\#,$	$TE'\#,$	14862	)
	$\xrightarrow{(FT',4)}$	(	$i*i\#,$	$FT'E'\#,$	148624	)
	$\xrightarrow{(i,8)}$	(	$i*i\#,$	$iT'E'\#,$	1486248	)
	$\xrightarrow{pop}$	(	$*i\#,$	$T'E'\#,$	1486248	)
	$\xrightarrow{(*FT',5)}$	(	$*i\#,$	$*FT'E'\#,$	14862485	)
	$\xrightarrow{pop}$	(	$i\#,$	$FT'E'\#,$	14862485	)
	$\xrightarrow{(i,8)}$	(	$i\#,$	$iT'E'\#,$	148624858	)
	$\xrightarrow{pop}$	(	$\#,$	$T'E'\#,$	148624858	)
	$\xrightarrow{(\varepsilon,6)}$	(	$\#,$	$E'\#,$	1486248586	)
	$\xrightarrow{(\varepsilon,3)}$	(	$\#,$	$\#,$	14862485863	)
	$\xrightarrow{accept}$		$O.K.$			

The syntax tree of the analysed text is the Figure 2.11.

Recursive-descent parsing method There is another frequently used method for the backtrackless top-down parsing. Its essence is that we write a real program

for the applied grammar. We create procedures to the symbols of grammar, and using these procedures the recursive procedure calls realize the stack of the parser and the stack management. This is a top-down parsing method, and the procedures call each other recursively; it is the origin of the name of this method, that is, ***recursive-descent method.***

To check the terminal symbols we create the procedure *Check*. Let the parameter of this procedure be the „expected symbol", that is the leftmost unchecked terminal symbol of the sentential form, and let the *actual symbol* be the symbol which is analysed in that moment.

```
procedure Check(a);
begin
  if actual_symbol = a
      then Next_symbol
      else Error_report
end;
```

The procedure *Next_symbol* reads the next symbol, it is a call for the lexical analyser. This procedure determines the next symbol and put this symbol into the *actual_symbol* variable. The procedure *Error_report* creates an error report and then finishes the parsing.

We create procedures to symbols of the grammar as follows. The procedure of the nonterminal symbol A is the next.

```
procedure  A;
begin
  T(A)
end;
```

where `T(A)` is determined by symbols of the right part of derivation rule having symbol A in its left part.

The grammars which are used for syntactic analysis are *reduced grammars*. It means that no unnecessary symbols in the grammar, and all of symbols occur at the left side at least one reduction rule. Therefore, if we consider the symbol A, there is at least one $A \to \alpha$ production rule.

1. If there is only one production rule for the symbol A,
   (a) let the program of the rule $A \to a$ is as follows: `Check(a)`,
   (b) for the rule $A \to B$ we give the procedure call `B` ,
   (c) for the rule $A \to X_1 X_2 \dots X_n$ $(n \geq 2)$ we give the next block:

```
begin
  T(X_1);
  T(X_2);
  ...
  T(X_n)
end;
```

2. If there are more rules for the symbol A:

   (a) If the rules $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ are ε-free, that is from α_i $(1 \leq i \leq n)$ it is not possible to deduce ε, then `T(A)`

   ```
   case actual_symbol of
     First(alpha_1) : T(alpha_1);
     First(alpha_2) : T(alpha_2);
     ...
     First(alpha_n) : T(alpha_n)
   end;
   ```

   where `First(alpha_i)` is the sign of the set $\mathit{First}(\alpha_i)$.

   We note that this is the first point of the method of recursive-descent parser where we use the fact that the grammar is an $LL(1)$ grammar.

   (b) We use the $LL(1)$ grammar to write a programming language, therefore it is not comfortable to require that the grammar is a ε-free grammar. For the rules $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_{n-1} \mid \varepsilon$ we create the next `T(A)` program:

   ```
   case actual_symbol of
     First(alpha_1)     : T(alpha_1);
     First(alpha_2)     : T(alpha_2);
     ...
     First(alpha_(n-1)) : T(alpha_(n-1));
     Follow(A)          : skip
   end;
   ```

   where `Follow(A)` is the set $\mathit{Follow}(A)$.

   In particular, if the rules $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ for some i $(1 \leq i \leq n)$ $\alpha_i \overset{*}{\Longrightarrow} \varepsilon$, that is $\varepsilon \in \mathit{First}(\alpha_i)$, then the i-th row of the `case` statement is `Follow(A) : skip`

In the program `T(A)`, if it is possible, we use `if-then-else` or `while` statement instead of the statement `case`.

The start procedure, that is the main program of this parsing program, is the procedure which is created for the start symbol of the grammar.

We can create the recursive-descent parsing program with the next algorithm. The input of this algorithm is the grammar G, and the result is parsing program P. In this algorithm we use a WRITE-PROGRAM procedure, which concatenates the new program lines to the program P. We will not go into the details of this algorithm.

CREATE-REC-DESC(G)

```
 1  P ← ∅
 2  WRITE-PROGRAM(
 3       procedure Check(a);
 4       begin
 5       if actual_symbol = a
 6          then Next_symbol
 7          else Error_report
 8       end;
 9       )
10  for all symbol A ∈ N of the grammar G
11       do if A = S
12            then WRITE-PROGRAM(
13                      program S;
14                      begin
15                        REC-DESC-STAT(S, P)
16                      end.
17                      )
18            else WRITE-PROGRAM(
19                     procedure A;
20                     begin
21                       REC-DESC-STAT(A, P)
22                     end;
23                     )
24  return P
```

The algorithm creates the *Check* procedure in lines 2–9/ Then, for all nonterminals of grammar G, it determines their procedures using the algorithm REC-DESC-STAT. In the lines 11–17, we can see that for the start symbol S we create the main program. The output of the algorithm is the parsing program.

REC-DESC-STAT(A, P)

```
1  if there is only one rule A → α
2     then REC-DESC-STAT1(α, P)                          ▷ A → α.
3     else REC-DESC-STAT2(A, (α_1, ..., α_n), P)         ▷ A → α_1 | ··· | α_n.
4  return P
```

The form of the statements of the parsing program depends on the derivation rules of the symbol A. Therefore the algorithm REC-DESC-STAT divides the next tasks into two parts. The algorithm REC-DESC-STAT1 deals with the case when there is only one derivation rule, and the algorithm REC-DESC-STAT2 creates the program for the alternatives.

REC-DESC-STAT1(α, P)

```
1  if α = a
2    then WRITE-PROGRAM(
3               Check(a)
4               )
5  if α = B
6    then WRITE-PROGRAM(
7               B
8               )
9  if α = X_1 X_2 ... X_n (n ≥ 2)
10   then WRITE-PROGRAM(
11              begin
12                REC-DESC-STAT1(X_1, P) ;
13                REC-DESC-STAT1(X_2, P) ;
14                ...
15                REC-DESC-STAT1(X_n, P)
16              end;
17 return P
```

REC-DESC-STAT2$(A, (\alpha_1, \ldots, \alpha_n), P)$

```
1  if the rules α_1, ..., α_n are ε- free
2    then WRITE-PROGRAM(
3                case actual_symbol of
4                  First(alpha_1) : REC-DESC-STAT1 (α_1, P) ;
5                  ...
6                  First(alpha_n) : REC-DESC-STAT1 (α_n, P)
7                end;
8                )
9  if there is a ε-rule, α_i = ε (1 ≤ i ≤ n)
10   then WRITE-PROGRAM(
11               case actual_symbol of
12                 First(alpha_1)     : REC-DESC-STAT1 (α_1, P) ;
13                 ...
14                 First(alpha_(i-1)) : REC-DESC-STAT1 (α_{i-1}, P) ;
15                 Follow(A)          : skip;
16                 First(alpha_(i+1)) : REC-DESC-STAT1 (α_{i+1}, P) ;
17                 ...
18                 First(alpha_n)     : REC-DESC-STAT1 (α_1, P)
19               end;
20               )
21 return P
```

These two algorithms create the program described above.

Checking the end of the parsed text is achieved by the recursive- descent parsing method with the next modification. We generate a new derivation rule for the end

mark #. If the start symbol of the grammar is S, then we create the new rule $S' \to S\#$, where the new symbol S' is the start symbol of our new grammar. The mark # is considered as terminal symbol. Then we generate the parsing program for this new grammar.

Example 2.10 We augment the grammar of the Example 2.8 in the above manner. The production rules are as follows.

$S' \to E\#$
$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid i$

In the example 2.8 we give the necessary *First* and *Follow* sets. We use the next sets:
$First(+TE') = \{+\}$,
$First(*FT') = \{*\}$,
$First((E)) = \{(\}$,
$First(i) = \{i\}$,
$Follow(E') = \{), \#\}$,
$Follow(T') = \{+,), \#\}$.

In the comments of the program lines we give the using of these sets. The first characters of the comment are the character pair --.

The program of the recursive-descent parser is the following.

```
program S';
begin
  E;
  Check(#)
end.
procedure E;
begin
  T;
  E'
end;
procedure E';
begin
  case actual_symbol of
  +     : begin                 -- First(+TE')
              Check(+);
              T;
              E'
          end;
  ),#   : skip                  -- Follow(E')
  end
end;
procedure T;
begin
  F;
  T'
end;
```

```
procedure T';
begin
  case actual_symbol of
  *     : begin                -- First(*FT')
              Check(*);
              F;
              T'
          end;
  +,),# : skip                 -- Follow(T')
  end
end;
procedure F;
begin
  case actual_symbol of
  (     : begin                -- First((E))
              Check((;
              E;
              Check())
          end;
  i     : Check(i)             -- First(i)
  end
end;
```

We can see that the main program of this parser belongs to the symbol S'.

2.3.2. *LR*(1) parsing

If we analyse from bottom to up, then we start with the program text. We search the handle of the sentential form, and we substitute the nonterminal symbol that belongs to the handle, for this handle. After this first step, we repeat this procedure several times. Our goal is to achieve the start symbol of the grammar. This symbol will be the root of the syntax tree, and by this time the terminal symbols of the program text are the leaves of the tree.

First we review the notions which are necessary in the parsing.

To analyse bottom-up, we have to determine the handle of the sentential form. The problem is to create a good method which finds the handle, and to find the best substitution if there are more than one possibilities.

Definition 2.12 *If* $A \to \alpha \in P$, *then the* ***rightmost substitution*** *of the sentential form* βAx $(x \in T^*,\ \alpha, \beta \in (N \cup T)^*)$ *is* $\beta\alpha x$, *that is*

$$\beta Ax \underset{rightmost}{\Longrightarrow} \beta\alpha x\ .$$

Definition 2.13 *If the derivation* $S \overset{*}{\Longrightarrow} x$ $(x \in T^*)$ *all of the substitutions were rightmost substitution, then this derivation is a* ***rightmost derivation****,*

$$S \underset{rightmost}{\overset{*}{\Longrightarrow}} x\ .$$

In a rightmost derivation, terminal symbols are at the right side of the sentential form. By the connection of the notion of the handle and the rightmost derivation, if we apply the steps of a rightmost derivation backwards, then we obtain the steps of a bottom-up parsing. Hence the bottom-up parsing is equivalent with the „inverse" of a rightmost derivation. Therefore, if we deal with bottom-up methods, we will not write the text "rightmost" at the arrows.

General bottom-up parsing methods are realized by using backtrack algorithms. They are similar to the top-down parsing methods. But the backtrack steps make the parser very slow. Therefore we only deal with grammars such that have parsing methods without backtracks.

Hence forward we produce a very efficient algorithm for a large class of context-free grammars. This class contains the grammars for the programming languages.

The parsing is called $LR(k)$ parsing; the grammar is called $LR(k)$ grammar. LR means the "Left to Right" method, and k means that if we look ahead k symbols then we can determine the handles of the sentential forms. The $LR(k)$ parsing method is a shift-reduce method.

We deal with $LR(1)$ parsing only, since for all $LR(k)$ $(k > 1)$ grammar there is an equivalent $LR(1)$ grammar. This fact is very important for us since, using this type of grammars, it is enough to look ahead one symbol in all cases.

Creating $LR(k)$ parsers is not an easy task. However, there are such standard programs (for example the `yacc` in UNIX systems), that create the complete parsing program from the derivation rules of a grammar. Using these programs the task of writing parsers is not too hard.

After studying the $LR(k)$ grammars we will deal with the $LALR(1)$ parsing method. This method is used in the compilers of modern programming languages.

$LR(k)$ grammars As we did previously, we write a mark # to the right end of the text to be analysed. We introduce a new nonterminal symbol S' and a new rule $S' \to S$ into the grammar.

Definition 2.14 *Let G' be the **augmented grammar** belongs to grammar $G = (N, T, P, S)$, where G' **augmented grammar***

$$G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S') \ .$$

Assign serial numbers to the derivation rules of grammar, and let $S' \to S$ be the 0th rule. Using this numbering, if we apply the 0th rule, it means that the parsing process is concluded and the text is correct.

We notice that if the original start symbol S does not happen on the right side of any rules, then there is no need for this augmentation. However, for the sake of generality, we deal with augmented grammars only.

Definition 2.15 *The augmented grammar G' is an **$LR(k)$ grammar** $(k \geq 0)$, if for derivations*

$$S' \overset{*}{\Longrightarrow} \alpha A w \Longrightarrow \alpha\beta w \ ,$$
$$S' \overset{*}{\Longrightarrow} \gamma B x \Longrightarrow \gamma\delta x = \alpha\beta y$$

$(A, B \in N,\ x, y, w \in T^*,\ \alpha, \beta, \gamma, \delta \in (N \cup T)^*)$ the equality

$$First_k(w) = First_k(y)$$

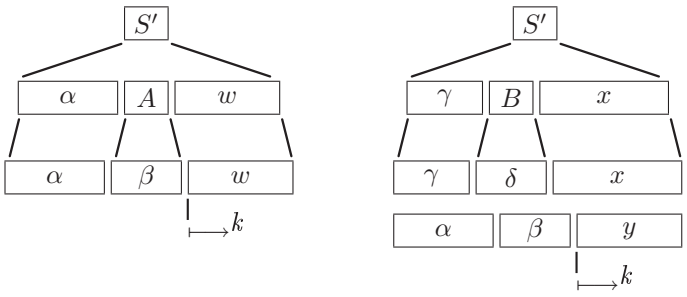


Figure 2.12. The $LR(k)$ grammar.

implies

$$\alpha = \gamma, A = B \text{ és } x = y \ .$$

The feature of $LR(k)$ grammars is that, in the sentential form $\alpha\beta w$, looking ahead k symbol from w unambiguously decides if β is or is not the handle. If the handle is *beta*, then we have to reduce the form using the rule $A \to \beta$, that results the new sentential form is αAw. Its reason is the following: suppose that, for sentential forms $\alpha\beta w$ and $\alpha\beta y$, (their prefixes $\alpha\beta$ are same), $First_k(w) = First_k(y)$, and we can reduce $\alpha\beta w$ to αAw and $\alpha\beta y$ to γBx. In this case, since the grammar is a $LR(k)$ grammar, $\alpha = \gamma$ and $A = B$ hold. Therefore in this case either the handle is β or β never is the handle.

Example 2.11 Let $G' = (\{S', S\}, \{a\}, P', S')$ be a grammar and let the derivation rules be as follows.

$S' \to S$

$S \to Sa \mid a$

This grammar is not an $LR(0)$ grammar, since using notations of the definition, in the derivations

$$\begin{array}{lcccccccccc} S' & \overset{*}{\Longrightarrow} & \varepsilon & S' & \varepsilon & \Longrightarrow & \varepsilon & S & \varepsilon, & & \\ & & \alpha & A & w & & \alpha & \beta & w & & \\ S' & \overset{*}{\Longrightarrow} & \varepsilon & S' & \varepsilon & \Longrightarrow & \varepsilon & Sa & \varepsilon & = & \varepsilon \quad S \quad a, \\ & & \gamma & B & x & & \gamma & \delta & x & & \alpha \quad \beta \quad y \end{array}$$

it holds that $First_0(\varepsilon) = First_0(a) = \varepsilon$, and $\gamma Bx \neq \alpha Ay$.

Example 2.12

The next grammar is a $LR(1)$ grammar. $G = (\{S', S\}, \{a, b\}, P', S')$, the derivation rules are:

$S' \to S$

$S \to SaSb \mid \varepsilon$

In the next example we show that there is a context-free grammar, such that is not $LR(k)$ grammar for any k. ($k \geq 0$).

Example 2.13 Let $G' = (\{S', S\}, \{a\}, P', S')$ be a grammar and let the derivation rules be

$S' \rightarrow S$

$S \rightarrow aSa \mid a$

Now for all k $(k \geq 0)$

$$S' \stackrel{*}{\Longrightarrow} a^k S a^k \Longrightarrow a^k a a^k = a^{2k+1} ,$$
$$S' \stackrel{*}{\Longrightarrow} a^{k+1} S a^{k+1} \Longrightarrow a^{k+1} a a^{k+1} = a^{2k+3} ,$$

and

$$First_k(a^k) = First_k(a a^{k+1}) = a^k ,$$

but

$$a^{k+1} S a^{k+1} \neq a^k S a^{k+2} .$$

It is not sure that, for a $LL(k)$ $(k > 1)$ grammar, we can find an equivalent $LL(1)$ grammar. However, $LR(k)$ grammars have this nice property.

Theorem 2.16 *For all $LR(k)$ $(k > 1)$ grammar there is an equivalent $LR(1)$ grammar.*

The great significance of this theorem is that it makes sufficient to study the $LR(1)$ grammars instead of $LR(k)$ $(k > 1)$ grammars.

***LR*(1) canonical sets** Now we define a very important notion of the LR parsings.

Definition 2.17 *If β is the handle of the $\alpha\beta x$ $(\alpha, \beta \in (N \cup T)^*, x \in T^*)$ sentential form, then the prefixes of $\alpha\beta$ are the **viable prefixes** of $\alpha\beta x$.*

Example 2.14 Let $G' = (\{E, T, S'\}, \{i, +, (,)\}, P', S')$ be a grammar and the derivation rule as follows.

(0) $S' \rightarrow E$

(1) $E \rightarrow T$

(2) $E \rightarrow E + T$

(3) $T \rightarrow i$

(4) $T \rightarrow (E)$

$E + (i + i)$ is a sentential form, and the first i is the handle. The viable prefixes of this sentential form are E, $E+$, $E + ($, $E + (i$.

By the above definition, symbols after the handle are not parts of any viable prefix. Hence the task of finding the handle is the task of finding the longest viable prefix.

For a given grammar, the set of viable prefixes is determined, but it is obvious that the size of this set is not always finite.

The significance of viable prefixes are the following. We can assign states of a deterministic finite automaton to viable prefixes, and we can assign state transitions to the symbols of the grammar. From the initial state we go to a state along the symbols of a viable prefix. Using this property, we will give a method to create an automaton that executes the task of parsing.

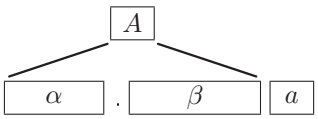


Figure 2.13. The $[A \to \alpha.\beta, a]$ $LR(1)$ -item.

Definition 2.18 *If $A \to \alpha\beta$ is a rule of a G' grammar, then let*

$$[A \to \alpha.\beta, a], \quad (a \in T \cup \{\#\}),$$

be a ***LR(1)-item****, where $A \to \alpha.\beta$ is the* ***core*** *of the $LR(1)$-item, and a is the* ***lookahead symbol*** *of the $LR(1)$-item.*

The lookahead symbol is instrumental in reduction, i.e. it has form $[A \to \alpha., a]$. It means that we can execute reduction only if the symbol a follows the handle *alpha*.

Definition 2.19 *The $LR(1)$-item $[A \to \alpha.\beta, a]$ is* ***valid*** *for the viable prefix $\gamma\alpha$ if*

$$S' \overset{*}{\Longrightarrow} \gamma Ax \Longrightarrow \gamma\alpha\beta x \ (\gamma \in (N \cup T)^*, \ x \in T^*),$$

and a is the first symbol of x or if $x = \varepsilon$ then $a = \#$.

Example 2.15 Let $G' = (\{S', S, A\}, \{a, b\}, P', S')$ a grammar and the derivation rules as follows.

(0) $S' \to S$
(1) $S \to AA$
(2) $A \to aA$
(3) $A \to b$

Using these rules, we can derive $S' \overset{*}{\Longrightarrow} aaAab \Longrightarrow aaaAab$. Here aaa is a viable prefix, and $[A \to a.A, a]$ is valid for this viable prefix. Similarly, $S' \overset{*}{\Longrightarrow} AaA \Longrightarrow AaaA$, and $LR(1)$-item $[A \to a.A, \#]$ is valid for viable prefix Aaa.

Creating a $LR(1)$ parser, we construct the canonical sets of $LR(1)$-items. To achieve this we have to define the *closure* and *read* functions.

Definition 2.20 *Let the set $\mathcal{H}$ be a set of $LR(1)$-items for a given grammar. The set* ***closure($\mathcal{H}$)*** *consists of the next $LR(1)$-items:*

1. *every element of the set $\mathcal{H}$ is an element of the set closure($\mathcal{H}$),*
2. *if $[A \to \alpha.B\beta, a] \in$ closure($\mathcal{H}$), and $B \to \gamma$ is a derivation rule of the grammar, then $[B \to .\gamma, b] \in$ closure($\mathcal{H}$) for all $b \in First(\beta a)$,*
3. *the set closure($\mathcal{H}$) is needed to expand using the step 2 until no more items can be added to it.*

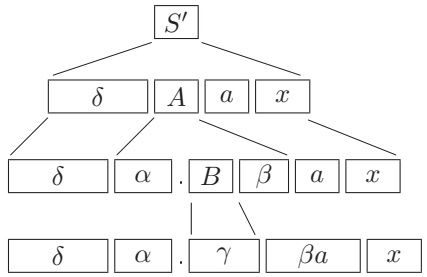


Figure 2.14. The function *closure*($[A \to \alpha.B\beta, a]$).

By definitions, if the $LR(1)$-item $[A \to \alpha.B\beta, a]$ is valid for the viable prefix $\delta\alpha$, then the $LR(1)$-item $[B \to .\gamma, b]$ is valid for the same viable prefix in the case of $b \in \mathit{First}(\beta a)$. (Figure 2.14). It is obvious that the function *closure* creates all of $LR(1)$-items which are valid for viable prefix $\delta\alpha$.

We can define the function *closure*($\mathcal{H}$), i.e. the closure of set $\mathcal{H}$ by the following algorithm. The result of this algorithm is the set $\mathcal{K}$.

CLOSURE-SET-OF-ITEMS($\mathcal{H}$)

```
1  K ← ∅
2  for all E ∈ H LR(1)-item
3       do K ← K ∪ CLOSURE-ITEM(E)
4  return K
```

CLOSURE-ITEM(E)

```
1  K_E ← {E}
2  if the LR(1)-item E has form [A → α.Bβ, a]
3     then I ← ∅
4          J ← K_E
5          repeat
6                for for all LR(1)-items ∈ J which have form [C → γ.Dδ, b]
7                    do for for all rules D → η ∈ P
8                           do for for all symbols c ∈ FIRST(δb)
9                                 do I ← I ∪ [D → .η, c]
10               J ← I
11               if I ≠ ∅
12                  then K_E ← K_E ∪ I
13                       I ← ∅
14         until J ≠ ∅
15 return K_E
```

The algorithm CLOSURE-ITEM creates $\mathcal{K}_E$, the closure of item E. If, in the argument E, the "point" is followed by a terminal symbol, then the result is this

item only (line 1). If in E the "point" is followed by a nonterminal symbol B, then we can create new items from every rule having the symbol B at their left side (line 9). We have to check this condition for all new items, too, the **repeat** cycle is in line 5–14. These steps are executed until no more items can be added (line 14). The set J contains the items to be checked, the set I contains the new items. We can find the operation $J \leftarrow I$ in line 10.

Definition 2.21 *Let $\mathcal{H}$ be a set of $LR(1)$-items for the grammar G. Then the set* ***read($\mathcal{H}$, X)*** *($X \in (N \cup T)$) consists of the following $LR(1)$-items.*

1. *if $[A \to \alpha.X\beta, a] \in \mathcal{H}$, then all items of the set closure$([A \to \alpha X.\beta, a])$ are in read$(\mathcal{H}, X)$,*

2. *the set read$(\mathcal{H}, X)$ is extended using step 1 until no more items can be added to it.*

The function *read*$(\mathcal{H}, X)$ "reads symbol X" in items of $\mathcal{H}$, and after this operation the sign "point" in the items gets to the right side of X. If the set $\mathcal{H}$ contains the valid $LR(1)$-items for the viable prefix γ then the set *read*$(\mathcal{H}, X)$ contains the valid $LR(1)$-items for the viable prefix γX.

The algorithm READ-SET-OF-ITEMS executes the function *read*. The result is the set $\mathcal{K}$.

READ-SET-OF-ITEMS$(\mathcal{H}, Y)$

```
1  K ← ∅
2  for all E ∈ H
3       do K ← K ∪ READ-ITEM(E, Y)
4  return K
```

READ-ITEM(E, Y)

```
1  if E = [A → α.Xβ, a] and X = Y
2     then K_{E,Y} ← CLOSURE-ITEM([A → αX.β, a])
3     else  K_{E,Y} ← ∅
4  return K_{E,Y}
```

Using these algorithms we can create all of items which writes the state after reading of symbol Y.

Now we introduce the following notation for $LR(1)$-items, to give shorter descriptions. Let

$$[A \to \alpha.X\beta, a/b]$$

be a notation for items

$$[A \to \alpha.X\beta, a] \text{ and } [A \to \alpha.X\beta, b]\,.$$

Example 2.16 The $LR(1)$-item $[S' \to .S, \#]$ is an item of the grammar in the example 2.15. For this item

$$closure(\ S' \to .S, \#\ ) = \{\ S' \to .S, \#\ \ , [S \to .AA, \#]\,, [A \to .aA, a/b]\,, [A \to .b, a/b]\}\ .$$

We can create the *canonical sets of $LR(1)$-items* or shortly the *$LR(1)$-canonical sets* with the following method.

Definition 2.22 *Canonical sets of $LR(1)$-items $\mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_m$ are the following.*

- $\mathcal{H}_0 = closure([S' \to .S, \#])$,
- *Create the set $read(\mathcal{H}_0, X)$ for a symbol X. If this set is not empty and it is not equal to canonical set $\mathcal{H}_0$ then it is the next canonical set $\mathcal{H}_1$.*

  Repeat this operation for all possible terminal and nonterminal symbol X. If we get a nonempty set which is not equal to any of previous sets then this set is a new canonical set, and its index is greater by one as the maximal index of previously generated canonical sets.
- *repeat the above operation for all previously generated canonical sets and for all symbols of the grammar until no more items can be added to it.*

  The sets

$$\mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_m$$

are the canonical sets of $LR(1)$-items of the grammar G.

The number of elements of $LR(1)$-items for a grammar is finite, hence the above method is terminated in finite time.

The next algorithm creates canonical sets of the grammar G.

CREATE-CANONICAL-SETS(G)

```
1  i ← 0
2  H_i ← CLOSURE-ITEM([S' → .S, #])
3  I ← {H_i}, K ← {H_i}
4  repeat
5          L ← K
6          for all M ∈ I-re
7              do I ← I \ M
8                  for all X ∈ T ∪ N-re
9                      do J ← CLOSURE-SET-OF-ITEMS(READ-SET-OF-ITEMS(M, X))
10                         if J ≠ ∅ and J ∉ K
11                            then i ← i + 1
12                                 H_i ← J
13                                 K ← K ∪ {H_i}
14                                 I ← I ∪ {H_i}
15 until K = L
16 return K
```

The result of the algorithm is K. The first canonical set is the set $\mathcal{H}_0$ in the line 2. Further canonical sets are created by functions CLOSURE-SET-OF-ITEMS(READ-SET-OF-ITEMS) in the line 9. The program in the line 10 checks that the new set differs from previous sets, and if the answer is true then this set will be a new set in lines 11–12. The **for** cycle in lines 6–14 guarantees that these operations are executed for all sets previously generated. In lines 3–14 the **repeat** cycle generate new canonical sets as long as it is possible.

Example 2.17 The canonical sets of $LR(1)$-items for the Example 2.15 are as follows.

$$
\begin{array}{lllll}
\mathcal{H}_0 & & = \mathit{closure}([S' \to .S]) & = & \{[S' \to .S, \#], [S \to .AA, \#], \\
 & & & & [A \to .aA, a/b], [A \to .b, a/b]\} \\
\mathcal{H}_1 = \mathit{read}(\mathcal{H}_0, S) & & = \mathit{closure}([S' \to S., \#]) & = & \{[S' \to S., \#]\} \\
\mathcal{H}_2 = \mathit{read}(\mathcal{H}_0, A) & & = \mathit{closure}([S' \to A.A, \#]) & = & \{[S \to A.A, \#], [A \to .aA, \#], \\
 & & & & [A \to .b, \#]\} \\
\mathcal{H}_3 = \mathit{read}(\mathcal{H}_0, a) & & = \mathit{closure}([A \to a.A, a/b]) & = & \{[A \to a.A, a/b], [A \to .aA, a/b], \\
 & & & & [A \to .b, a/b]\} \\
\mathcal{H}_4 = \mathit{read}(\mathcal{H}_0, b) & & = \mathit{closure}([A \to b., a/b]) & = & \{[A \to b., a/b]\} \\
\mathcal{H}_5 = \mathit{read}(\mathcal{H}_2, A) & & = \mathit{closure}([S \to AA., \#]) & = & \{[S \to AA., \#]\} \\
\mathcal{H}_6 = \mathit{read}(\mathcal{H}_2, a) & & = \mathit{closure}([A \to a.A, \#]) & = & \{[A \to a.A, \#], [A \to .aA, \#], \\
 & & & & [A \to .b, \#]\} \\
\mathcal{H}_7 = \mathit{read}(\mathcal{H}_2, b) & & = \mathit{closure}([A \to b., \#]) & = & \{[A \to b., \#]\} \\
\mathcal{H}_8 = \mathit{read}(\mathcal{H}_3, A) & & = \mathit{closure}([A \to aA., a/b]) & = & \{[A \to aA., a/b]\} \\
 & \mathit{read}(\mathcal{H}_3, a) & = \mathcal{H}_3 & & \\
 & \mathit{read}(\mathcal{H}_3, b) & = \mathcal{H}_4 & & \\
\mathcal{H}_9 = \mathit{read}(\mathcal{H}_6, A) & & = \mathit{closure}([A \to aA., \#]) & = & \{[A \to aA., \#]\} \\
 & \mathit{read}(\mathcal{H}_6, a) & = \mathcal{H}_6 & & \\
 & \mathit{read}(\mathcal{H}_6, b) & = \mathcal{H}_7 & &
\end{array}
$$

The automaton of the parser is in Figure 2.15.

***LR*(1) parser** If the canonical sets of $LR(1)$-items

$$\mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_m$$

were created, then assign the state k of an automaton to the set $\mathcal{H}_k$. Relation between the states of the automaton and the canonical sets of $LR(1)$-items is stated by the next theorem. This theorem is the ***"great" theorem of the LR(1)-parsing***.

Theorem 2.23 *The set of the $LR(1)$-items being valid for a viable prefix γ can be assigned to the automaton-state k such that there is path from the initial state to state k labeled by gamma.*

This theorem states that we can create the automaton of the parser using canonical sets. Now we give a method to create this $LR(1)$ parser from canonical sets of $LR(1)$-items.

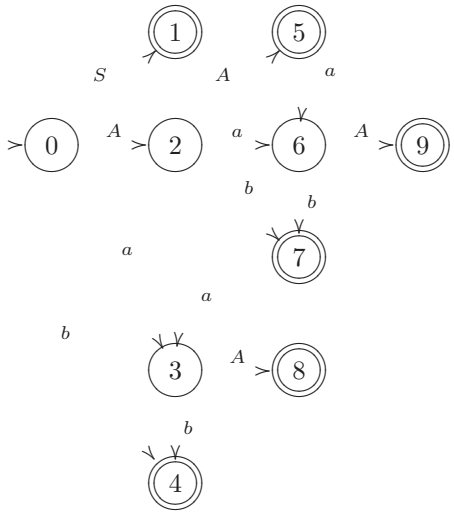


Figure 2.15. The automaton of the Example 2.15.

The deterministic finite automaton can be described with a table, that is called $LR(1)$ *parsing table*. The rows of the table are assigned to the states of the automaton.

The parsing table has two parts. The first is the *action* table. Since the operations of parser are determined by the symbols of analysed text, the *action* table is divided into columns labeled by the terminal symbols. The *action* table contains information about the action performing at the given state and at the given symbol. These actions can be shifts or reductions. The sign of a shift operation is sj, where j is the next state. The sign of the reduction is ri, where i is the serial number of the applied rule. The reduction by the rule having the serial number zero means the termination of the parsing and that the parsed text is syntactically correct; for this reason we call this operation *accept*.

The second part of the parsing table is the *goto* table. In this table are informations about shifts caused by nonterminals. (Shifts belong to terminals are in the action table.)

Let $\{0, 1, \ldots, m\}$ be the set of states of the automata. The i-th row of the table is filled in from the $LR(1)$-items of canonical set $\mathcal{H}_i$.

The i-th row of the *action* table:

- if $[A \to \alpha.a\beta, b] \in \mathcal{H}_i$ and $read(\mathcal{H}_i, a) = \mathcal{H}_j$ then $action[i, a] = sj$,
- if $[A \to \alpha., a] \in \mathcal{H}_i$ and $A \neq S'$, then $action[i, a] = rl$, where $A \to \alpha$ is the l-th rule of the grammar,
- if $[S' \to S., \#] \in \mathcal{H}_i$, then $action[i, \#] = accept$.

The method of filling in the *goto* table:

- if $read(\mathcal{H}_i, A) = \mathcal{H}_j$, then $goto[i, A] = j$.
- In both table we have to write the text *error* into the empty positions.

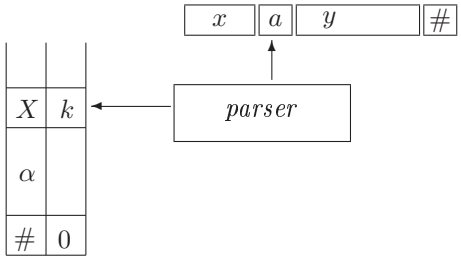


Figure 2.16. The structure of the $LR(1)$ parser.

These *action* and *goto* tables are called *canonical parsing tables.*

Theorem 2.24 *The augmented grammar G' is $LR(1)$ grammar iff we can fill in the parsing tables created for this grammar without conflicts.*

We can fill in the parsing tables with the next algorithm.

FILL-IN-LR(1)-TABLE(G)

```
1  for all LR(1) canonical sets H_i
2      do for all LR(1)-items
3             if [A → α.aβ, b] ∈ H_i and read(H_i, a) = H_j
4                then action[i, a] = sj
5             if [A → α., a] ∈ H_i and A ≠ S' and A → α the l-th rule
6                then action[i, a] = rl
7             if [S' → S., #] ∈ H_i
8                then action[i, #] = accept
9             if read(H_i, A) = H_j
10               then goto[i, A] = j
11     for all a ∈ (T ∪ {#})
12         do if action[i, a] = „empty"
13               then action[i, a] ← error
14     for all X ∈ N
15         do if goto[i, X] = „empty"
16              then goto[i, X] ← error
17 return action, goto
```

We fill in the tables its line-by-line. In lines 2–6 of the algorithm we fill in the *action* table, in lines 9–10 we fill in the *goto* table. In lines 11–13 we write the *error* into the positions which remained empty.

Now we deal with the steps of the $LR(1)$ parsing. (Figure 2.16).

The *state of the parsing* is written by configurations. A configuration of the $LR(1)$ parser consists of two parts, the first is the stack and the second is the unexpended input text.

The stack of the parsing is a double stack, we write or read two data with the operations *push* or *pop*. The stack consists of pairs of symbols, the first element of pairs there is a terminal or nonterminal symbol, and the second element is the serial number of the state of automaton. The content of the start state is #0.

The *start configuration* is $(\#0, z\#)$, where z means the unexpected text.

The parsing is successful if the parser moves to *final state*. In the final state the content of the stack is #0, and the parser is at the end of the text.

Suppose that the parser is in the configuration $(\#0 \dots Y_k i_k, ay\#)$. The next move of the parser is determined by $action[i_k, a]$.

State transitions are the following.

- If $action[i_k, a] = sl$, i.e. the parser executes a shift, then the actual symbol a and the new state l are written into the stack. That is, the new configuration is

$$(\#0 \dots Y_k i_k, ay\#) \to (\#0 \dots Y_k i_k a i_l, y\#) \ .$$

- If $action[i_k, a] = rl$, then we execute a reduction by the i-th rule $A \to \alpha$. In this step we delete $|\alpha|$ rows, i.e. we delete $2|\alpha|$ elements from the stack, and then we determine the new state using the *goto* table. If after the deletion there is the state i_{k-r} at the top of the stack, then the new state is $goto[i_{k-r}, A] = i_l$.

$$(\#0 \dots Y_{k-r} i_{k-r} Y_{k-r+1} i_{k-r+1} \dots Y_k i_k, y\#) \to (\#0 \dots Y_{k-r} i_{k-r} A i_l, y\#) \ ,$$

  where $|\alpha| = r$.
- If $action[i_k, a] = accept$, then the parsing is completed, and the analysed text was correct.
- If $action[i_k, a] = error$, then the parsing terminates, and a *syntactic error* was discovered at the symbol a.

The $LR(1)$ parser is often named *canonical* $LR(1)$ *parser*.

Denote the *action* and *goto* tables together by T. We can give the following algorithm for the steps of parser.

LR(1)-PARSER$(xay\#, T)$

```
1  s ← (#0, xay#), s' ← parsing
2  repeat
3          s = (#0 ... Y_{k-r} i_{k-r} Y_{k-r+1} i_{k-r+1} ... Y_k i_k, ay#)
4          if action[i_k, a] = sl
5             then s ← (#0 ... Y_k i_k a i_l, y#)
6             else if action[i_k, a] = rl and A → α is the l-th rule and
7                     |α| = r and goto[i_{k-r}, A] = i_l
8                     then s ← (#0 ... Y_{k-r} i_{k-r} A i_l, ay#)
9                     else if action[i_k, a] = accept
10                            then s' ← O.K.
11                            else  s' ← ERROR
12 until s' = O.K. or s' = ERROR
13 return s', s
```

The input parameters of the algorithm are the text xay and table T. The variable s' indicates the action of the parser. It has value *parsing* in the intermediate states, and its value is *O.K.* or *ERROR* at the final states. In line 3 we detail the configuration of the parser, that is necessary at lines 6–8. Using the *action* table, the parser determines its move from the symbol x_k at the top of the stack and from the actual symbol a. In lines 4–5 we execute a shift step, in lines 6–8 a reduction. The algorithm is completed in lines 9–11. At this moment, if the parser is at the end of text and the state 0 is at the top of stack, then the text is correct, otherwise a syntax error was detected. According to this, the output of the algorithm is *O.K.* or *ERROR*, and the final configuration is at the output, too. In the case of error, the first symbol of the second element of the configuration is the erroneous symbol.

Example 2.18 The *action* and *goto* tables of the $LR(1)$ parser for the grammar of Example 2.15 are as follows. The empty positions denote *errors*.

state	*action*			*goto*	
	a	b	$\#$	S	A
0	$s3$	$s4$		1	2
1			*accept*		
2	$s6$	$s7$			5
3	$s3$	$s4$			8
4	$r3$	$r3$			
5			$r1$		
6	$s6$	$s7$			9
7			$r3$		
8	$r2$	$r2$			
9			$r2$		

Example 2.19 Using the tables of the previous example, analyse the text $abb\#$.

			rule
$(\#0,$	$aab\#)$		
$\xrightarrow{s3}$	$(\#0a3,$	$bb\#)$	
$\xrightarrow{s4}$	$(\#0a3b4,$	$b\#)$	
$\xrightarrow{r3}$	$(\#0a3A8,$	$b\#)$	$A \to b$
$\xrightarrow{r2}$	$(\#0A2,$	$b\#)$	$A \to aA$
$\xrightarrow{s7}$	$(\#0A2b7,$	$\#)$	
$\xrightarrow{r3}$	$(\#0A2A5,$	$\#)$	$A \to b$
$\xrightarrow{r1}$	$(\#0S1,$	$\#)$	$S \to AA$
$\xrightarrow{elfogad}$	*O.K.*		

The syntax tree of the sentence is in Figure 2.17.

***LALR*(1) parser** Our goal is to decrease the number of states of the parser, since not only the size but the speed of the compiler is dependent on the number of states. At the same time, we wish not to cut radically the set of $LR(1)$ grammars

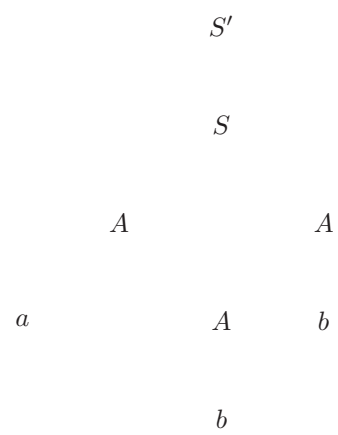


Figure 2.17. The syntax tree of the sentence *aab*.

and languages, by using our new method.

There are a lot of $LR(1)$-items in the canonical sets, such that are very similar: their core are the same, only their lookahead symbols are different. If there are two or more canonical sets in which there are similar items only, then we merge these sets.

If the canonical sets $\mathcal{H}_i$ és a $\mathcal{H}_j$ are mergeable, then let $\mathcal{K}_{[i,j]} = \mathcal{H}_i \cup \mathcal{H}_j$.

Execute all of possible merging of $LR(1)$ canonical sets. After renumbering the indexes we obtain sets $\mathcal{K}_0, \mathcal{K}_1, \ldots, \mathcal{K}_n$; these are the *merged* $LR(1)$ *canonical sets* or $LALR(1)$ *canonical sets*.

We create the $LALR(1)$ parser from these united canonical sets.

Example 2.20 Using the $LR(1)$ canonical sets of the example 2.17, we can merge the next canonical sets:

$\mathcal{H}_3$ and $\mathcal{H}_6$,

$\mathcal{H}_4$ and $\mathcal{H}_7$,

$\mathcal{H}_8$ and $\mathcal{H}_9$.

In the Figure 2.15 it can be seen that mergeable sets are in equivalent or similar positions in the automaton.

There is no difficulty with the function *read* if we use merged canonical sets. If

$$\mathcal{K} = \mathcal{H}_1 \cup \mathcal{H}_2 \cup \ldots \cup \mathcal{H}_k \ ,$$

$$read(\mathcal{H}_1, X) = \mathcal{H}_1^{'}, read(\mathcal{H}_2, X) = \mathcal{H}_2^{'}, \ldots, read(\mathcal{H}_k, X) = \mathcal{H}_k^{'} \ ,$$

and

$$\mathcal{K}' = \mathcal{H}_1^{'} \cup \mathcal{H}_2^{'} \cup \ldots \cup \mathcal{H}_k^{'} \ ,$$

then

$$read(\mathcal{K}, X) = \mathcal{K}' \ .$$

We can prove this on the following way. By the definition of function *read*, the

set $read(\mathcal{H}, X)$ depends on the core of $LR(1)$-items in $\mathcal{H}$ only, and it is independent of the lookahead symbols. Since the cores of $LR(1)$-items in the sets $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_k$ are the same, the cores of $LR(1)$-items of

$$read(\mathcal{H}_1, X), read(\mathcal{H}_2, X), \ldots, read(\mathcal{H}_k, X)$$

are also the same. It follows that these sets are mergeable into a set $\mathcal{K}'$, thus $read(\mathcal{K}, X) = \mathcal{K}'$.

However, after merging canonical sets of $LR(1)$-items, elements of this set can raise difficulties. Suppose that

$\mathcal{K}_{[i,j]} = \mathcal{H}_i \cup \mathcal{H}_j$.

- After merging there are not *shift-shift conflicts*. If

  $$[A \to \alpha.a\beta, b] \in \mathcal{H}_i$$

  and

  $$[B \to \gamma.a\delta, c] \in \mathcal{H}_j$$

  then there is a shift for the symbol a and we saw that the function *read* does not cause problem, i.e. the set $read(\mathcal{K}_{[i,j]}, a)$ is equal to the set $read(\mathcal{H}_i, a) \cup read(\mathcal{H}_j, a)$.
- If there is an item

  $$[A \to \alpha.a\beta, b]$$

  in the canonical set $\mathcal{H}_i$ and there is an item

  $$[B \to \gamma., a]$$

  in the set a $\mathcal{H}_j$, then the merged set is an inadequate set with the symbol a, i.e. there is a *shift-reduce conflict* in the merged set.

  But this case never happens. Both items are elements of the set $\mathcal{H}_i$ and of the set $\mathcal{H}_j$. These sets are mergeable sets, thus they are different in lookahead symbols only. It follows that there is an item $[A \to \alpha.a\beta, c]$ in the set $\mathcal{H}_j$. Using the Theorem 2.24 we get that the grammar is not a $LR(1)$ grammar; we get shift-reduce conflict from the set $\mathcal{H}_j$ for the $LR(1)$ parser, too.
- However, after merging *reduce-reduce conflict* may arise. The properties of $LR(1)$ grammar do not exclude this case. In the next example we show such a case.

Example 2.21 Let $G' = (\{S', S, A, B\}, \{a, b, c, d, e\}, P', S')$ be a grammar, and the derivation rules are as follows.

$S' \to S$
$S \to aAd \mid bBd \mid aBe \mid bAe$
$A \to c$
$B \to c$

This grammar is a $LR(1)$ grammar. For the viable prefix ac the $LR(1)$-items

$$\{[A \to c., d], [B \to c., e]\},$$

for the viable prefix bc the $LR(1)$-items

$$\{[A \to c., e], [B \to c., d]\}$$

create two canonical sets.

After merging these two sets we get a reduce-reduce conflict. If the input symbol is d or e then the handle is c, but we cannot decide that if we have to use the rule $A \to c$ or the rule $B \to c$ for reducing.

Now we give the method for creating a $LALR(1)$ parsing table. First we give the canonical sets of $LR(1)$-items

$$\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_m$$

, then we merge canonical sets in which the sets constructed from the core of the items are identical ones. Let

$$\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_n \quad (n \leq m)$$

be the $LALR(1)$ canonical sets.

For the calculation of the size of the *action* and *goto* tables and for filling in these tables we use the sets $\mathcal{K}_i \quad (1 \leq i \leq n)$. The method is the same as it was in the $LR(1)$ parsers. The constructed tables are named by $LALR(1)$ *parsing tables.*

Definition 2.25 *If the filling in the $LALR(1)$ parsing tables do not produce conflicts then the grammar is said to be an* ***LALR(1)*** *grammar.*

The run of $LALR(1)$ parser is the same as it was in $LR(1)$ parser.

Example 2.22 Denote the result of merging canonical sets $\mathcal{H}_i$ and $\mathcal{H}_j$ by $\mathcal{K}_{[i,j]}$. Let $[i, j]$ be the state which belonging to this set.

The $LR(1)$ canonical sets of the grammar of Example 2.15 were given in the Example 2.17 and the mergeable sets were seen in the example 2.20. For this grammar we can create the next $LALR(1)$ parsing tables.

állapot	*action*			*goto*	
	a	b	#	S	A
0	$s\,[3,6]$	$s\,[4,7]$		1	2
1			*accept*		
2	$s\,[3,6]$	$s\,[4,7]$			5
$[3,6]$	$s\,[3,6]$	$s\,[4,7]$			$[8,9]$
$[4,7]$	$r3$	$r3$	$r3$		
5			$r1$		
$[8,9]$	$r2$	$r2$	$r2$		

The filling in the $LALR(1)$ tables are conflict free, therefore the grammar is an $LALR(1)$ grammar. The automaton of this parser is in Figure 2.18.

Example 2.23 Analyse the text $abb\#$ using the parsing table of the previous example.

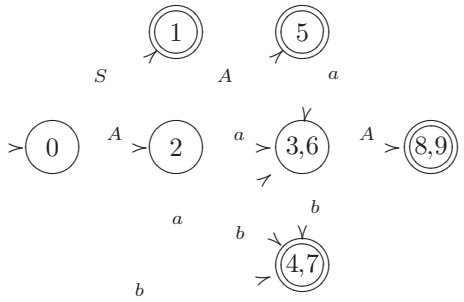


Figure 2.18. The automaton of the Example 2.22.

					rule
$(\#0,$	$aab\#)$	$\xrightarrow{s[3,6]}$	$(\#0a\,[3,6]\,,$	$bb\#)$	
		$\xrightarrow{s[4,7]}$	$(\#0a\,[3,6]\,b\,[4,7]\,,$	$b\#)$	
		$\xrightarrow{r3}$	$(\#0a\,[3,6]\,A[8,9],$	$b\#)$	$A \to b$
		$\xrightarrow{r2}$	$(\#0A2,$	$b\#)$	$A \to aA$
		$\xrightarrow{s[4,7]}$	$(\#0A2b\,[4,7]\,,$	$\#)$	
		$\xrightarrow{r3}$	$(\#0A2A5,$	$\#)$	$A \to b$
		$\xrightarrow{r1}$	$(\#0S1,$	$\#)$	$S \to AA$
		$\xrightarrow{elfogad}$	$O.K.$		

The syntax tree of the parsed text is in the Figure 2.17.

As it can be seen from the previous example, the $LALR(1)$ grammars are $LR(1)$ grammars. The converse assertion is not true. In the example 2.21 there is a grammar which is $LR(1)$, but it is not $LALR(1)$ grammar.

Programming languages can be written by $LALR(1)$ grammars. The most frequently used methods in compilers of programming languages is the $LALR(1)$ method. The advantage of the $LALR(1)$ parser is that the sizes of parsing tables are smaller than the size of $LR(1)$ parsing tables.

For example, the $LALR(1)$ parsing tables for the Pascal language have a few hundreds of lines, whilst the $LR(1)$ parsers for this language have a few thousands of lines.

Exercises

2.3-1 Find the $LL(1)$ grammars among the following grammars (we give their derivation rules only).

1. $S \to ABc$
   $A \to a \mid \varepsilon$
   $B \to b \mid \varepsilon$

2. $S \to Ab$
   $A \to a \mid B \mid \varepsilon$
   $B \to b \mid \varepsilon$

3. $S \rightarrow ABBA$
   $A \rightarrow a \mid \varepsilon$
   $B \rightarrow b \mid \varepsilon$

4. $S \rightarrow aSe \mid A$
   $A \rightarrow bAe \mid B$
   $B \rightarrow cBe \mid d$

2.3-2 Prove that the next grammars are $LL(1)$ grammars (we give their derivation rules only).

1. $S \rightarrow Bb \mid Cd$
   $B \rightarrow aB \mid \varepsilon$
   $C \rightarrow cC \mid \varepsilon$

2. $S \rightarrow aSA \mid \varepsilon$
   $A \rightarrow c \mid bS$

3. $S \rightarrow AB$
   $A \rightarrow a \mid \varepsilon$
   $B \rightarrow b \mid \varepsilon$

2.3-3 Prove that the next grammars are not $LL(1)$ grammars (we give their derivation rules only).

1. $S \rightarrow aAa \mid Cd$
   $A \rightarrow abS \mid c$

2. $S \rightarrow aAaa \mid bAba$
   $A \rightarrow b \mid \varepsilon$

3. $S \rightarrow abA \mid \varepsilon$
   $A \rightarrow Saa \mid b$

2.3-4 Show that a $LL(0)$ language has only one sentence.

2.3-5 Prove that the next grammars are $LR(0)$ grammars (we give their derivation rules only).

1. $S' \rightarrow S$
   $S \rightarrow aSa \mid aSb \mid c$

2. $S' \rightarrow S$
   $S \rightarrow aAc$
   $A \rightarrow Abb \mid b$

2.3-6 Prove that the next grammars are $LR(1)$ grammars. (we give their derivation rules only).

1. $S' \rightarrow S$
   $S \rightarrow aSS \mid b$

2. $S' \rightarrow S$
   $S \rightarrow SSa \mid b$

2.3-7 Prove that the next grammars are not $LR(k)$ grammars for any k (we give their derivation rules only).

1. $S' \rightarrow S$
   $S \rightarrow aSa \mid bSb \mid a \mid b$

2. $S' \rightarrow S$
   $S \rightarrow aSa \mid bSa \mid ab \mid ba$

2.3-8 Prove that the next grammars are $LR(1)$ but are not $LALR(1)$ grammars (we give their derivation rules only).

1. $S' \rightarrow S$
   $S \rightarrow Aa \mid bAc \mid Bc \mid bBa$
   $A \rightarrow d$
   $B \rightarrow d$

2. $S' \rightarrow S$
   $S \rightarrow aAcA \mid A \mid B$
   $A \rightarrow b \mid Ce$
   $B \rightarrow dD$
   $C \rightarrow b$
   $D \rightarrow CcS \mid CcD$

2.3-9 Create parsing table for the above $LL(1)$ grammars.
2.3-10 Using the recursive descent method, write the parsing program for the above $LL(1)$ grammars.
2.3-11 Create canonical sets and the parsing tables for the above $LR(1)$ grammars.
2.3-12 Create merged canonical sets and the parsing tables for the above $LALR(1)$ grammars.

Problems

2-1 Lexical analysis of a program text
The algorithm LEX-ANALYSE in Section 2.2 gives a scanner for the text that is described by *only one* regular expression or deterministic finite automaton, i.e. this scanner is able to analyse only one symbol. Create an automaton which executes total lexical analysis of a program language, and give the algorithm LEX-ANALYSE-LANGUAGE for this automaton. Let the input of the algorithm be the text of a program, and the output be the series of symbols. It is obvious that if the automaton goes into a finite state then its new work begins at the initial state, for analysing the next symbol. The algorithm finishes his work if it is at the end of the text or a lexical error is detected.

2-2 Series of symbols augmented with data of symbols
Modify the algorithm of the previous task on such way that the output is the series of symbols augmented with the appropriate attributes. For example, the attribute of a variable is the character string of its name, or the attribute of a number is its value and type. It is practical to write pointers to the symbols in places of data.

2-3 LALR(1) parser from LR (0) canonical sets
If we omit lookahead symbols from the $LR(1)$-items then we get ***LR(0)-items***. We can define functions *closure* and *read* for $LR(0)$-items too, doing not care for lookahead symbols. Using a method similar to the method of $LR(1)$, we can construct ***LR(0) canonical sets***

$$\mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_n.$$

One can observe that the number of merged canonical sets is equal to the number of $LR(0)$ canonical sets, since the cores of $LR(1)$-items of the merged canonical sets are the same as the items of the $LR(0)$ canonical sets. Therefore the number of states of $LALR(1)$ parser is equal to the number of states of its $LR(0)$ parser.

Using this property, we can construct $LALR(1)$ canonical sets from $LR(0)$ canonical sets, by completing the items of the $LR(0)$ canonical sets with lookahead symbols. The result of this procedure is the set of $LALR(1)$ canonical sets.

It is obvious that the right part of an $LR(1)$-item begins with symbol point only if this item was constructed by the function closure. (We notice that there is one exception, the $[S' \to .S]$ item of the canonical set $\mathcal{H}_0$.) Therefore it is no need for all items of $LR(1)$ canonical sets. Let the ***kernel*** of the canonical set $\mathcal{H}_0$ be the $LR(1)$-item $[S' \to .S, \#]$, and let the kernel of any other canonical set be the set of the $LR(1)$-items such that there is no point at the first position on the right side of the item. We give an $LR(1)$ canonical set by its kernel, since all of items can be construct from the kernel using the function *closure*.

If we complete the items of the kernel of $LR(0)$ canonical sets then we get the kernel of the merged $LR(1)$ canonical sets. That is, if the kernel of an $LR(0)$ canonical set is $\mathcal{I}_j$, then from it with completions we get the kernel of the $LR(1)$ canonical set, $\mathcal{K}_j$.

If we know $\mathcal{I}_j$ then we can construct $read(\mathcal{I}_j, X)$ easily. If $[B \to \gamma.C\delta] \in \mathcal{I}_j$, $C \stackrel{*}{\to} A\eta$ and $A \to X\alpha$, then $[A \to X.\alpha] \in read(\mathcal{I}_j, X)$. For $LR(1)$-items, if $[B \to \gamma.C\delta, b] \in \mathcal{K}_j$, $C \stackrel{*}{\to} A\eta$ and $A \to X\alpha$ then we have to determine also the lookahead symbols, i.e. the symbols a such that $[A \to X.\alpha, a] \in read(\mathcal{K}_j, X)$.

If $\eta\delta \neq \varepsilon$ and $a \in First(\eta\delta b)$ then it is sure that $[A \to X.\alpha, a] \in read(\mathcal{K}_j, X)$. In this case, we say that the lookahead symbol was ***spontaneously generated*** for this item of canonical set $read(\mathcal{K}_j, X)$. The symbol b do not play important role in the construction of the lookahead symbol.

If $\eta\delta = \varepsilon$ then $[A \to X.\alpha, b]$ is an element of the set $read(\mathcal{K}_j, X)$, and the lookahead symbol is b. In this case we say that the lookahead symbol is ***propagated*** from $\mathcal{K}_j$ into the item of the set $read(\mathcal{K}_j, X)$.

If the kernel $\mathcal{I}_j$ of an $LR(0)$ canonical set is given then we construct the propagated and spontaneously generated lookahead symbols for items of $read(\mathcal{K}_j, X)$ by the following algorithm.

For all items $[B \to \gamma.\delta] \in \mathcal{I}_j$ we construct the set $\mathcal{K}_j = closure([B \to \gamma.\delta, @])$,

where @ is a dummy symbol,

- if $[A \to \alpha.X\beta, a] \in \mathcal{K}_j$ and $a \neq @$ then $[A \to \alpha X.\beta, a] \in \mathit{read}(\mathcal{K}_j, X)$ and the symbol a is spontaneously generated into the item of the set $\mathit{read}(\mathcal{K}_j, X)$,
- if $[A \to \alpha.X\beta, @] \in \mathcal{K}_j$ then $[A \to \alpha X.\beta, @] \in \mathit{read}(\mathcal{K}_j, X)$, and the symbol @ is propagated from $\mathcal{K}_j$ into the item of the set $\mathit{read}(\mathcal{K}_j, X)$.

The kernel of the canonical set $\mathcal{K}_0$ has only one element. The core of this element is $[S' \to .S]$. For this item we can give the lookahead symbol # directly. Since the core of the kernel of all $\mathcal{K}_j$ canonical sets are given, using the above method we can calculate all of propagated and spontaneously generated symbols.

Give the algorithm which constructs $LALR(1)$ canonical sets from $LR(0)$ canonical sets using the methods of propagation and spontaneously generation.

Chapter notes

The theory and practice of compilers, computers and program languages are of the same age. The construction of first compilers date back to the 1950's. The task of writing compilers was a very hard task at that time, the first Fortran compiler took 18 man-years to implement [6]. From that time more and more precise definitions and solutions have been given to the problems of compilation, and better and better methods and utilities have been used in the construction of translators.

The development of formal languages and automata was a great leap forward, and we can say that this development was urged by the demand of writing of compilers. In our days this task is a simple routine project. New results, new discoveries are expected in the field of code optimisation only.

One of the earliest nondeterministic and backtrack algorithms appeared in the 1960's. The first two dynamic programming algorithms were the CYK (Cocke-Younger-Kasami) algorithm from 1965–67 and the Earley-algorithm from 1965. The idea of precedence parsers is from the end of 1970's and from the beginning of 1980's. The LR(k) grammars was defined by Knuth in 1965; the definition of LL(k) grammars is dated from the beginning of 1970's. $LALR(1)$ grammars were studied by De Remer in 1971, the elaborating of $LALR(1)$ parsing methods were finished in the beginning of 1980's [4, 5, 6].

To the middle of 1980's it became obvious that the LR parsing methods are the real efficient methods and since than the $LALR(1)$ methods are used in compilers [4].

A lot of very excellent books deal with the theory and practice of compiles. Perhaps the most successful of them was the book of Gries [94]; in this book there are interesting results for precedence grammars. The first successful book which wrote about the new LR algorithms was of Aho and Ullman [5], we can find here also the CYK and the Early algorithms. It was followed by the "dragon book" of Aho and Ullman[6]; the extended and corrected issue of it was published in 1986 by authors Aho, Ullman and Sethi [4].

Without completeness we notice the books of Fischer and LeBlanc [69], Tremblay and Sorenson [243], Waite and Goos [250], Hunter[115], Pittman [192] and Mak

[157]. Advanced achievements are in recently published books, among others in the book of Muchnick [175], Grune, Bal, Jacobs and Langendoen [96], in the book of Cooper and Torczon [49] and in a chapter of the book by Louden [154].

3. Compression and Decompression

Algorithms for data compression usually proceed as follows. They encode a text over some finite alphabet into a sequence of bits, hereby exploiting the fact that the letters of this alphabet occur with different frequencies. For instance, an "e" occurs more frequently than a "q" and will therefore be assigned a shorter codeword. The quality of the compression procedure is then measured in terms of the average codeword length.
So the underlying model is probabilistic, namely we consider a finite alphabet and a probability distribution on this alphabet, where the probability distribution reflects the (relative) frequencies of the letters. Such a pair—an alphabet with a probability distribution—is called a source. We shall first introduce some basic facts from Information Theory. Most important is the notion of entropy, since the source entropy characterises the achievable lower bounds for compressibility.

The source model to be best understood, is the discrete memoryless source. Here the letters occur independently of each other in the text. The use of prefix codes, in which no codeword is the beginning of another one, allows to compress the text down to the entropy of the source. We shall study this in detail. The lower bound is obtained via Kraft's inequality, the achievability is demonstrated by the use of Huffman codes, which can be shown to be optimal.

There are some assumptions on the discrete memoryless source, which are not fulfilled in most practical situations. Firstly, usually this source model is not realistic, since the letters do not occur independently in the text. Secondly, the probability distribution is not known in advance. So the coding algorithms should be universal for a whole class of probability distributions on the alphabet. The analysis of such universal coding techniques is much more involved than the analysis of the discrete memoryless source, such that we shall only present the algorithms and do not prove the quality of their performance. Universal coding techniques mainly fall into two classes.

Statistical coding techniques estimate the probability of the next letters as accurately as possible. This process is called modelling of the source. Having enough information about the probabilities, the text is encoded, where usually arithmetic coding is applied. Here the probability is represented by an interval and this interval will be encoded.

Dictionary-based algorithms store patterns, which occurred before in the text, in a dictionary and at the next occurrence of a pattern this is encoded via its position in the dictionary. The most prominent procedure of this kind is due to Ziv and Lempel.

We shall also present a third universal coding technique which falls in neither of these two classes. The algorithm due to Burrows and Wheeler has become quite prominent in recent years, since implementations based on it perform very well in practice.

All algorithms mentioned so far are lossless, i. e., there is no information lost after decoding. So the original text will be recovered without any errors. In contrast, there are lossy data compression techniques, where the text obtained after decoding does not completely coincide with the original text. Lossy compression algorithms are used in applications like image, sound, video, or speech compression. The loss should, of course, only marginally effect the quality. For instance, frequencies not realizable by the human eye or ear can be dropped. However, the understanding of such techniques requires a solid background in image, sound or speech processing, which would be far beyond the scope of this paper, such that we shall illustrate only the basic concepts behind image compression algorithms such as JPEG.

We emphasise here the recent developments such as the Burrows–Wheeler transform and the context–tree weighting method. Rigorous proofs will only be presented for the results on the discrete memoryless source which is best understood but not a very realistic source model in practice. However, it is also the basis for more complicated source models, where the calculations involve conditional probabilities. The asymptotic computational complexity of compression algorithms is often linear in the text length, since the algorithms simply parse through the text. However, the running time relevant for practical implementations is mostly determined by the constants as dictionary size in Ziv-Lempel coding or depth of the context tree, when arithmetic coding is applied. Further, an exact analysis or comparison of compression algorithms often heavily depends on the structure of the source or the type of file to be compressed, such that usually the performance of compression algorithms is tested on benchmark files. The most well-known collections of benchmark files are the Calgary Corpus and the Canterbury Corpus.

3.1. Facts from information theory

3.1.1. The Discrete Memoryless Source

The source model discussed throughout this chapter is the ***Discrete Memoryless Source*** (DMS). Such a source is a pair $(\mathcal{X}, P)$, where $\mathcal{X} = \{1, \ldots, m\}$, is a finite alphabet and $P = (P(1), \ldots, P(m))$ is a probability distribution on $\mathcal{X}$. A discrete memoryless source can also be described by a random variable X, where $\mathrm{Prob}(X = x) = P(x)$ for all $x \in \mathcal{X}$. A word $x^n = (x_1 x_2 \ldots x_n) \in \mathcal{X}^n$ is the realization of the random variable $(X_1 \ldots X_n)$, where the X_i's are identically distributed and independent of each other. So the probability $P^n(x_1 x_2 \ldots x_n) = P(x_1) \cdot P(x_2) \cdot \cdots \cdot P(x_n)$ is the product of the probabilities of the single letters.

A	64	H	42	N	56	U	31
B	14	I	63	O	56	V	10
C	27	J	3	P	17	W	10
D	35	K	6	Q	4	X	3
E	100	L	35	R	49	Y	18
F	20	M	20	S	56	Z	2
G	14	T	71				

Space/Punctuation mark 166

Figure 3.1. Frequency of letters in 1000 characters of English.

Estimations for the letter probabilities in natural languages are obtained by statistical methods. If we consider the English language and choose for $\mathcal{X}$ the latin alphabet with an additional symbol for Space and punctuation marks, the probability distribution can be derived from the frequency table in 3.1, which is obtained from the copy–fitting tables used by professional printers. So $P(A) = 0.064$, $P(B) = 0.014$, etc.

Observe that this source model is often not realistic. For instance, in English texts e.g. the combination 'th' occurs more often than 'ht'. This could not be the case, if an English text was produced by a discrete memoryless source, since then $P(th) = P(t) \cdot P(h) = P(ht)$.

In the discussion of the communication model it was pointed out that the encoder wants to compress the original data into a short sequence of binary digits, hereby using a binary code, i. e., a function $c : \mathcal{X} \longrightarrow \{0,1\}^* = \bigcup_{n=0}^{\infty} \{0,1\}^n$. To each element $x \in \mathcal{X}$ a codeword $c(x)$ is assigned. The aim of the encoder is to minimise the average length of the codewords. It turns out that the best possible data compression can be described in terms of the ***entropy*** $H(P)$ of the probability distribution P. The entropy is given by the formula

$$H(P) = -\sum_{x \in \mathcal{X}} P(x) \cdot \lg P(x) \ ,$$

where the logarithm is to the base 2. We shall also use the notation $H(X)$ according to the interpretation of the source as a random variable.

3.1.2. Prefix codes

A ***code*** (of variable length) is a function $c : \mathcal{X} \longrightarrow \{0,1\}^*$, $\mathcal{X} = \{1, \ldots, m\}$. Here $\{c(1), c(2), \ldots, c(m)\}$ is the set of ***codewords***, where for $x = 1, \ldots, m$ the codeword is $c(x) = \big(c_1(x), c_2(x), \ldots, c_{L(x)}(x)\big)$ where $L(x)$ denotes the ***length*** of $c(x)$, i. e., the number of bits used to present$c(x)$.

A code c is ***uniquely decipherable*** (UDC) , if every word in $\{0,1\}^*$ is representable by at most one sequence of codewords.

A code c is a ***prefix code,*** if no codeword is prefix of another one, i. e., for any two codewords $c(x)$ and $c(y)$, $x \neq y$, with $L(x) \leq L(y)$ holds

$\big(c_1(x), c_2(x), \ldots, c_{L(x)}(x)\big) \neq \big(c_1(y), c_2(y), \ldots, c_{L(x)}(y)\big)$. So in at least one of the first $L(x)$ components $c(x)$ and $c(y)$ differ.

Messages encoded using a prefix code are uniquely decipherable. The decoder proceeds by reading the next letter until a codeword $c(x)$ is formed. Since $c(x)$ cannot be the beginning of another codeword, it must correspond to the letter $x \in \mathcal{X}$. Now the decoder continues until another codeword is formed. The process may be repeated until the end of the message. So after having found the codeword $c(x)$ the decoder instantaneously knows that $x \in \mathcal{X}$ is the next letter of the message. Because of this property a prefix code is also denoted as instantaneous code.

The criterion for data compression is to minimise the average length of the codewords. So if we are given a source $(\mathcal{X}, P)$, where $\mathcal{X} = \{1, \ldots, m\}$ and $P = \big(P(1), P(2), \ldots, P(m)\big)$ is a probability distribution on $\mathcal{X}$, the ***average length*** $\overline{L}(c)$ is defined by

$$\overline{L}(c) = \sum_{x \in \mathcal{X}} P(x) \cdot L(x) \ .$$

The following prefix code c for English texts has average length $\overline{L}(c) = 3 \cdot 0.266 + 4 \cdot 0.415 + 5 \cdot 0.190 + 6 \cdot 0.101 + 7 \cdot 0.016 + 8 \cdot 0.012 = 4.222$.

$A \longrightarrow 0110$,	$B \longrightarrow 010111$,	$C \longrightarrow 10001$,	$D \longrightarrow 01001$,
$E \longrightarrow 110$,	$F \longrightarrow 11111$,	$G \longrightarrow 111110$,	$H \longrightarrow 00100$,
$I \longrightarrow 0111$,	$J \longrightarrow 11110110$,	$K \longrightarrow 1111010$,	$L \longrightarrow 01010$,
$M \longrightarrow 001010$,	$N \longrightarrow 1010$,	$O \longrightarrow 1001$,	$P \longrightarrow 010011$,
$Q \longrightarrow 01011010$,	$R \longrightarrow 1110$,	$S \longrightarrow 1011$,	$T \longrightarrow 0011$,
$U \longrightarrow 10000$,	$V \longrightarrow 0101100$,	$W \longrightarrow 001011$,	$X \longrightarrow 01011011$,
$Y \longrightarrow 010010$,	$Z \longrightarrow 11110111$,	$SP \longrightarrow 000$.	

We can still do better, if we do not encode single letters, but blocks of n letters for some $n \in N$. In this case we replace the source $(\mathcal{X}, P)$ by $(\mathcal{X}^n, P^n)$ for some $n \in N$. Remember that $P^n(x_1 x_2 \ldots x_n) = P(x_1) \cdot P(x_2) \cdot \cdots \cdot P(x_n)$ for a word $(x_1 x_2 \ldots x_n) \in \mathcal{X}^n$, since the source is memoryless. If e.g. we are given an alphabet with two letters, $\mathcal{X} = \{a, b\}$ and $P(a) = 0.9$, $P(b) = 0.1$, then the code c defined by $c(a) = 0$, $c(b) = 1$ has average length $\overline{L}(c) = 0.9 \cdot 1 + 0.1 \cdot 1 = 1$. Obviously we cannot find a better code. The combinations of two letters now have the following probabilities:

$$P^2(aa) = 0.81, \quad P^2(ab) = 0.09, \quad P^2(ba) = 0.09, \quad P^2(bb) = 0.01 \ .$$

The prefix code c^2 defined by

$$c^2(aa) = 0, \quad c^2(ab) = 10, \quad c^2(ba) = 110, \quad c^2(bb) = 111$$

has average length $\overline{L}(c^2) = 1 \cdot 0.81 + 2 \cdot 0.09 + 3 \cdot 0.09 + 3 \cdot 0.01 = 1.29$. So $\frac{1}{2}\overline{L}(c^2) = 0.645$ could be interpreted as the average length the code c^2 requires per letter of the alphabet $\mathcal{X}$. When we encode blocks of n letters we are interested in the behaviour of

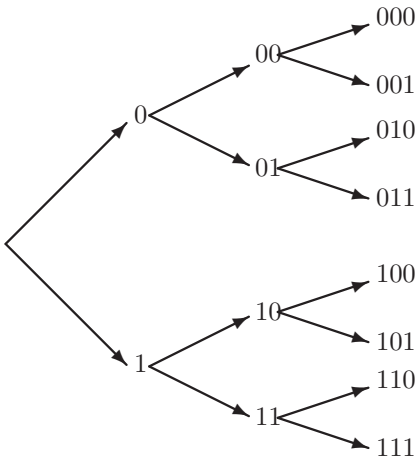


Figure 3.2. Example of a code tree.

$$L(n,P) = \min_{cUDC} \frac{1}{n} \sum_{(x_1 \ldots x_n) \in \mathcal{X}^n} P^n(x_1 \ldots x_n) L(x_1 \ldots x_n) \ .$$

It follows from the Noiseless Coding Theorem, which is stated in the next section, that $\lim_{n \longrightarrow \infty} L(n,P) = H(P)$ the entropy of the source $(\mathcal{X}, P)$.

In our example for the English language we have $H(P) \approx 4.19$. So the code presented above, where only single letters are encoded, is already nearly optimal in respect of $L(n,P)$. Further compression is possible, if we consider the dependencies between the letters.

3.1.3. Kraft's inequality and noiseless coding theorem

We shall now introduce a necessary and sufficient condition for the existence of a prefix code with prescribed word lengths $L(1), \ldots, L(m)$.

Theorem 3.1 (Kraft's inequality). *Let $\mathcal{X} = \{1, \ldots, m\}$. A uniquely decipherable code $c : \mathcal{X} \longrightarrow \{0,1\}^*$ with word lengths $L(1), \ldots, L(m)$ exists, if and only if*

$$\sum_{x \in \mathcal{X}} 2^{-L(x)} \leq 1 \ .$$

Proof The central idea is to interpret the codewords as nodes of a rooted binary tree with depth $T = \max_{x \in \mathcal{X}} \{L(x)\}$. The tree is required to be complete (every path from the root to a leaf has length T) and regular (every inner node has outdegree 2). The example in Figure 3.2 for $T = 3$ may serve as an illustration.

So the nodes with distance n from the root are labeled with the words $x^n \in \{0,1\}^n$. The upper successor of $x_1 x_2 \ldots x_n$ is labeled $x_1 x_2 \ldots x_n 0$, its lower successor is labeled $x_1 x_2 \ldots x_n 1$.

The ***shadow*** of a node labeled by $x_1x_2\ldots x_n$ is the set of all the leaves which are labeled by a word (of length T) beginning with $x_1x_2\ldots x_n$. In other words, the shadow of $x_1\ldots x_n$ consists of the leaves labeled by a sequence with prefix $x_1\ldots x_n$. In our example $\{000, 001, 010, 011\}$ is the shadow of the node labeled by 0.

Now suppose we are given positive integers $L(1), \ldots, L(m)$. We further assume that $L(1) \leq L(2) \leq \cdots \leq L(m)$. As first codeword $c(1) = \underbrace{00\ldots0}_{L(1)}$ is chosen. Since $\sum_{x\in\mathcal{X}} 2^{T-L(x)} \leq 2^T$, we have $2^{T-L(1)} < 2^T$ (otherwise only one letter has to be encoded). Hence there are left some nodes on the T-th level, which are not in the shadow of $c(1)$. We pick the first of these remaining nodes and go back $T-L(2)$ steps in direction to the root. Since $L(2) \geq L(1)$ we shall find a node labeled by a sequence of $L(2)$ bits, which is not a prefix of $c(1)$. So we can choose this sequence as $c(2)$. Now again, either $m = 2$, and we are ready, or by the hypothesis $2^{T-L(1)} + 2^{T-L(2)} < 2^T$ and we can find a node on the T-th level, which is not contained in the shadows of $c(1)$ and $c(2)$. We find the next codeword as shown above. The process can be continued until all codewords are assigned.

Conversely, observe that $\sum_{x\in\mathcal{X}} 2^{-L(x)} = \sum_{j=1}^{T} w_j\ 2^{-j}$, where w_j is the number of codewords with length j in the uniquely decipherable prefix code and T again denotes the maximal word length.

The s-th power of this term can be expanded as

$$\left(\sum_{j=1}^{T} w_j\ 2^{-j}\right)^s = \sum_{k=s}^{T\cdot s} N_k\ 2^{-k}\ .$$

Here $N_k = \sum_{i_1+\cdots+i_s=k} w_{i_1}\ldots w_{i_s}$ is the total number of messages whose coded representation is of length k.

Since the code is uniquely decipherable, to every sequence of k letters corresponds at most one possible message. Hence $N_k \leq 2^k$ and $\sum_{k=s}^{T\cdot s} N_k\ 2^{-k} \leq \sum_{k=s}^{T\cdot s} 1 = T\cdot s - s + 1 \leq T\cdot s$. Taking s–th root this yields $\sum_{j=1}^{T} w_j\ 2^{-j} \leq (T\cdot s)^{\frac{1}{s}}$.

Since this inequality holds for any s and $\lim_{s\longrightarrow\infty} (T\cdot s)^{\frac{1}{s}} = 1$, we have the desired result

$$\sum_{j=1}^{T} w_j\ 2^{-j} = \sum_{x\in\mathcal{X}} 2^{-L(x)} \leq 1\ .$$

■

Theorem 3.2 (Noiseless Coding Theorem). *For a source $(\mathcal{X}, P)$, $\mathcal{X} = \{1, \ldots, m\}$ it is always possible to find a uniquely decipherable code $c : \mathcal{X} \longrightarrow \{0,1\}^*$ with average length*

$$H(P) \leq \overline{L}(c) < H(P) + 1\ .$$

Proof Let $L(1), \ldots, L(m)$ denote the codeword lengths of an optimal uniquely decipherable code. Now we define a probability distribution Q by $Q(x) = \frac{2^{-L(x)}}{r}$ for $x = 1, \ldots, m$, where $r = \sum_{x=1}^{m} 2^{-L(x)}$. By Kraft's inequality $r \leq 1$.

For two probability distributions P and Q on $\mathcal{X}$ the ***I-divergence*** $D(P||Q)$ is defined by

$$D(P||Q) = \sum_{x \in \mathcal{X}} P(x) \lg \frac{P(x)}{Q(x)} \ .$$

I-divergence is a good measure for the distance of two probability distributions. Especially, always the I-divergence $D(P||Q) \geq 0$. So for any probability distribution P

$$D(P||Q) = -H(P) - \sum_{x \in \mathcal{X}} P(x) \cdot \lg\bigl(2^{-L(x)} \cdot r^{-1}\bigr) \geq 0 \ .$$

From this it follows that

$$\begin{aligned} H(P) &\leq -\sum_{x \in \mathcal{X}} P(x) \cdot \lg\ 2^{-L(x)} \cdot r^{-1} \\ &= \sum_{x \in \mathcal{X}} P(x) \cdot L(x) - \sum_{x \in \mathcal{X}} P(x) \cdot \lg r^{-1} = L_{\min}(P) + \lg r. \end{aligned}$$

Since $r \leq 1$, $\lg r \leq 0$ and hence $L_{\min}(P) \geq H(P)$.

In order to prove the right-hand side of the Noiseless Coding Theorem for $x = 1, \ldots, m$ we define $L(x) = \lceil -\lg P(x) \rceil$. Observe that $-\lg P(x) \leq L(x) < -\lg P(x) + 1$ and hence $P(x) \geq 2^{-L(x)}$.

So $1 = \sum_{x \in \mathcal{X}} P(x) \geq \sum_{x \in \mathcal{X}} 2^{-L(x)}$ and from Kraft's Inequality we know that there exists a uniquely decipherable code with word lengths $L(1), \ldots, L(m)$. This code has average length

$$\sum_{x \in \mathcal{X}} P(x) \cdot L'(x) < \sum_{x \in \mathcal{X}} P(x)(-\lg P(x) + 1) = H(P) + 1 \ .$$

■

3.1.4. Shannon-Fano-Elias-codes and the Shannon-Fano-algorithm

In the proof of the Noiseless Coding Theorem it was explicitly shown how to construct a prefix code c to a given probability distribution $P = (P(1), \ldots, P(a))$. The idea was to assign to each x a codeword of length $L(x)$ by choosing an appropriate vertex in the tree introduced. However, this procedure does not always yield an optimal code. If e.g. we are given the probability distribution $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, we would encode $1 \longrightarrow 00$, $2 \longrightarrow 01$, $3 \longrightarrow 10$ and thus achieve an average codeword length 2. But the code with $1 \longrightarrow 00$, $2 \longrightarrow 01$, $3 \longrightarrow 1$ has only average length $\frac{5}{3}$.

Shannon gave an explicit procedure for obtaining codes with codeword lengths $\lceil \lg \frac{1}{P(x)} \rceil$ using the binary representation of cumulative probabilities (Shannon remarked this procedure was originally due to Fano). The elements of the source are ordered according to increasing probabilities $P(1) \geq P(2) \geq \cdots \geq P(m)$. Then the

x	$P(x)$	$Q(x)$	$\overline{Q}(x)$	$\lceil \lg \frac{1}{P(x)} \rceil$	$c_S(x)$	$c_{SFE}(x)$
1	0.25	0	0.125	2	00	001
2	0.2	0.25	0.35	3	010	0101
3	0.11	0.45	0.505	4	0111	10001
4	0.11	0.56	0.615	4	1000	10100
5	0.11	0.67	0.725	4	1010	10111
6	0.11	0.78	0.835	4	1100	11010
7	0.11	0.89	0.945	4	1110	11110
				$\overline{L}$	3.3	4.3

Figure 3.3. Example of Shannon code and Shannon-Fano-Elias-code.

x	$P(x)$	$c(x)$	$L(x)$
1	0.25	00	2
2	0.2	01	2
3	0.11	100	3
4	0.11	101	3
5	0.11	110	3
6	0.11	1110	4
7	0.11	1111	4
		$\overline{L}(c)$	2.77

Figure 3.4. Example of the Shannon-Fano-algorithm.

codeword $c_S(x)$ consists of the first $\lceil \lg \frac{1}{P(x)} \rceil$ bits of the binary expansion of the sum $Q(x) = \sum_{j<x} P(j)$.

This procedure was further developed by Elias. The elements of the source now may occur in any order. The ***Shannon-Fano-Elias-code*** has as codewords $c_{SFE}(x)$ the first $\lceil \lg \frac{1}{P(x)} \rceil + 1$ bits of the binary expansion of the sum $\overline{Q}(x) = \sum_{j<x} P(j) + \frac{1}{2}P(x)$.

We shall illustrate these procedures with the example in Figure 3.3.

A more efficient procedure is also due to Shannon and Fano. The ***Shannon-Fano-algorithm*** will be illustrated by the same example in Figure 3.4.

The messages are first written in order of nonincreasing probabilities. Then the message set is partitioned into two most equiprobable subsets X_0 and X_1. A 0 is assigned to each message contained in one subset and a 1 to each of the remaining messages. The same procedure is repeated for subsets of X_0 and X_1; that is, X_0 will be partitioned into two subsets X_{00} and X_{01}. Now the code word corresponding to a message contained in X_{00} will start with 00 and that corresponding to a message in X_{01} will begin with 01. This procedure is continued until each subset contains only one message.

However, this algorithm neither yields an optimal code in general, since the prefix code $1 \longrightarrow 01$, $2 \longrightarrow 000$, $3 \longrightarrow 001$, $4 \longrightarrow 110$, $5 \longrightarrow 111$, $6 \longrightarrow 100$, $7 \longrightarrow 101$ has average length 2.75.

p_1	0.25	p_1	0.25	p_1	0.25	P_{23}	0.31	p_{4567}	0.44	p_{123}	0.56
p_2	0.2	p_{67}	0.22	p_{67}	0.22	p_1	0.25	p_{23}	0.31	p_{4567}	0.44
p_3	0.11	p_2	0.2	p_{45}	0.22	p_{67}	0.22	p_1	0.25		
p_4	0.11	p_3	0.11	p_2	0.2	p_{45}	0.22				
p_5	0.11	p_4	0.11	p_3	0.11						
P_6	0.11	p_5	0.11								
p_7	0.11										

C_{123}	0	c_{4567}	1	c_{23}	00	c_1	01	c_1	01	c_1	01
c_{4567}	1	c_{23}	00	c_1	01	c_{67}	10	c_{67}	10	c_2	000
		c_1	01	c_{67}	10	c_{45}	11	c_2	000	c_3	001
				c_{45}	11	c_2	000	c_3	001	c_4	110
						c_3	001	c_4	110	c_5	111
								c_5	111	c_6	100
										c_7	101

Figure 3.5. Example of a Huffman code.

3.1.5. The Huffman coding algorithm

The ***Huffman coding algorithm*** is a recursive procedure, which we shall illustrate with the same example as for the Shannon-Fano-algorithm in Figure 3.5 with $p_x = P(x)$ and $c_x = c(x)$. The source is successively reduced by one element. In each reduction step we add up the two smallest probabilities and insert their sum $P(m)+P(m-1)$ in the increasingly ordered sequence $P(1) \geq \cdots \geq P(m-2)$, thus obtaining a new probability distribution P' with $P'(1) \geq \cdots \geq P'(m-1)$. Finally we arrive at a source with two elements ordered according to their probabilities. The first element is assigned a 0, the second element a 1. Now we again "blow up" the source until the original source is restored. In each step $c(m-1)$ and $c(m)$ are obtained by appending 0 or 1, respectively, to the codeword corresponding to $P(m)+P(m-1)$.

Correctness

The following theorem demonstrates that the Huffman coding algorithm always yields a prefix code optimal with respect to the average codeword length.

Theorem 3.3 *We are given a source $(\mathcal{X}, P)$, where $\mathcal{X} = \{1, \ldots, m\}$ and the probabilities are ordered non–increasingly $P(1) \geq P(2) \geq \cdots \geq P(m)$. A new probability distribution is defined by*

$$P' = \big(P(1), \ldots, P(m-2), P(m-1) + P(m)\big) \ .$$

Let $c' = \big(c'(1), c'(2), \ldots, c'(m-1)\big)$ be an optimal prefix code for P'. Now we define a code c for the distribution P by

$$\begin{aligned} c(x) &= c'(x) \text{ for } x = 1, \ldots, m-2 , \\ c(m-1) &= c'(m-1)0 , \\ c(m) &= c'(m-1)1 . \end{aligned}$$

Then c is an optimal prefix code for P and $L_{\text{opt}}(P) - L_{\text{opt}}(P') = p(m-1) + p(m)$, where $L_{\text{opt}}(P)$ denotes the length of an optimal prefix code for probability distribution P

Proof For a probability distribution P on $\mathcal{X} = \{1, \ldots, m\}$ with $P(1) \geq P(2) \geq \cdots \geq P(m)$ there exists an optimal prefix code c with

i) $L(1) \leq L(2) \leq \cdots \leq L(m)$

ii) $L(m-1) = L(m)$

iii) $c(m-1)$ and $c(m)$ differ exactly in the last position.

This holds, since:

i) Assume that there are $x, y \in \mathcal{X}$ with $P(x) \geq P(y)$ and $L(x) > L(y)$. Then the code d obtained by interchanging the codewords $c(x)$ and $c(y)$ has average length $\overline{L}(d) \leq \overline{L}(c)$, since

$$\begin{aligned} \overline{L}(d) - \overline{L}(c) &= P(x) \cdot L(y) + P(y) \cdot L(x) - P(x) \cdot L(x) - P(y) \cdot L(y) \\ &= (P(x) - P(y)) \cdot (L(y) - L(x)) \leq 0 \end{aligned}$$

ii) Assume we are given a code with $L(1) \leq \cdots \leq L(m-1) < L(m)$. Because of the prefix property we may drop the last $L(m) - L(m-1)$ bits and thus obtain a new code with $L(m) = L(m-1)$.

iii) If no two codewords of maximal length agree in all places but the last, then we may drop the last digit of all such codewords to obtain a better code.

Now we are ready to prove the statement from the theorem. From the definition of c and c' we have

$$L_{\text{opt}}(P) \leq \overline{L}(c) = \overline{L}(c') + p(m-1) + p(m) .$$

Now let d be an optimal prefix code with the properties ii) and iii) from the preceding lemma. We define a prefix code d' for

$$P' = (P(1), \ldots, P(m-2), P(m-1) + P(m))$$

by $d'(x) = d(x)$ for $x = 1, \ldots, m-2$ and $d'(m-1)$ is obtained by dropping the last bit of $d(m-1)$ or $d(m)$.

Now

$$\begin{aligned} L_{\text{opt}}(P) &= \overline{L}(d) = \overline{L}(d') + P(m-1) + P(m) \\ &\geq L_{\text{opt}}(P') + P(m-1) + P(m) \end{aligned}$$

and hence $L_{\text{opt}}(P) - L_{\text{opt}}(P') = P(m-1) + P(m)$, since $\overline{L}(c') = L_{\text{opt}}(P')$. ∎

Analysis

If m denotes the size of the source alphabet, the Huffman coding algorithm needs

$m-1$ additions and $m-1$ code modifications (appending 0 or 1). Further we need $m-1$ insertions, such that the total complexity can be roughly estimated to be $O(m \lg m)$. However, observe that with the Noiseless Coding Theorem, the quality of the compression rate can only be improved by jointly encoding blocks of, say, k letters, which would result in a Huffman code for the source $\mathcal{X}^k$ of size m^k. So, the price for better compression is a rather drastic increase in complexity. Further, the codewords for all m^k letters have to be stored. Encoding a sequence of n letters can since be done in $O(\frac{n}{k} \cdot (m^k \lg m^k))$ steps.

Exercises

3.1-1 Show that the code $c : \{a, b\} \longrightarrow \{0,1\}^*$ with $c(a) = 0$ and $c(b) = \underbrace{0 \ldots 01}_{n}$ is uniquely decipherable but not instantaneous for any $n > 0$.

3.1-2 Compute the entropy of the source $(\mathcal{X}, P)$, with $\mathcal{X} = \{1, 2\}$ and $P = (0.8, 0, 2)$.

3.1-3 Find the Huffman-codes and the Shannon-Fano-codes for the sources $(\mathcal{X}^n, P^n)$ with $(\mathcal{X}, P)$ as in the previous exercise for $n = 1, 2, 3$ and calculate their average codeword lengths.

3.1-4 Show that always $0 \leq H(P) \leq \lg |\mathcal{X}|$.

3.1-5 Show that the redundancy $\rho(c) = \overline{L}(c) - H(P)$ of a prefix code c for a source with probability distribution P can be expressed as a special I–divergence.

3.1-6 Show that the I-divergence $D(P||Q) \geq 0$ for all probability distributions P and Q over some alphabet $\mathcal{X}$ with equality exactly if $P = Q$ but that the I-divergence is not a metric.

3.2. Arithmetic coding and modelling

In statistical coding techniques as Shannon-Fano- or Huffman-coding the probability distribution of the source is modelled as accurately as possible and then the words are encoded such that a higher probability results in a shorter codeword length.

We know that Huffman-codes are optimal with respect to the average codeword length. However, the entropy is approached by increasing the block length. On the other hand, for long blocks of source symbols, Huffman-coding is a rather complex procedure, since it requires the calculation of the probabilities of all sequences of the given block length and the construction of the corresponding complete code.

For compression techniques based on statistical methods often ***arithmetic coding*** is preferred. Arithmetic coding is a straightforward extension of the Shannon-Fano-Elias-code. The idea is to represent a probability by an interval. In order to do so, the probabilities have to be calculated very accurately. This process is denoted as ***modelling*** of the source . So statistical compression techniques consist of two stages: modelling and coding. As just mentioned, coding is usually done by arithmetic coding. The different algorithms like, for instance, DCM (Discrete Markov Coding) and PPM (Prediction by Partial Matching) vary in the way of modelling the source. We are going to present the context-tree weighting method, a transparent algorithm for the estimation of block probabilities due to Willems, Shtarkov, and Tjalkens,

which also allows a straightforward analysis of the efficiency.

3.2.1. Arithmetic coding

The idea behind arithmetic coding is to represent a message $x^n = (x_1 \dots x_n)$ by interval $I(x^n) = [Q^n(x^n), Q^n(x^n)+P^n(x^n))$, where $Q^n(x^n) = \sum_{y^n<x^n} P^n(y^n)$ is the sum of the probabilities of those sequences which are smaller than x^n in lexicographic order.

A codeword $c(x^n)$ assigned to message x^n also corresponds to an interval. Namely, we identify codeword $c = c(x^n)$ of length $L = L(x^n)$ with interval $J(c) = [bin(c), bin(c) + 2^{-L})$, where $bin(c)$ is the binary expansion of the nominator in the fraction $\frac{c}{2^L}$. The special choice of codeword $c(x^n)$ will be obtained from $P^n(x^n)$ and $Q^n(x^n)$ as follows:

$$L(x^n) = \lceil \lg \frac{1}{P^n(x^n)} \rceil + 1, \qquad bin(c) = \lceil Q^n(x^n) \cdot 2^{L(x^n)} \rceil \ .$$

So message x^n is encoded by a codeword $c(x^n)$, whose interval $J(x^n)$ is inside interval $I(x^n)$.

Let us illustrate arithmetic coding by the following example of a discrete memoryless source with $P(1) = 0.1$ and $n = 2$.

x^n	$P^n(x^n)$	$Q^n(x^n)$	$L(x^n)$	$c(x^n)$
00	0.81	0.00	2	00
01	0.09	0.81	5	11010
10	0.09	0.90	5	11101
11	0.01	0.99	8	11111110 .

At first glance it may seem that this code is much worse than the Huffman code for the same source with codeword lengths $(1, 2, 3, 3)$ we found previously. On the other hand, it can be shown that arithmetic coding always achieves an average codeword length $\overline{L}(c) < H(P^n) + 2$, which is only two bits apart from the lower bound in the noiseless coding theorem. Huffman coding would usually yield an even better code. However, this "negligible" loss in compression rate is compensated by several advantages. The codeword is directly computed from the source sequence, which means that we do not have to store the code as in the case of Huffman coding. Further, the relevant source models allow to easily compute $P^n(x_1x_2 \dots x_{n-1}x_n)$ and $Q^n(x_1x_2 \dots x_{n-1}x_n)$ from $P^{n-1}(x_1x_2 \dots x_{n-1})$, usually by multiplication by $P(x_n)$. This means that the sequence to be encoded can be parsed sequentially bit by bit, unlike in Huffman coding, where we would have to encode blockwise.

Encoding: The basic algorithm for encoding a sequence $(x_1 \dots x_n)$ by arithmetic coding works as follows. We assume that $P^n(x_1 \dots x_n) = P_1(x_1) \cdot P_2(x_2) \cdots P_n(x_n)$, (in the case $P_i = P$ for all i the discrete memoryless source arises, but in the section on modelling more complicated formulae come into play) and hence $Q_i(x_i) = \sum_{y<x_i} P_i(x_i)$

Starting with $B_0 = 0$ and $A_0 = 1$ the first i letters of the text to be compressed determine the ***current interval*** $[B_i, B_i + A_i)$. These current intervals are successively refined via the recursions

$$B_{i+1} = B_i + A_i \cdot Q_i(x_i), \quad A_{i+1} = A_i \cdot P_i(x_i) \ .$$

$A_i \cdot P_i(x)$ is usually denoted as augend. The final interval $[B_n, B_n + A_n) = [Q^n(x^n), Q^n(x^n) + P^n(x^n))$ will then be encoded by interval $J(x^n)$ as described above. So the algorithm looks as follows.

ARITHMETIC-ENCODER(x)

```
1  B ← 0
2  A ← 1
3  for i ← 1 to n
4       do B ← B + A · Q_i(x[i])
5          A ← A · P_i(x[i])
6  L ← ⌈lg 1/A⌉ + 1
7  c ← ⌈B · 2^L⌉
8  return c
```

We shall illustrate the encoding procedure by the following example from the literature. Let the discrete, memoryless source $(\mathcal{X}, P)$ be given with ternary alphabet $\mathcal{X} = \{1, 2, 3\}$ and $P(1) = 0.4$, $P(2) = 0.5$, $P(3) = 0.1$. The sequence $x^4 = (2, 2, 2, 3)$ has to be encoded. Observe that $P_i = P$ and $Q_i = Q$ for all $i = 1, 2, 3, 4$. Further $Q(1) = 0$, $Q(2) = P(1) = 0.4$, and $Q(3) = P(1) + P(2) = 0.9$.

The above algorithm yields

i	B_i	A_i
0	0	1
1	$B_0 + A_0 \cdot Q(2) = 0.4$	$A_0 \cdot P(2) = 0.5$
2	$B_1 + A_1 \cdot Q(2) = 0.6$	$A_1 \cdot P(2) = 0.25$
3	$B_2 + A_2 \cdot Q(2) = 0.7$	$A_2 \cdot P(2) = 0.125$
4	$B_3 + A_3 \cdot Q(3) = 0.8125$	$A_3 \cdot P(3) = 0.0125$.

Hence $Q(2, 2, 2, 3) = B_4 = 0.8125$ and $P(2, 2, 2, 3) = A_4 = 0.0125$. From this can be calculated that $L = \lceil \lg \frac{1}{A} \rceil + 1 = 8$ and finally $\lceil B \cdot 2^L \rceil = \lceil 0.8125 \cdot 256 \rceil = 208$ whose binary representation is codeword $c(2, 2, 2, 3) = 11010000$.

Decoding: Decoding is very similar to encoding. The decoder recursively "undoes" the encoder's recursion. We divide the interval $[0, 1)$ into subintervals with bounds defined by Q_i. Then we find the interval in which codeword c can be found. This interval determines the next symbol. Then we subtract $Q_i(x_i)$ and rescale by multiplication by $\frac{1}{P_i(x_i)}$.

ARITHMETIC-DECODER(c)

```
1  for i ← 1 to n
2      do j ← 1
3         while (c < Q_i(j))
4               do j ← j + 1
5                  x[i] ← j − 1
6                  c ← (c − Q_i(x[i]))/P_i(x[i])
7  return x
```

Observe that when the decoder only receives codeword c he does not know when the decoding procedure terminates. For instance $c = 0$ can be the codeword for $x^1 = (1)$, $x^2 = (1,1)$, $x^3 = (1,1,1)$, etc. In the above pseudocode it is implicit that the number n of symbols has also been transmitted to the decoder, in which case it is clear what the last letter to be encoded was. Another possibility would be to provide a special end-of-file (EOF)-symbol with a small probability, which is known to both the encoder and the decoder. When the decoder sees this symbol, he stops decoding. In this case line 1 would be replaced by

1 **while** $(x[i] \neq \text{EOF})$

(and i would have to be increased). In our above example, the decoder would receive the codeword 11010000, the binary expansion of 0.8125 up to $L = 8$ bits. This number falls in the interval $[0.4, 0.9)$ which belongs to the letter 2, hence the first letter $x_1 = 2$. Then he calculates $(0.8075 - Q(2))\frac{1}{P(2)} = (0.815 - 0.4) \cdot 2 = 0.83$. Again this number is in the interval $[0.4, 0.9)$ and the second letter is $x_2 = 2$. In order to determine x_3 the calculation $(0.83 - Q(2))\frac{1}{P(2)} = (0.83 - 0.4) \cdot 2 = 0.86$ must be performed. Again $0.86 \in [0.4, 0.9)$ such that also $x_3 = 2$. Finally $(0.86 - Q(2))\frac{1}{P(2)} = (0.86 - 0.4) \cdot 2 = 0.92$. Since $0.92 \in [0.9, 1)$, the last letter of the sequence must be $x_4 = 3$.

Correctness

Recall that message x^n is encoded by a codeword $c(x^n)$, whose interval $J(x^n)$ is inside interval $I(x^n)$. This follows from $\lceil Q^n(x^n) \cdot 2^{L(x^n)} \rceil 2^{-L(x^n)} + 2^{-L(x^n)} < Q^n(x^n) + 2^{1-L(x^n)} = Q^n(x^n) + 2^{-\lceil \lg \frac{1}{P^n(x^n)} \rceil} \leq Q^n(x^n) + P^n(x^n)$.

Obviously a prefix code is obtained, since a codeword can only be a prefix of another one, if their corresponding intervals overlap – and the intervals $J(x^n) \subset I(x^n)$ are obviously disjoint for different n-s.

Further, we mentioned already that arithmetic coding compresses down to the entropy up to two bits. This is because for every sequence x^n it is $L(x^n) < \lg \frac{1}{P^n(x^n)} + 2$. It can also be shown that the additional transmission of block length n or the introduction of the EOF symbol only results in a negligible loss of compression.

However, the basic algorithms we presented are not useful in order to compress longer files, since with increasing block length n the intervals are getting smaller and smaller, such that rounding errors will be unavoidable. We shall present a technique to overcome this problem in the following.

Analysis

The basic algorithm for arithmetic coding is linear in the length n of the sequence to be encoded. Usually, arithmetic coding is compared to Huffman coding.

In contrast to Huffman coding, we do not have to store the whole code, but can obtain the codeword directly from the corresponding interval. However, for a discrete memoryless source, where the probability distribution $P_i = P$ is the same for all letters, this is not such a big advantage, since the Huffman code will be the same for all letters (or blocks of k letters) and hence has to be computed only once. Huffman coding, on the other hand, does not use any multiplications which slow down arithmetic coding.

For the adaptive case, in which the P_i's may change for different letters x_i to be encoded, a new Huffman code would have to be calculated for each new letter. In this case, usually arithmetic coding is preferred. We shall investigate such situations in the section on modelling.

For implementations in practice floating point arithmetic is avoided. Instead, the subdivision of the interval $[0, 1)$ is represented by a subdivision of the integer range $0, \ldots, M$, say, with proportions according to the source probabilities. Now integer arithmetic can be applied, which is faster and more precise.

Precision problem

In the basic algorithms for arithmetic encoding and decoding the shrinking of the current interval would require the use of high precision arithmetic for longer sequences. Further, no bit of the codeword is produced until the complete sequence x^n has been read in. This can be overcome by coding each bit as soon as it is known and then double the length of the current interval $[LO, HI)$, say, so that this expansion represents only the unknown part of the interval. This is the case when the leading bits of the lower and upper bound are the same, i. e. the interval is completely contained either in $[0, \frac{1}{2})$ or in $[\frac{1}{2}, 1)$. The following expansion rules guarantee that the current interval does not become too small.

Case 1 ($[LO, HI) \in [0, \frac{1}{2})$): $LO \leftarrow 2 \cdot Lo$, $HI \leftarrow 2 \cdot HI$.

Case 2 ($[LO, HI) \in [\frac{1}{2}, 1)$): $LO \leftarrow 2 \cdot LO - 1$, $HI \leftarrow 2 \cdot HI - 1$.

Case 3 ($\frac{1}{4} \leq LO < \frac{1}{2} \leq HI < \frac{3}{4}$): $LO \leftarrow 2 \cdot LO - \frac{1}{2}$, $HI \leftarrow 2 \cdot HI - \frac{1}{2}$.

The last case called ***underflow*** (or follow) prevents the interval from shrinking too much when the bounds are close to $\frac{1}{2}$. Observe that if the current interval is contained in $[\frac{1}{4}, \frac{3}{4})$ with $LO < \frac{1}{2} \leq HI$, we do not know the next output bit, but we do know that whatever it is, the following bit will have the opposite value. However, in contrast to the other cases we cannot continue encoding here, but have to wait (remain in the underflow state and adjust a counter $underflowcount$ to the number of subsequent underflows, i. e. $underflowcount \leftarrow underflowcount + 1$) until the current interval falls into either $[0, \frac{1}{2})$ or $[\frac{1}{2}, 1)$. In this case we encode the leading bit of this interval – 0 for $[0, \frac{1}{2})$ and 1 for $[\frac{1}{2}, 1)$ – followed by $underflowcount$ many inverse bits and then set $underflowcount = 0$. The procedure stops, when all letters are read in and the current interval does not allow any further expansion.

ARITHMETIC-PRECISION-ENCODER(x)

```
1  LO ← 0
2  HI ← 1
3  A ← 1
4  underflowcount ← 0
5  for i ← 1 to n
6      do LO ← LO + Q_i(x[i]) · A
7         A ← P_i(x[i])
8         HI ← LO + A
9         while HI − LO < 1/2 AND NOT (LO < 1/4 AND HI ≥ 1/2)
10                do if HI < 1/2
11                      then c ← c||0, underflowcount many 1s
12                           underflowcount ← 0
13                           LO ← 2· LO
14                           HI ← 2· HI
15                      else if LO ≥ 1/2
16                             then c ← c||1, underflowcount many 0s
17                                  underflowcount ← 0
18                                  LO ← 2 · LO − 1
19                                  HI ← 2 · HI − 1
20                             else if LO ≥ 1/4 AND HI < 3/4
21                            then underflowcount ← underflowcount +1
22                                 LO ← 2 · LO − 1/2
23                                 HI ← 2 · HI − 1/2
24 if underflowcount > 0
25    then c ← c||0, underflowcount many 1s)
26 return c
```

We shall illustrate the encoding algorithm in Figure 3.6 by our example – the encoding of the message $(2, 2, 2, 3)$ with alphabet $\mathcal{X} = \{1, 2, 3\}$ and probability distribution $P = (0.4, 0.5, 0.1)$. An underflow occurs in the sixth row: we keep track of the underflow state and later encode the inverse of the next bit, here this inverse bit is the 0 in the ninth row. The encoded string is 1101000.

Precision-decoding involves the consideration of a third variable besides the interval bounds LO and HI.

3.2.2. Modelling

Modelling of memoryless sources with the Krichevsky-Trofimov-Estimator In this section we shall only consider binary sequences $x^n \in \{0, 1\}^n$ to be compressed by an arithmetic coder. Further, we shortly write $P(x^n)$ instead of $P^n(x^n)$ in order to allow further subscripts and superscripts for the description of the special situation. P_e will denote estimated probabilities, P_w weighted probabilities, and P^s probabilities assigned to a special context s.

The application of arithmetic coding is quite appropriate if the probability distribution of the source is such that $P(x_1 x_2 \dots x_{n-1} x_n)$ can easily be calculated from

Current Interval	Action	Subintervals 1	2	3	Input
$[0.00, 1.00)$	subdivide	$[0.00, 0.40)$	$[0.40, 0.90)$	$[0.90, 1.00)$	2
$[0.40, 0.90)$	subdivide	$[0.40, 0.60)$	$[0.60, 0.85)$	$[0.85, 0.90)$	2
$[0.60, 0.85)$	encode 1				
	expand $[\frac{1}{2}, 1)$				
$[0.20, 0.70)$	subdivide	$[0.20, 0.40)$	$[0.40, 0.65)$	$[0.65, 0.70)$	2
$[0.40, 0.65)$	underflow				
	expand $[\frac{1}{4}, \frac{3}{4})$				
$[0.30, 0.80)$	subdivide	$[0.30, 0.50)$	$[0.50, 0.75)$	$[0.75, 0.80)$	3
$[0.75, 0.80)$	encode 10				
	expand $[\frac{1}{2}, 1)$				
$[0.50, 0.60)$	encode 1				
	expand $[\frac{1}{2}, 1)$				
$[0.00, 0.20)$	encode 0				
	expand $[0, \frac{1}{2})$				
$[0.00, 0.40)$	encode 0				
	expand $[0, \frac{1}{2})$				
$[0.00, 0.80)$	encode 0				

Figure 3.6. Example of arithmetic encoding with interval expansion.

$P(x_1x_2 \dots x_{n-1})$. Obviously this is the case, when the source is discrete and memoryless, since then $P(x_1x_2 \dots x_{n-1}x_n) = P(x_1x_2 \dots x_{n-1}) \cdot P(x_n)$.

Even when the underlying parameter $\theta = P(1)$ of a binary, discrete memoryless source is not known, there is an efficient way due to Krichevsky and Trofimov to estimate the probabilities via

$$P(X_n = 1|x^{n-1}) = \frac{b + \frac{1}{2}}{a + b + 1} \ ,$$

where a and b denote the number of 0s and 1s, respectively, in the sequence $x^{n-1} = (x_1x_2 \dots x_{n-1})$. So given the sequence x^{n-1} with a many 0s and b many 1s, the probability that the next letter x_n will be a 1 is estimated as $\frac{b+\frac{1}{2}}{a+b+1}$. The estimated block probability of a sequence containing a zeros and b ones then is

$$P_e(a, b) = \frac{\frac{1}{2} \cdots (a - \frac{1}{2}) \frac{1}{2} \cdots (b - \frac{1}{2})}{1 \cdot 2 \cdots (a + b)}$$

with initial values $a = 0$ and $b = 0$ as in Figure 3.7, where the values of the ***Krichevsky-Trofimov estimator*** $P_e(a, b)$ for small (a, b) are listed.

Note that the summand $\frac{1}{2}$ in the nominator guarantees that the probability for the next letter to be a 1 is positive even when the symbol 1 did not occur in the sequence so far. In order to avoid infinite codeword length, this phenomenon has to be carefully taken into account when estimating the probability of the next letter in all approaches to estimate the parameters, when arithmetic coding is applied.

a	b	0	1	2	3	4	5
0		1	1/2	3/8	5/16	35/128	63/256
1		1/2	1/8	1/16	5/128	7/256	21/1024
2		3/8	1/16	3/128	3/256	7/1024	9/2048
3		5/16	5/128	3/256	5/1024	5/2048	45/32768

Figure 3.7. Table of the first values for the Krichevsky-Trofimov-estimator.

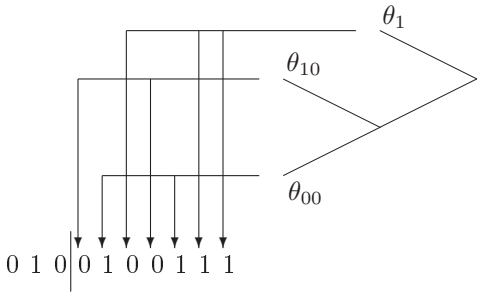


Figure 3.8. An example for a tree source.

Models with known context tree In most situations the source is not memoryless, i. e., the dependencies between the letters have to be considered. A suitable way to represent such dependencies is the use of a suffix tree, which we denote as ***context tree***. The context of symbol x_t is suffix s preceding x_t. To each context (or leaf in the suffix tree) s there corresponds a parameter $\theta_s = P(X_t = 1|s)$, which is the probability of the occurrence of a 1 when the last sequence of past source symbols is equal to context s (and hence $1 - \theta_s$ is the probability for a 0 in this case). We are distinguishing here between the model (the suffix tree) and the parameters (θ_s).

Example 3.1 Let $\mathcal{S} = \{00, 10, 1\}$ and $\theta_{00} = \frac{1}{2}$, $\theta_{10} = \frac{1}{3}$, and $\theta_1 = \frac{1}{5}$. The corresponding suffix tree jointly with the parsing process for a special sequence can be seen in Figure 3.8.

The actual probability of the sequence '0100111' given the past '...010' is $P^s(0100111|\ldots 010) = (1-\theta_{10})\theta_{00}(1-\theta_1)(1-\theta_{10})\theta_{00}\theta_1\theta_1 = \frac{2}{3}\cdot\frac{1}{2}\cdot\frac{4}{5}\cdot\frac{2}{3}\cdot\frac{1}{2}\cdot\frac{1}{5}\cdot\frac{1}{5} = \frac{4}{1075}$, since the first letter 0 is preceded by suffix 10, the second letter 1 is preceded by suffix 00, etc.

Suppose the model $\mathcal{S}$ is known, but not the parameters θ_s. The problem now is to find a good coding distribution for this case. The tree structure allows to easily determine which context precedes a particular symbol. All symbols having the same context (or suffix) $s \in \mathcal{S}$ form a memoryless source subsequence whose probability is determined by the unknown parameter θ_s. In our example these subsequences are '11' for θ_{00}, '00' for θ_{10} and '011' for θ_1. One uses the Krichevsky-Trofimov-estimator for this case. To each node s in the suffix tree, we count the numbers a_s of zeros and b_s of ones preceded by suffix s. For the children $0s$ and $1s$ of parent node s obviously

$a_{0s} + a_{1s} = a_s$ and $b_{0s} + b_{1s} = b_s$ must be satisfied.

In our example $(a_\lambda, b_\lambda) = (3,4)$ for the root λ, $(a_1, b_1) = (1,2)$, $(a_0, b_0) = (2,2)$ and $(a_{10}, b_{10}) = (2,0)$, $(a_{00}, b_{00}) = (0,2)$. Further $(a_{11}, b_{11}) = (0,1)$, $(a_{01}, b_{01}) = (1,1)$, $(a_{111}, b_{111}) = (0,0)$, $(a_{011}, b_{011}) = (0,1)$, $(a_{101}, b_{101}) = (0,0)$,$(a_{001}, b_{001}) = (1,1)$, $(a_{110}, b_{110}) = (0,0)$, $(a_{010}, b_{010}) = (2,0)$, $(a_{100}, b_{100}) = (0,2)$, and $(a_{000}, b_{000}) = (0,0)$. These last numbers are not relevant for our special source $\mathcal{S}$ but will be important later on, when the source model or the corresponding suffix tree, respectively, is not known in advance.

Example 3.2 Let $\mathcal{S} = \{00, 10, 1\}$ as in the previous example. Encoding a subsequence is done by successively updating the corresponding counters for a_s and b_s. For example, when we encode the sequence '0100111' given the past '...010' using the above suffix tree and Krichevsky–Trofimov–estimator we obtain

$$P_e^s(0100111|\ldots 010) = \frac{1}{2}\cdot\frac{1}{2}\cdot\frac{1}{2}\cdot\frac{3}{4}\cdot\frac{3}{4}\cdot\frac{1}{4}\cdot\frac{1}{2} = \frac{3}{8}\cdot\frac{3}{8}\cdot\frac{1}{16} = \frac{9}{1024}\ ,$$

where $\frac{3}{8}$, $\frac{3}{8}$ and $\frac{1}{16}$ are the probabilities of the subsequences '11', '00' and '011' in the context of the leaves. These subsequences are assumed to be memoryless.

The context-tree weighting method Suppose we have a good coding distribution P_1 for source 1 and another one, P_2, for source 2. We are looking for a good coding distribution for both sources. One possibility is to compute P_1 and P_2 and then 1 bit is needed to identify the best model which then will be used to compress the sequence. This method is called selecting. Another possibility is to employ the weighted distribution, which is

$$P_w(x^n) = \frac{P_1(x^n) + P_2(x^n)}{2}.$$

We shall present now the ***context-tree weighting algorithm*** . Under the assumption that a context tree is a full tree of depth D, only a_s and b_s, i. e. the number of zeros and ones in the subsequence of bits preceded by context s, are stored in each node s of the context tree.

Further, to each node s is assigned a weighted probability P_w^s which is recursively defined as

$$P_w^s = \begin{cases} \frac{P_e(a_s,b_s)+P_w^{0s}P_w^{1s}}{2} & \text{for } 0 \le L(s) < D\ , \\ P_e(a_s, b_s) & \text{for } L(s) = D\ , \end{cases}$$

where $L(s)$ describes the length of the (binary) string s and $P_e(a_s, b_s)$ is the estimated probability using the Krichevsky - Trofimov estimator.

Example 3.3 After encoding the sequence '0100111' given the past '...010' we obtain the context tree of depth 3 in Figure 3.9. The weighted probability $P_w\lambda = \frac{35}{4096}$ of the root node λ finally yields the coding probability corresponding to the parsed sequence.

Recall that for the application in arithmetic coding it is important that probabilities $P(x_1 \ldots x_{n-1}0)$ and $P(x_1 \ldots x_{n-1}1)$ can be efficiently calculated from

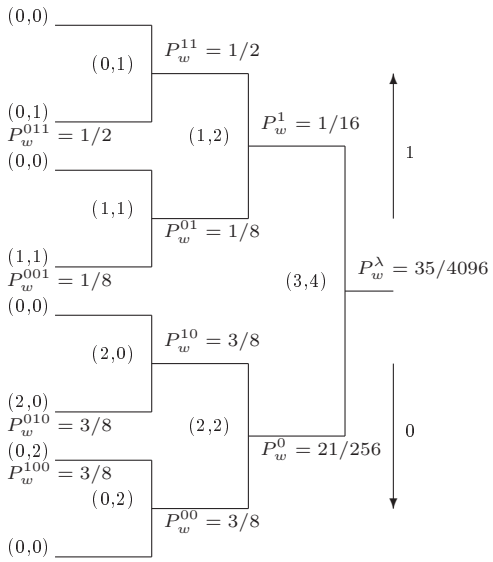


Figure 3.9. Weighted context tree for source sequence '0100111' with past ...010. The pair (a_s, b_s) denotes a_s zeros and b_s ones preceded by the corresponding context s. For the contexts $s = 111, 101, 110, 000$ it is $P_w^s = P_e(0,0) = 1$.

$P(x_1 \dots x_n)$. This is possible with the context-tree weighting method, since the weighted probabilities P_w^s only have to be updated, when s is changing. This just occurs for the contexts along the path from the root to the leaf in the context tree preceding the new symbol x_n—namely the $D+1$ contexts $x_{n-1}, \dots, x_{n-i}$ for $i = 1, \dots, D-1$ and the root λ. Along this path, $a_s = a_s + 1$ has to be performed, when $x_n = 0$, and $b_s = b_s + 1$ has to be performed, when $x_n = 1$, and the corresponding probabilities $P_e(a_s, b_s)$ and P_w^s have to be updated.

This suggests the following algorithm for updating the context tree $CT(x_1, \dots, x_{n-1}|x_{-D+1}, \dots x_0)$ when reading the next letter x_n. Recall that to each node of the tree we store the parameters (a_s, b_s), $P_e(a_s, b_s)$ and P_w^s. These parameters have to be updated in order to obtain $CT(x_1, \dots, x_n|x_{-D+1}, \dots x_0)$. We assume the convention that the ordered pair (x_{n-1}, x_n) denotes the root λ.

UPDATE-CONTEXT-TREE$(x_n, CT(x_1 \dots x_{n-1}|x_{-D+1} \dots x_0))$

1 $s \leftarrow (x_{n-1} \dots x_{n-D})$
2 **if** $x_n = 0$
3 **then** $P_w^s \leftarrow P_w^s \cdot \frac{a_s + 1/2}{a_s + b_s + 1}$
4 $a_s \leftarrow a_s + 1$
5 **else** $P_w^s \leftarrow P_w^s \cdot \frac{b_s + 1/2}{a_s + b_s + 1}$
6 $b_s \leftarrow b_s + 1$

```
7  for i ← 1 to D
8      do s ← (x_{n−1}, ..., x_{n−D+i})
9          if x_n = 0
10             then P_e(a_s, b_s) ← P_e(a_s, b_s) · (a_s+1/2)/(a_s+b_s+1)
11                  a_s ← a_s + 1
12             else P_e(a_s, b_s) ← P_e(a_s, b_s) · (a_s+1/2)/(a_s+b_s+1)
13                  b_s ← b_s + 1
14     P_w^s ← 1/2 · (P_e(a_s, b_s) + P_w^{0s} · P_w^{1s})
15 return P_w^s
```

The probability P_w^λ assigned to the root in the context tree will be used for the successive subdivisions in arithmetic coding. Initially, before reading x_1, the parameters in the context tree are $(a_s, b_s) = (0, 0)$, $P_e(a_s, b_s) = 1$, and $P_w^s = 1$ for all contexts s in the tree. In our example the updates given the past $(x_{-2}, x_{-1}, x_0) = (0, 1, 0)$ would yield the successive probabilities P_w^λ: $\frac{1}{2}$ for $x_1 = 0$, $\frac{9}{32}$ for $(x_1 x_2) = (01)$, $\frac{5}{64}$ for $(x_1 x_2 x_3) = (010)$, $\frac{13}{256}$ for $(x_1 x_2 x_3 x_4) = (0100)$, $\frac{27}{1024}$ for $(x_1 x_2 x_3 x_4) = (01001)$, $\frac{13}{1024}$ for $(x_1 x_2 x_3 x_4 x_5) = (010011)$, $\frac{13}{1024}$ for $(x_1 x_2 x_3 x_4 x_5 x_6) = (010011)$, and finally $\frac{35}{4096}$ for $(x_1 x_2 x_3 x_4 x_5 x_6 x_7) = (0100111)$.

Correctness

Recall that the quality of a code concerning its compression capability is measured with respect to the average codeword length. The average codeword length of the best code comes as close as possible to the entropy of the source. The difference between the average codeword length and the entropy is denoted as the ***redundancy*** $\overline{\rho}(c)$ of code c, hence

$$\overline{\rho}(c) = \overline{L}(c) - H(P) \ ,$$

which obviously is the weighted (by $P(x^n)$) sum of the individual redundancies

$$\rho(x^n) = L(x^n) - \lg \frac{1}{P(x^n)} \quad .$$

The individual redundancy $\rho(x^n|\mathcal{S})$ of sequences x^n given the (known) source $\mathcal{S}$ for all $\theta_s \in [0, 1]$ for $s \in \mathcal{S}$, $|\mathcal{S}| \leq n$ is bounded by

$$\rho(x^n|\mathcal{S}) \leq \frac{|\mathcal{S}|}{2} \lg \frac{n}{|\mathcal{S}|} + |\mathcal{S}| + 2 \ .$$

The individual redundancy $\rho(x^n|\mathcal{S})$ of sequences x^n using the context-tree weighting algorithm (and hence a complete tree of all possible contexts as model $\mathcal{S}$) is bounded by

$$\rho(x^n|\mathcal{S}) < 2|\mathcal{S}| - 1 + \frac{|\mathcal{S}|}{2} \lg \frac{n}{|\mathcal{S}|} + |\mathcal{S}| + 2 \ .$$

Comparing these two formulae, we see that the difference of the individual redundancies is $2|\mathcal{S}| - 1$ bits. This can be considered as the cost of not knowing the model, i.e. the model redundancy. So, the redundancy splits into the parameter redundancy, i. e. the cost of not knowing the parameter, and the model redundancy.

It can be shown that the expected redundancy behaviour of the context-tree weighting method achieves the asymptotic lower bound due to Rissanen who could demonstrate that about $\frac{1}{2}\lg n$ bits per parameter is the minimum possible expected redundancy for $n \longrightarrow \infty$.

Analysis

The computational complexity is proportional to the number of nodes that are visited when updating the tree, which is about $n(D+1)$. Therefore, the number of operations necessary for processing n symbols is linear in n. However, these operations are mainly multiplications with factors requiring high precision.

As for most modelling algorithms, the backlog of implementations in practice is the huge amount of memory. A complete tree of depth D has to be stored and updated. Only with increasing D the estimations of the probabilities are becoming more accurate and hence the average codeword length of an arithmetic code based on these estimations would become shorter. The size of the memory, however, depends exponentially on the depth of the tree.

We presented the context–tree weighting method only for binary sequences. Note that in this case the cumulative probability of a binary sequence $(x_1 \ldots x_n)$ can be calculated as

$$Q^n(x_1 x_2 \ldots x_{n-1} x_n) = \sum_{j=1,\ldots,n;x_j=1} P^j(x_1 x_2 \ldots x_{j-1} 0) \ .$$

For compression of sources with larger alphabets, for instance ASCII-files, we refer to the literature.

Exercises

3.2-1 Compute the arithmetic codes for the sources $(\mathcal{X}^n, P^n)$, $n = 1, 2, 3$ with $\mathcal{X} = \{1, 2\}$ and $P = (0.8, 0.2)$ and compare these codes with the corresponding Huffman-codes derived previously.

3.2-2 For the codes derived in the previous exercise compute the individual redundancies of each codeword and the redundancies of the codes.

3.2-3 Compute the estimated probabilities $P_e(a, b)$ for the sequence 0100110 and all its subsequences using the Krichevsky-Trofimov-estimator.

3.2-4 Compute all parameters (a_s, b_s) and the estimated probability P_e^s for the sequence 0100110 given the past 110, when the context tree $\mathcal{S} = \{00, 10, 1\}$ is known. What will be the codeword of an arithmetic code in this case?

3.2-5 Compute all parameters (a_s, b_s) and the estimated probability P_λ for the sequence 0100110 given the past 110, when the context tree is not known, using the context-tree weighting algorithm.

3.2-6 Based on the computations from the previous exercise, update the estimated probability for the sequence 01001101 given the past 110.

Show that for the cumulative probability of a binary sequence $(x_1 \ldots x_n)$ it is

$$Q^n(x_1 x_2 \ldots x_{n-1} x_n) = \sum_{j=1,\ldots,n;x_j=1} P^j(x_1 x_2 \ldots x_{j-1} 0) \ .$$

3.3. Ziv-Lempel-coding

In 1976–1978 Jacob Ziv and Abraham Lempel introduced two universal coding algorithms, which in contrast to statistical coding techniques, considered so far, do not make explicit use of the underlying probability distribution. The basic idea here is to replace a previously seen string with a pointer into a history buffer (LZ77) or with the index of a dictionary (LZ78). LZ algorithms are widely used—"zip" and its variations use the LZ77 algorithm. So, in contrast to the presentation by several authors, Ziv-Lempel-coding is not a single algorithm. Originally, Lempel and Ziv introduced a method to measure the complexity of a string—like in Kolmogorov complexity. This led to two different algorithms, LZ77 and LZ78. Many modifications and variations have been developed since. However, we shall present the original algorithms and refer to the literature for further information.

3.3.1. LZ77

The idea of LZ77 is to pass a ***sliding window*** over the text to be compressed. One looks for the longest substring in this window representing the next letters of the text. The window consists of two parts: a history window of length l_h, say, in which the last l_h bits of the text considered so far are stored, and a lookahead window of length l_f containing the next l_f bits of the text. In the simplest case l_h and l_f are fixed. Usually, l_h is much bigger than l_f. Then one encodes the triple (offset, length, letter). Here the ***offset*** is the number of letters one has to go back in the text to find the matching substring, the length is just the length of this matching substring, and the letter to be stored is the letter following the matching substring. Let us illustrate this procedure with an example. Assume the text to be compressed is *...abaabbaabbaaabbbaaaabbabbbabbb...*, the window is of size 15 with $l_h = 10$ letters history and $l_f = 5$ letters lookahead buffer. Assume, the sliding window now arrived at

$$...aba||abbaabbaaa|bbbaa|| \ ,$$

i. e., the history window contains the 10 letters *abbaabbaaa* and the lookahead window contains the five letters *bbbaa*. The longest substring matching the first letters of the lookahead window is *bb* of length 2, which is found nine letters back from the right end of the history window. So we encode $(9, 2, b)$, since b is the next letter (the string *bb* is also found five letters back, in the original LZ77 algorithm one would select the largest offset). The window then is moved 3 letters forward

$$...abaabb||aabbaaabbb|aaaab|| \ .$$

The next codeword is $(6, 3, a)$, since the longest matching substring is *aaa* of length 3 found 6 letters backwards and a is the letter following this substring in the lookahead window. We proceed with

$$...abaabbaabb||aaabbbaaaa|bbabb|| \ ,$$

and encode $(6, 3, b)$. Further

$$...abaabbaabbaaab||bbaaaabbab|babbb|| \ .$$

Here we encode $(3, 4, b)$. Observe that the match can extend into the lookahead window.

There are many subtleties to be taken into account. If a symbol did not appear yet in the text, offset and length are set to 0. If there are two matching strings of the same length, one has to choose between the first and the second offset. Both variations have advantages. Initially one might start with an empty history window and the first letters of the text to be compressed in the lookahead window - there are also further variations.

A common modification of the original scheme is to output only the pair (offset, length) and not the following letter of the text. Using this coding procedure one has to take into consideration the case in which the next letter does not occur in the history window. In this case, usually the letter itself is stored, such that the decoder has to distinguish between pairs of numbers and single letters. Further variations do not necessarily encode the longest matching substring.

3.3.2. LZ78

LZ78 does not use a sliding window but a dictionary which is represented here as a table with an index and an entry. LZ78 parses the text to be compressed into a collection of strings, where each string is the longest matching string α seen so far plus the symbol s following α in the text to be compressed. The new string αs is added into the dictionary. The new entry is coded as (i, s), where i is the index of the existing table entry α and s is the appended symbol.

As an example, consider the string "*abaabbaabbaaabbbaaaabba*". It is divided by LZ78 into strings as shown below. String 0 is here the empty string.

Input	a	b	aa	bb	aab	ba	$aabb$	baa	$aabba$
String Index	1	2	3	4	5	6	7	8	9
Output	$(0, a)$	$(0, b)$	$(1, a)$	$(2, b)$	$(3, b)$	$(2, a)$	$(5, b)$	$(6, a)$	$(7, a)$.

Since we are not using a sliding window, there is no limit for how far back strings can reach. However, in practice the dictionary cannot continue to grow infinitely. There are several ways to manage this problem. For instance, after having reached the maximum number of entries in the dictionary, no further entries can be added to the table and coding becomes static. Another variation would be to replace older entries. The decoder knows how many bits must be reserved for the index of the string in the dictionary, and hence decompression is straightforward.

Correctness

Ziv-Lempel coding asymptotically achieves the best possible compression rate which again is the entropy rate of the source. The source model, however, is much more general than the discrete memoryless source. The stochastic process generating the next letter, is assumed to be stationary (the probability of a sequence does not depend on the instant of time, i. e. $P(X_1 = x_1, \ldots, X_n = x_n) = P(X_{t+1} = x_1, \ldots, X_{t+n} = x_n)$ for all t and all sequences $(x_1 \ldots x_n)$). For stationary processes

the limit $\lim_{n\to\infty} \frac{1}{n} H(X_1, \ldots X_n)$ exists and is defined to be the entropy rate.

If s(n) denotes the number of strings in the parsing process of LZ78 for a text generated by a stationary source, then the number of bits required to encode all these strings is $s(n) \cdot (\lg s(n) + 1)$. It can be shown that $\frac{s(n)\cdot(\lg s(n)+1)}{n}$ converges to the entropy rate of the source. However, this would require that all strings can be stored in the dictionary.

Analysis

If we fix the size of the sliding window or the dictionary, the running time of encoding a sequence of n letters will be linear in n. However, as usually in data compression, there is a tradeoff between compression rate and speed. A better compression is only possible with larger memory. Increasing the size of the dictionary or the window will, however, result in a slower performance, since the most time consuming task is the search for the matching substring or the position in the dictionary.

Decoding in both LZ77 and LZ78 is straightforward. Observe that with LZ77 decoding is usually much faster than encoding, since the decoder already obtains the information at which position in the history he can read out the next letters of the text to be recovered, whereas the encoder has to find the longest matching substring in the history window. So algorithms based on LZ77 are useful for files which are compressed once and decompressed more frequently.

Further, the encoded text is not necessarily shorter than the original text. Especially in the beginning of the encoding the coded version may expand a lot. This expansion has to be taken into consideration.

For implementation it is not optimal to represent the text as an array. A suitable data structure will be a circular queue for the lookahead window and a binary search tree for the history window in LZ77, while for LZ78 a dictionary tree should be used.

Exercises

3.3-1 Apply the algorithms LZ77 and LZ78 to the string "abracadabra".

3.3-2 Which type of files will be well compressed with LZ77 and LZ78, respectively? For which type of files are LZ77 and LZ78 not so advantageous?

3.3-3 Discuss the advantages of encoding the first or the last offset, when several matching substrings are found in LZ77.

3.4. The Burrows-Wheeler-transform

The ***Burrows-Wheeler-transform*** will best be demonstrated by an example. Assume that our original text is $\vec{X}$ = "WHEELER". This text will be mapped to a second text $\vec{L}$ and an index I according to the following rules.

1) We form a matrix M consisting of all cyclic shifts of the original text $\vec{X}$. In our example

$$M = \begin{pmatrix} W & H & E & E & L & E & R \\ H & E & E & L & E & R & W \\ E & E & L & E & R & W & H \\ E & L & E & R & W & H & E \\ L & E & R & W & H & E & E \\ E & R & W & H & E & E & L \\ R & W & H & E & E & L & E \end{pmatrix}.$$

2) From M we obtain a new matrix M' by simply ordering the rows in M lexicographically. Here this yields the matrix

$$M' = \begin{pmatrix} E & E & L & E & R & W & H \\ E & L & E & R & W & H & E \\ E & R & W & H & E & E & L \\ H & E & E & L & E & R & W \\ L & E & R & W & H & E & E \\ R & W & H & E & E & L & E \\ W & H & E & E & L & E & R \end{pmatrix}.$$

3) The transformed string $\vec{L}$ then is just the last column of the matrix M' and the index I is the number of the row of M', in which the original text is contained. In our example $\vec{L}$ = "HELWEER" and $I = 6$ – we start counting the the rows with row no. 0.

This gives rise to the following pseudocode. We write here X instead of $\vec{X}$ and L instead of $\vec{L}$, since the purpose of the vector notation is only to distinguish the vectors from the letters in the text.

BWT-ENCODER(X)

```
1  for j ← 0 to n − 1
2      do M[0, j] ← X[j]
3  for i ← 0 to n − 1
4      do for j ← 0 to n − 1
5             do M[i, j] ← M[i − 1, j + 1 mod n]
6  for i ← 0 to n − 1
7      do row i of M′ ← row i of M in lexicographic order
8  for i ← 0 to n − 1
9      do L[i] ← M′[i, n − 1]
10 i = 0
11 while (row i of M′ ≠ row i of M)
12       do i ← i + 1
13 I ← i
14 return L and I
```

It can be shown that this transformation is invertible, i. e., it is possible to reconstruct the original text $\vec{X}$ from its transform $\vec{L}$ and the index I. This is because

these two parameters just yield enough information to find out the underlying permutation of the letters. Let us illustrate this reconstruction using the above example again. From the transformed string $\vec{L}$ we obtain a second string $\vec{E}$ by simply ordering the letters in $\vec{L}$ in ascending order. Actually, $\vec{E}$ is the first column of the matrix M' above. So, in our example

$$\vec{L} = \text{“}H \quad E \quad L \quad W \quad E \quad E \quad R''$$

$$\vec{E} = \text{“}E \quad E \quad E \quad H \quad L \quad R \quad W'' \,.$$

Now obviously the first letter $\vec{X}(0)$ of our original text $\vec{X}$ is the letter in position I of the sorted string $\vec{E}$, so here $\vec{X}(0) = \vec{E}(6) = W$. Then we look at the position of the letter just considered in the string $\vec{L}$ – here there is only one W, which is letter no. 3 in $\vec{L}$. This position gives us the location of the next letter of the original text, namely $\vec{X}(1) = \vec{E}(3) = H$. H is found in position no. 0 in $\vec{L}$, hence $\vec{X}(2) = \vec{E}(0) = E$. Now there are three E–s in the string $\vec{L}$ and we take the first one not used so far, here the one in position no. 1, and hence $\vec{X}(3) = \vec{E}(1) = E$. We iterate this procedure and find $\vec{X}(4) = \vec{E}(4) = L$, $\vec{X}(5) = \vec{E}(2) = E$, $\vec{X}(6) = \vec{E}(5) = R$.

This suggests the following pseudocode.

BWT-DECODER(L, I)

```
1  E[0..n − 1] ← sort L[0..n − 1]
2  pi[−1] ← I
3  for i ← 0 to n − 1
4       do j = 0
5            while (L[j]) ≠ E[pi[i − 1]] OR j is a component of pi)
6                 do j ← j + 1
7                    pi[i] ← j
8                    X[i] ← L[j]
9  return X
```

This algorithm implies a more formal description. Since the decoder only knows $\vec{L}$, he has to sort this string to find out $\vec{E}$. To each letter $\vec{L}(j)$ from the transformed string $\vec{L}$ record the position $\pi(j)$ in $\vec{E}$ from which it was jumped to by the process described above. So the vector pi in our pseudocode yields a permutation π such that for each $j = 0, \ldots, n-1$ row j it is $\vec{L}(j) = \vec{E}(\pi(j))$ in matrix M. In our example $\pi = (3, 0, 1, 4, 2, 5, 6)$. This permutation can be used to reconstruct the original text $\vec{X}$ of length n via $\vec{X}(n-1-j) = \vec{L}(\pi^j(I))$, where $\pi^0(x) = x$ and $\pi^j(x) = \pi(\pi^{j-1}(x))$ for $j = 1, \ldots, n-1$.

Observe that so far the original data have only been transformed and are not compressed, since string $\vec{L}$ has exactly the same length as the original string $\vec{L}$. So what is the advantage of the Burrows-Wheeler transformation? The idea is that the transformed string can be much more efficiently encoded than the original string. The dependencies among the letters have the effect that in the transformed string $\vec{L}$ there appear long blocks consisting of the same letter.

In order to exploit such frequent blocks of the same letter, Burrows and Wheeler

suggested the following ***move-to-front-code***, which we shall illustrate again with our example above.

We write down a list containing the letters used in our text in alphabetic order indexed by their position in this list.

E	H	L	R	W
0	1	2	3	4

Then we parse through the transformed string $\vec{L}$ letter by letter, note the index of the next letter and move this letter to the front of the list. So in the first step we note 1—the index of the H, move H to the front and obtain the list

H	E	L	R	W
0	1	2	3	4

Then we note 1 and move E to the front,

E	H	L	R	W
0	1	2	3	4

note 2 and move L to the front,

L	E	H	R	W
0	1	2	3	4

note 4 and move W to the front,

W	L	E	H	R
0	1	2	3	4

note 2 and move E to the front,

E	W	L	H	R
0	1	2	3	4

note 0 and leave E at the front,

E	W	L	H	R
0	1	2	3	4

note 4 and move R to the front,

R	E	W	L	H
0	1	2	3	4

So we obtain the sequence $(1, 1, 2, 4, 2, 0, 4)$ as our move-to-front-code. The pseudocode may look as follows, where Q is a list of the letters occuring in the string $\vec{L}$.

```
MOVE-TO-FRONT(L)
1  Q[0..n − 1] ← list of m letters occuring in L ordered alphabetically
2  for i ← 0 to n − 1
3      do j = 0
4          while (j ≠ L[i])
5                  j ← j + 1
6          c[i] ← j
7  for l ← 0 to j
8      do Q[l] ← Q[l − 1 mod j + 1]
9  return c
```

The move-to-front-code c will finally be compressed, for instance by Huffman-coding.

Correctness

The compression is due to the move-to-front-code obtained from the transformed string $\vec{L}$. It can easily be seen that this move-to-front coding procedure is invertible, so one can recover the string $\vec{L}$ from the code obtained as above.

Now it can be observed that in the move-to-front-code small numbers occur more frequently. Unfortunately, this will become obvious only with much longer texts than in our example—in long strings it was observed that even about 70 per cent of the numbers are 0. This irregularity in distribution can be exploited by compressing the sequence obtained after move-to-front-coding, for instance by Huffman-codes or run-length codes.

The algorithm performed very well in practice regarding the compression rate as well as the speed. The asymptotic optimality of compression has been proven for a wide class of sources.

Analysis

The most complex part of the Burrows-Wheeler transform is the sorting of the block yielding the transformed string $\vec{L}$. Due to fast sorting procedures, especially suited for the type of data to be compressed, compression algorithms based on the Burrows-Wheeler transform are usually very fast. On the other hand, compression is done blockwise. The text to be compressed has to be divided into blocks of appropriate size such that the matrices M and M' still fit into the memory. So the decoder has to wait until the whole next block is transmitted and cannot work sequentially bit by bit as in arithmetic coding or Ziv-Lempel coding.

Exercises

3.4-1 Apply the Burrows-Wheeler-transform and the move-to-front code to the text "abracadabra".

3.4-2 Verify that the transformed string $\vec{L}$ and the index i of the position in the sorted text $\vec{E}$ (containing the first letter of the original text to be compressed) indeed yield enough information to reconstruct the original text.

3.4-3 Show how in our example the decoder would obtain the string $\vec{L}$ ="HELWEER" from the move-to-front code $(1, 1, 2, 4, 2, 0, 4)$ and the letters E,H,L,W,R occuring in the text. Describe the general procedure for decoding move-to-front codes.

3.4-4 We followed here the encoding procedure presented by Burrows and Wheeler. Can the encoder obtain the transformed string $\vec{L}$ even without constructing the two matrices M and M'?

3.5. Image compression

The idea of image compression algorithms is similar to the one behind the Burrows-Wheeler-transform. The text to be compressed is transformed to a format which is suitable for application of the techniques presented in the previous sections, such as Huffman coding or arithmetic coding. There are several procedures based on the type of image (for instance, black/white, greyscale or colour image) or compression (lossless or lossy). We shall present the basic steps—representation of data, discrete cosine transform, quantisation, coding—of lossy image compression procedures like the standard ***JPEG***.

3.5.1. Representation of data

A greyscale image is represented as a two-dimensional array X, where each entry $X(i,j)$ represents the intensity (or brightness) at position (i,j) of the image. Each $X(i,j)$ is either a signed or an unsigned k-bit integers, i. e., $X(i,j) \in \{0,\ldots,2^k-1\}$ or $X(i,j) \in \{-2^{k-1},\ldots,2^{k-1}-1\}$.

A position in a colour image is usually represented by three greyscale values $R(i,j)$, $G(i,j)$, and $B(i,j)$ per position corresponding to the intensity of the primary colours red, green and blue.

In order to compress the image, the three arrays (or channels) R, G, B are first converted to the luminance/chrominance space by the YC_bC_r***-transform*** (performed entry–wise)

$$\begin{pmatrix} Y \\ C_b \\ C_r \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.5 \\ 0.5 & -0.419 & -0.0813 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$Y = 0.299R + 0.587G + 0.114B$ is the luminance or intensity channel, where the coefficients weighting the colours have been found empirically and represent the best possible approximation of the intensity as perceived by the human eye. The chrominance channels $C_b = 0.564(B-Y)$ and $C_r = 0.713(R-Y)$ contain the colour information on red and blue as the differences from Y. The information on green is obtained as big part in the luminance Y.

A first compression for colour images commonly is already obtained after application of the YC_bC_r-transform by removing ***irrelevant information***. Since the human eye is less sensitive to rapid colour changes than to changes in intensity, the resolution of the two chrominance channels C_b and C_r is reduced by a factor of 2 in both vertical and horizontal direction, which results after sub-sampling in arrays of $\frac{1}{4}$ of the original size.

The arrays then are subdivided into 8×8 blocks, on which successively the actual (lossy) data compression procedure is applied.

Let us consider the following example based on a real image, on which the steps of compression will be illustrated. Assume that the 8×8 block of 8-bit unsigned integers below is obtained as a part of an image.

$$f = \begin{pmatrix} 139 & 144 & 149 & 153 & 155 & 155 & 155 & 155 \\ 144 & 151 & 153 & 156 & 159 & 156 & 156 & 155 \\ 150 & 155 & 160 & 163 & 158 & 156 & 156 & 156 \\ 159 & 161 & 162 & 160 & 160 & 159 & 159 & 159 \\ 159 & 160 & 161 & 161 & 160 & 155 & 155 & 155 \\ 161 & 161 & 161 & 161 & 160 & 157 & 157 & 157 \\ 162 & 162 & 161 & 163 & 162 & 157 & 157 & 157 \\ 161 & 162 & 161 & 161 & 163 & 158 & 158 & 158 \end{pmatrix}$$

3.5.2. The discrete cosine transform

Each 8×8 block $(f(i,j))_{i,j=0,\ldots,7}$, say, is transformed into a new block $(F(u,v))_{u,v=0,\ldots,7}$. There are several possible transforms, usually the ***discrete cosine transform*** is applied, which here obeys the formula

$$F(u,v) = \frac{1}{4} c_u c_v \left(\sum_{i=0}^{7} \sum_{j=0}^{7} f(i,j) \cdot \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16} \right)$$

The cosine transform is applied after shifting the unsigned integers to signed integers by subtraction of 2^{k-1}.

DCT(f)

1 **for** $u \leftarrow 0$ **to** 7
2 **do for** $v \leftarrow 0$ **to** 7
3 **do** $F(u,v) \leftarrow$ DCT - coefficient of matrix f
4 **return** F

The coefficients need not be calculated according to the formula above. They can also be obtained via a related Fourier transform (see Exercises) such that a Fast Fourier Transform may be applied. JPEG also supports wavelet transforms, which may replace the discrete cosine transform here.

The discrete cosine transform can be inverted via

$$f(i,j) = \frac{1}{4} \left(\sum_{u=0}^{7} \sum_{v=0}^{7} c_u c_v F(u,v) \cdot \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16} \right) ,$$

where $c_u = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ 1 & \text{for } u \neq 0 \end{cases}$ and $c_v = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } v = 0 \\ 1 & \text{for } v \neq 0 \end{cases}$ are normalisation constants.

In our example, the transformed block F is

$$F = \begin{pmatrix} 235.6 & -1.0 & -12.1 & -5.2 & 2.1 & -1.7 & -2.7 & 1.3 \\ -22.6 & -17.5 & -6.2 & -3.2 & -2.9 & -0.1 & 0.4 & -1.2 \\ -10.9 & -9.3 & -1.6 & 1.5 & 0.2 & -0.9 & -0.6 & -0.1 \\ -7.1 & -1.9 & 0.2 & 1.5 & 0.9 & -0.1 & 0.0 & 0.3 \\ -0.6 & -0.8 & 1.5 & 1.6 & -0.1 & -0.7 & 0.6 & 1.3 \\ 1.8 & -0.2 & 1.6 & -0.3 & -0.8 & 1.5 & 1.0 & -1.0 \\ -1.3 & -0.4 & -0.3 & -1.5 & -0.5 & 1.7 & 1.1 & -0.8 \\ -2.6 & 1.6 & -3.8 & -1.8 & 1.9 & 1.2 & -0.6 & -0.4 \end{pmatrix}$$

where the entries are rounded.

The discrete cosine transform is closely related to the discrete Fourier transform and similarly maps signals to frequencies. Removing higher frequencies results in a less sharp image, an effect that is tolerated, such that higher frequencies are stored with less accuracy.

Of special importance is the entry $F(0,0)$, which can be interpreted as a measure for the intensity of the whole block.

3.5.3. Quantisation

The discrete cosine transform maps integers to real numbers, which in each case have to be rounded to be representable. Of course, this rounding already results in a loss of information. However, the transformed block F will now be much easier to manipulate. A ***quantisation*** takes place, which maps the entries of F to integers by division by the corresponding entry in a luminance quantisation matrix Q. In our example we use

$$Q = \begin{pmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{pmatrix}.$$

The quantisation matrix has to be carefully chosen in order to leave the image at highest possible quality. Quantisation is the lossy part of the compression procedure. The idea is to remove information which should not be "visually significant". Of course, at this point there is a tradeoff between the compression rate and the quality of the decoded image. So, in JPEG the quantisation table is not included into the standard but must be specified (and hence be encoded).

QUANTISATION(F)

```
1  for i ← 0 to 7
2      do for j ← 0 to 7
3             do T(i,j) ← {F(i,j)/Q(i,j)}
4  return T
```

This quantisation transforms block F to a new block T with $T(i,j) = \{\frac{F(i,j)}{Q(i,j)}\}$, where $\{x\}$ is the closest integer to x. This block will finally be encoded. Observe that in the transformed block F besides the entry $F(0,0)$ all other entries are relatively small numbers, which has the effect that T mainly consists of 0s .

$$T = \begin{pmatrix} 15 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Coefficient $T(0,0)$, in this case 15, deserves special consideration. It is called DC term (direct current), while the other entries are denoted AC coefficients (alternate current).

3.5.4. Coding

Matrix T will finally be encoded by a Huffman code. We shall only sketch the procedure. First the DC term will be encoded by the difference to the DC term of the previously encoded block. For instance, if the previous DC term was 12, then $T(0,0)$ will be encoded as -3.

After that the AC coefficients are encoded according to the zig-zag order $T(0,1)$, $T(1,0)$, $T(2,0)$, $T(1,1)$, $T(0,2)$, $T(0,3)$, $T(1,2)$, etc.. In our example, this yields the sequence $0, -2, -1, -1, -1, 0, 0, -1$ followed by 55 zeros. This zig–zag order exploits the fact that there are long runs of successive zeros. These runs will be even more efficiently represented by application of ***run-length coding,*** i. e., we encode the number of zeros before the next nonzero element in the sequence followed by this element.

Integers are written in such a way that small numbers have shorter representations. This is achieved by splitting their representation into size (number of bits to be reserved) and amplitude (the actual value). So, 0 has size 0, 1 and -1 have size 1. -3, -2, 2, and 3 have size 2, etc.

In our example this yields the sequence (2)(3) for the DC term followed by $(1,2)(-2)$, $(0,1)(-1)$, $(0,1)(-1)$, $(0,1)(-1)$, $(2,1)(-1)$, and a final $(0,0)$ as an end-of-block symbol indicating that only zeros follow from now on. $(1,2)(-2)$, for instance, means that there is 1 zero followed by an element of size 2 and amplitude -2.

These pairs are then assigned codewords from a Huffman code. There are diffe-

rent Huffman codes for the pairs (run, size) and for the amplitudes. These Huffman codes have to be specified and hence be included into the code of the image.

In the following pseudocode for the encoding of a single 8×8-block T we shall denote the different Huffman codes by encode-1, encode-2, encode-3.

RUN-LENGTH-CODE(T)

```
1  c ← encode-1(size(DC − T[0,0]))
2  c ← c|| encode-3(amplitude(DC − T[00]))
3  DC ← T[0,0]
4  for l ← 1 to 14
5      do for i ← 0 to l
6             do if l = 1 mod 2
7                    then u ← i
8                    else  u ← l − i
9             if T[u, l − u] = 0
10               then run ← run + 1
11               else  c ← c|| encode -2(run, size(T[u, l − u]))
12                     c ← c|| encode-3(amplitude(T[u, l − u])
13                     run ← 0
14 if run > 0
15    then encode-2(0,0)
16 return c
```

At the decoding end matrix T will be reconstructed. Finally, by multiplication of each entry $T(i,j)$ by the corresponding entry $Q(i,j)$ from the quantisation matrix Q we obtain an approximation $\overline{F}$ to the block F, here

$$\overline{F} = \begin{pmatrix} 240 & 0 & -10 & 0 & 0 & 0 & 0 & 0 \\ -24 & -12 & 0 & 0 & 0 & 0 & 0 & 0 \\ -14 & -13 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

To $\overline{F}$ the inverse cosine transform is applied. This allows to decode the original 8×8–block f of the original image – in our example as

$$\overline{f} = \begin{pmatrix} 144 & 146 & 149 & 152 & 154 & 156 & 156 & 156 \\ 148 & 150 & 152 & 154 & 156 & 156 & 156 & 156 \\ 155 & 156 & 157 & 158 & 158 & 157 & 156 & 155 \\ 160 & 161 & 161 & 162 & 161 & 159 & 157 & 155 \\ 163 & 163 & 164 & 163 & 162 & 160 & 158 & 156 \\ 163 & 164 & 164 & 164 & 162 & 160 & 158 & 157 \\ 160 & 161 & 162 & 162 & 162 & 161 & 159 & 158 \\ 158 & 159 & 161 & 161 & 162 & 161 & 159 & 158 \end{pmatrix}.$$

Exercises

3.5-1 Find size and amplitude for the representation of the integers 5, -19, and 32.

3.5-2 Write the entries of the following matrix in zig – zag order.

$$\begin{pmatrix} 5 & 0 & -2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

How would this matrix be encoded if the difference of the DC term to the previous one was -2?

3.5-3 In our example after quantising the sequence (2)(3), $(1,2)(-2)$, $(0,1)(-1)$, $(0,1)(-1)$, $(0,1)(-1)$, $(2,1)(-1)$, $(0,0)$ has to be encoded. Assume the Huffman codebooks would yield 011 to encode the difference 2 from the preceding block's DC, 0, 01, and 11 for the amplitudes -1, -2, and 3, respectively, and 1010, 00, 11011, and 11100 for the pairs $(0,0)$, $(0,1)$, $(1,2)$, and $(2,1)$, respectively. What would be the bitstream to be encoded for the 8×8 block in our example? How many bits would hence be necessary to compress this block?

3.5-4 What would be matrices T, $\overline{F}$ and $\overline{f}$, if we had used

$$Q = \begin{pmatrix} 8 & 6 & 5 & 8 & 12 & 20 & 26 & 31 \\ 6 & 6 & 7 & 10 & 13 & 29 & 30 & 28 \\ 7 & 7 & 8 & 12 & 20 & 29 & 35 & 28 \\ 7 & 9 & 11 & 15 & 26 & 44 & 40 & 31 \\ 9 & 11 & 19 & 28 & 34 & 55 & 52 & 39 \\ 12 & 18 & 28 & 32 & 41 & 52 & 57 & 46 \\ 25 & 32 & 39 & 44 & 57 & 61 & 60 & 51 \\ 36 & 46 & 48 & 49 & 56 & 50 & 57 & 50 \end{pmatrix}$$

for quantising after the cosine transform in the block of our example?

3.5-5 What would be the zig-zag-code in this case (assuming again that the DC term would have difference -3 from the previous DC term)?

3.5-6 For any sequence $(f(n))_{n=0,\ldots,m-1}$ define a new sequence $(\hat{f}(n))_{n=0,\ldots,2m-1}$ by

$$\hat{f}(n) = \begin{cases} f(n) & \text{for } n = 0,\ldots,m-1 \\ f(2m-1-n) & \text{for } n = m,\ldots,2m-1 \end{cases} .$$

This sequence can be expanded to a Fourier-series via

$$\hat{f}(n) = \frac{1}{\sqrt{2m}} \sum_{n=0}^{2m-1} \hat{g}(u) e^{i\frac{2\pi}{2m}nu} \quad \text{with } \hat{g}(u) = \frac{1}{\sqrt{2m}} \sum_{n=0}^{2m-1} \hat{f}(u) e^{-i\frac{2\pi}{2m}nu}, \quad i = \sqrt{-1} .$$

Show how the coefficients of the discrete cosine transform

$$F(u) = c_u \sum_{n=0}^{m-1} f(n) \cos(\frac{(2n+1)\pi u}{2m}, \quad c_u = \begin{cases} \frac{1}{\sqrt{m}} & \text{for } u = 0 \\ \frac{2}{\sqrt{m}} & \text{for } u \neq 0 \end{cases}$$

arise from this Fourier series.

Problems

3-1 Adaptive Huffman-codes

Dynamic and adaptive Huffman-coding is based on the following property. A binary code tree has the sibling property if each node has a sibling and if the nodes can be listed in order of nonincreasing probabilities with each node being adjacent to its sibling. Show that a binary prefix code is a Huffman-code exactly if the corresponding code tree has the sibling property.

3-2 Generalisations of Kraft's inequality

In the proof of Kraft's inequality it is essential to order the lengths $L(1) \leq \cdots \leq L(a)$. Show that the construction of a prefix code for given lengths $2, 1, 2$ is not possible if we are not allowed to order the lengths. This scenario of unordered lengths occurs with the Shannon-Fano-Elias-code and in the theory of alphabetic codes, which are related to special search problems. Show that in this case a prefix code with lengths $L(1) \leq \cdots \leq L(a)$ exists if and only if

$$\sum_{x \in \mathcal{X}} 2^{-L(x)} \leq \frac{1}{2} \, .$$

If we additionally require the prefix codes to be also suffix-free i. e., no codeword is the end of another one, it is an open problem to show that Kraft's inequality holds with the 1 on the right–hand side replaced by $3/4$, i. e.,

$$\sum_{x \in \mathcal{X}} 2^{-L(x)} \leq \frac{3}{4} \, .$$

3-3 Redundancy of Krichevsky-Trofimov-estimator

Show that using the Krichevsky-Trofimov-estimator, when parameter θ of a discrete memoryless source is unknown, the individual redundancy of sequence x^n is at most $\frac{1}{2} \lg n + 3$ for all sequences x^n and all $\theta \in \{0, 1\}$.

3-4 Alternatives to move-to-front-codes

Find further procedures which like move-to-front-coding prepare the text for compression after application of the Burrows-Wheeler-transform.

Chapter notes

The frequency table of the letters in English texts is taken from [254]. The Huffman coding algorithm was introduced by Huffman in [113]. A pseudocode can be found in

[51], where the Huffman coding algorithm is presented as a special Greedy algorithm. There are also adaptive or dynamic variants of Huffman-coding, which adapt the Huffman-code if it is no longer optimal for the actual frequency table, for the case that the probability distribution of the source is not known in advance. The "3/4-conjecture" on Kraft's inequality for fix-free-codes is due to Ahlswede, Balkenhol, and Khachatrian [3].

Arithmetic coding has been introduced by Rissanen [198] and Pasco [185]. For a discussion of implementation questions see [146, 146, 259]. In the section on modelling we are following the presentation of Willems, Shtarkov and Tjalkens in [257]. The exact calculations can be found in their original paper [256] which received the Best Paper Award of the IEEE Information Theory Society in 1996. The Krichevsky-Trofimov-estimator had been introduced in [140].

We presented the two original algorithms LZ77 and LZ78 [263, 264] due to Lempel and Ziv. Many variants, modifications and extensions have been developed since that – concerning the handling of the dictionary, the pointers, the behaviour after the dictionary is complete, etc. For a description, see, for instance, [23] or [24]. Most of the prominent tools for data compression are variations of Ziv-Lempel-coding. For example "zip" and "gzip" are based on LZ77 and a variant of LZ78 is used by the program "compress".

The Burrows-Wheeler transform was introduced in the technical report [34]. It became popular in the sequel, especially because of the Unix compression tool "bzip" based on the Burrows-Wheeler-transform, which outperformed most dictionary—based tools on several benchmark files. Also it avoids arithmetic coding, for which patent rights have to be taken into consideration. Further investigations on the Burrows-Wheeler-transform have been carried out, for instance in [20, 64, 143].

We only sketched the basics behind lossy image compression, especially the preparation of the data for application of techniques as Huffman coding. For a detailed discussion we refer to [241], where also the new JPEG2000 standard is described. Our example is taken from [251].

JPEG—short for Joint Photographic Experts Group—is very flexible. For instance, it also supports lossless data compression. All the topics presented in the section on image compression are not unique. There are models involving more basic colours and further transforms besides the YC_bC_r-transform (for which even different scaling factors for the chrominance channels were used, the formula presented here is from [241]). The cosine transform may be replaced by another operation like a wavelet transform. Further, there is freedom to choose the quantisation matrix, responsible for the quality of the compressed image, and the Huffman code. On the other hand, this has the effect that these parameters have to be explicitly specified and hence are part of the coded image.

The ideas behind procedures for video and sound compression are rather similar to those for image compression. In principal, they follow the same steps. The amount of data in these cases, however, is much bigger. Again information is lost by removing irrelevant information not realizable by the human eye or ear (for instance by psychoacoustic models) and by quantising, where the quality should not be reduced significantly. More refined quantising methods are applied in these cases.

Most information on data compression algorithms can be found in literature

on Information Theory, for instance [52, 100], since the analysis of the achievable compression rates requires knowledge of source coding theory. Recently, there have appeared several books on data compression, for instance [24, 101, 177, 210, 211], to which we refer to further reading. The benchmark files of the Calgary Corpus and the Canterbury Corpus are available under [35] or [36].

The book of I. Csiszár and J. Körner [55] analyses different aspects of information theory including the problems of data compression too.

4. Reliable Computation

Any planned computation will be subject to different kinds of unpredictable influences during execution. Here are some examples:

(1) Loss or change of stored data during execution.

(2) Random, physical errors in the computer.

(3) Unexpected interactions between different parts of the system working simultaneously, or loss of connections in a network.

(4) Bugs in the program.

(5) Malicious attacks.

Up to now, it does not seem that the problem of bugs can be solved just with the help of appropriate algorithms. The discipline of software engineering addresses this problem by studying and improving the structure of programs and the process of their creation.

Malicious attacks are addressed by the discipline of computer security. A large part of the recommended solutions involves cryptography.

Problems of kind (3) are very important and a whole discipline, distributed computing has been created to deal with them.

The problem of storage errors is similar to the problems of reliable communication, studied in information theory: it can be viewed as communication from the present to the future. In both cases, we can protect against noise with the help of ***error-correcting codes*** (you will see some examples below).

In this chapter, we will discuss some sample problems, mainly from category (2). In this category, distinction should also be made between permanent and transient errors. An error is ***permanent*** when a part of the computing device is damaged physically and remains faulty for a long time, until some outside intervention by repairmen to it. It is ***transient*** if it happens only in a single step: the part of the device in which it happened is not damaged, in the next step it operates correctly again. For example, if a position in memory turns from 0 to 1 by accident, but a subsequent write operation can write a 0 again then a transient error happened. If the bit turned to 1 and the computer cannot change it to 0 again, this is a permanent error.

Some of these problems, especially the ones for transient errors, are as old as computing. The details of any physical errors depend on the kind of computer it is implemented on (and, of course, on the kind of computation we want to carry out). But after abstracting away from a lot of distracting details, we are left with some clean but challenging theoretical formulations, and some rather pleasing solutions. There are also interesting connections to other disciplines, like statistical physics and biology.

The computer industry has been amazingly successful over the last five decades in making the computer components smaller, faster, and at the same time more reliable. Among the daily computer horror stories seen in the press, the one conspicuously missing is where the processor wrote a 1 in place of a 0, just out of caprice. (It undisputably happens, but too rarely to become the identifiable source of some visible malfunction.) On the other hand, the generality of some of the results on the correction of transient errors makes them applicable in several settings. Though individual physical processors are very reliable (error rate is maybe once in every 10^{20} executions), when considering a whole network as performing a computation, the problems caused by unreliable network connections or possibly malicious network participants is not unlike the problems caused by unreliable processors.

The key idea for making a computation reliable is *redundancy*, which might be formulated as the following two procedures:

(i) Store information in such a form that losing any small part of it is not fatal: it can be restored using the rest of the data. For example, store it in multiple copies.

(ii) Perform the needed computations repeatedly, to make sure that the faulty results can be outvoted.

Our chapter will only use these methods, but there are other remarkable ideas which we cannot follow up here. For example, method (ii) seems especially costly; it is desireable to avoid a lot of repeated computation. The following ideas target this dilemma.

(A) Perform the computation directly on the information in its redundant form: then maybe recomputations can be avoided.

(B) Arrange the computation into "segments" such a way that those partial results that are to be used later, can be cheaply checked at each "milestone" between segments. If the checking finds error, repeat the last segment.

4.1. Probability theory

The present chapter does not require great sophistication in probability theory but there are some facts coming up repeatedly which I will review here. If you need additional information, you will find it in any graduate probability theory text.

4.1.1. Terminology

A ***probability space*** is a triple $(\Omega, \mathcal{A}, \mathbf{P})$ where Ω is the set of ***elementary events***, $\mathcal{A}$ is a set of subsets of Ω called the set of ***events*** and $\mathbf{P} : \mathcal{A} \to [0,1]$ is a function. For $E \in \mathcal{A}$, the value $\mathbf{P}(E)$ is called the ***probability*** of event E. It is required that $\Omega \in \mathcal{A}$ and that $E \in \mathcal{A}$ implies $\Omega \setminus E \in \mathcal{A}$. Further, if a (possibly infinite) sequence of sets is in $\mathcal{A}$ then so is their union. Also, it is assumed that $\mathbf{P}(\Omega) = 1$ and that if $E_1, E_2, \ldots \in \mathcal{A}$ are disjoint then

$$\mathbf{P}\Big(\bigcup_i E_i\Big) = \sum_i \mathbf{P}(E_i) \ .$$

For $\mathbf{P}(F) > 0$, the ***conditional probability*** of E given F is defined as

$$\mathbf{P}(E \mid F) = \mathbf{P}(E \cap F)/\mathbf{P}(F) \ .$$

Events $E_1, \ldots, E_n$ are ***independent*** if for any sequence $1 \le i_1 < \cdots < i_k \le n$ we have

$$\mathbf{P}(E_{i_1} \cap \cdots \cap E_{i_k}) = \mathbf{P}(E_{i_1}) \cdots \mathbf{P}(E_{i_k}) \ .$$

Example 4.1 Let $\Omega = \{1, \ldots, n\}$ where $\mathcal{A}$ is the set of all subsets of Ω and $\mathbf{P}(E) = |E|/n$. This is an example of a ***discrete*** probability space: one that has a countable number of elements.

More generally, a discrete probability space is given by a countable set $\Omega = \{\omega_1, \omega_2, \ldots\}$, and a sequence $p_1, p_2, \ldots$ with $p_i \ge 0$, $\sum_i p_i = 1$. The set $\mathcal{A}$ of events is the set of all subsets of Ω, and for an event $E \subset \Omega$ we define $\mathbf{P}(E) = \sum_{\omega_i \in E} p_i$.

A ***random variable*** over a probability space Ω is a function f from Ω to the real numbers, with the property that every set of the form $\{\, \omega : f(\omega) < c \,\}$ is an event: it is in $\mathcal{A}$. Frequently, random variables are denoted by capital letters X, Y, Z, possibly with indices, and the argument ω is omitted from $X(\omega)$. The event $\{\, \omega : X(\omega) < c \,\}$ is then also written as $[\, X < c \,]$. This notation is freely and informally extended to more complicated events. The ***distribution*** of a random variable X is the function $F(c) = \mathbf{P}[\, X < c \,]$. We will frequently only specify the distribution of our variables, and not mention the underlying probability space, when it is clear from the context that it can be specified in one way or another. We can speak about the ***joint distribution*** of two or more random variables, but only if it is assumed that they can be defined as functions on a common probability space. Random variables $X_1, \ldots, X_n$ with a joint distribution are ***independent*** if every n-tuple of events of the form $[\, X_1 < c_1 \,]$, $\ldots$, $[\, X_n < c_n \,]$ is independent.

The ***expected value*** of a random variable X taking values $x_1, x_2, \ldots$ with probabilities $p_1, p_2, \ldots$ is defined as

$$\mathbf{E}X = p_1 x_1 + p_2 x_2 + \cdots \ .$$

It is easy to see that the expected value is a linear function of the random variable:

$$\mathbf{E}(\alpha X + \beta Y) = \alpha \mathbf{E}X + \beta \mathbf{E}Y \ ,$$

even if X, Y are not independent. On the other hand, if variables X, Y are independent then the expected values can also be multiplied:

$$\mathbf{E}XY = \mathbf{E}X \cdot \mathbf{E}Y \ . \tag{4.1}$$

There is an important simple inequality called the ***Markov inequality***, which says that for an arbitrary nonnegative random variable X and any value $\lambda > 0$ we have

$$\mathbf{P}[\, X \geq \lambda \,] \leq \mathbf{E}X/\lambda \ . \tag{4.2}$$

4.1.2. The law of large numbers (with "large deviations")

In what follows the bounds

$$\frac{x}{1+x} \leq \ln(1+x) \leq x \quad \text{for } x > -1 \tag{4.3}$$

will be useful. Of these, the well-known upper bound $\ln(1+x) \leq x$ holds since the graph of the function $\ln(1+x)$ is below its tangent line drawn at the point $x = 0$. The lower bound is obtained from the identity $\frac{1}{1+x} = 1 - \frac{x}{1+x}$ and

$$-\ln(1+x) = \ln\frac{1}{1+x} = \ln\left(1 - \frac{x}{1+x}\right) \leq -\frac{x}{1+x} \ .$$

Consider n independent random variables $X_1, \ldots, X_n$ that are identically distributed, with

$$\mathbf{P}[\, X_i = 1 \,] = p, \quad \mathbf{P}[\, X_i = 0 \,] = 1 - p \ .$$

Let

$$S_n = X_1 + \cdots + X_n.$$

We want to estimate the probability $\mathbf{P}[\, S_n \geq fn \,]$ for any constant $0 < f < 1$. The "law of large numbers" says that if $f > p$ then this probability converges fast to 0 as $n \to \infty$ while if $f < p$ then it converges fast to 1. Let

$$D(f,p) = f \ln\frac{f}{p} + (1-f)\ln\frac{1-f}{1-p} \tag{4.4}$$

$$> f \ln\frac{f}{p} - f = f \ln\frac{f}{ep} \ , \tag{4.5}$$

where the inequality (useful for small f and $ep < f$) comes via $1 > 1 - p > 1 - f$ and $\ln(1-f) \geq -\frac{f}{1-f}$ from (4.3). Using the concavity of logarithm, it can be shown that $D(f,p)$ is always nonnegative, and is 0 only if $f = p$ (see Exercise 4.1-1).

Theorem 4.1 (Large deviations for coin-toss). *If $f > p$ then*

$$\mathbf{P}[\, S_n \geq fn \,] \leq e^{-nD(f,p)}.$$

This theorem shows that if $f > p$ then $\mathbf{P}[\, S_n > fn \,]$ converges to 0 exponentially fast. Inequality (4.5) will allow the following simplification:

$$\mathbf{P}[\, S_n \geq fn \,] \leq e^{-nf\ln\frac{f}{ep}} = \left(\frac{ep}{f}\right)^{nf}, \tag{4.6}$$

useful for small f and $ep < f$.

Proof. For a certain real number $\alpha > 1$ (to be chosen later), let Y_n be the random variable that is α if $X_n = 1$ and 1 if $X_n = 0$, and let $P_n = Y_1 \cdots Y_n = \alpha^{S_n}$: then

$$\mathbf{P}[\, S_n \geq fn \,] = \mathbf{P}[\, P_n \geq \alpha^{fn} \,] \; .$$

Applying the Markov inequality (4.2) and (4.1), we get

$$\mathbf{P}[\, P_n \geq \alpha^{fn} \,] \leq \mathbf{E}P_n/\alpha^{fn} = (\mathbf{E}Y_1/\alpha^f)^n,$$

where $\mathbf{E}Y_1 = p\alpha + (1-p)$. Let us choose $\alpha = \frac{f(1-p)}{p(1-f)}$, this is > 1 if $p < f$. Then we get $\mathbf{E}Y_1 = \frac{1-p}{1-f}$, and hence

$$\mathbf{E}Y_1/\alpha^f = \frac{p^f(1-p)^{1-f}}{f^f(1-f)^{1-f}} = e^{-D(f,p)}.$$

■

This theorem also yields some convenient estimates for binomial coefficients. Let

$$h(f) = -f \ln f - (1-f)\ln(1-f) \; .$$

This is sometimes called the ***entropy*** of the probability distribution $(f, 1-f)$ (measured in logarithms over base e instead of base 2). From inequality (4.3) we obtain the estimate

$$-f \ln f \leq h(f) \leq f \ln \frac{e}{f} \tag{4.7}$$

which is useful for small f.

Korollar 4.1 *We have, for $f \leq 1/2$:*

$$\sum_{i \leq fn}^{n} \binom{n}{i} \leq e^{nh(f)} \leq \left(\frac{e}{f}\right)^{fn}. \tag{4.8}$$

In particular, taking $f = k/n$ with $k \leq n/2$ gives

$$\binom{n}{k} = \binom{n}{fn} \leq \left(\frac{e}{f}\right)^{fn} = \left(\frac{ne}{k}\right)^{k} \; . \tag{4.9}$$

Proof. Theorem 4.1 says for the case $f > p = 1/2$:

$$2^{-n} \sum_{i \geq fn}^{n} \binom{n}{i} = \mathbf{P}[\, S_n \geq fn \,] \leq e^{-nD(f,p)} = 2^{-n} e^{nh(f)},$$

$$\sum_{i \geq fn}^{n} \binom{n}{i} \leq e^{nh(f)} \; .$$

Substituting $g = 1 - f$, and noting the symmetries $\binom{n}{f} = \binom{n}{g}$, $h(f) = h(g)$ and (4.7) gives (4.8). ■

Remark 4.2 *Inequality* (4.6) *also follows from the trivial estimate* $\mathbf{P}[\,S_n \geq fn\,] \leq \binom{n}{fn} p^{fn}$ *combined with* (4.9).

Exercises

4.1-1 Prove that the statement made in the main text that $D(f,p)$ is always non-negative, and is 0 only if $f = p$.

4.1-2 For $f = p + \delta$, derive from Theorem 4.1 the useful bound

$$\mathbf{P}[\,S_n \geq fn\,] \leq e^{-2\delta^2 n} \ .$$

Hint. Let $F(x) = D(x,p)$, and use the Taylor formula: $F(p+\delta) = F(p) + F'(p)\delta + F''(p+\delta')\delta^2/2$, where $0 \leq \delta' \leq \delta$.

4.1-3 Prove that in Theorem 4.1, the assumption that X_i are independent and identically distributed can be weakened: replaced by the single inequality

$$\mathbf{P}[\,X_i = 1 \mid X_1, \ldots, X_{i-1}\,] \leq p \ .$$

4.2. Logic circuits

In a model of computation taking errors into account, the natural assumption is that errors occur *everywhere*. The most familiar kind of computer, which is separated into a single processor and memory, seems extremely vulnerable under such conditions: while the processor is not "looking", noise may cause irreparable damage in the memory. Let us therefore rather consider computation models that are *parallel*: information is being processed everywhere in the system, not only in some distinguished places. Then error correction can be built into the work of every part of the system. We will concentrate on the best known parallel computation model: Boolean circuits.

4.2.1. Boolean functions and expressions

Let us look inside a computer, (actually inside an integrated circuit, with a microscope). Discouraged by a lot of physical detail irrelevant to abstract notions of computation, we will decide to look at the blueprints of the circuit designer, at the stage when it shows the smallest elements of the circuit still according to their computational functions. We will see a network of lines that can be in two states (of electric potential), "high" or "low", or in other words "true" or "false", or, as we will write, 1 or 0. The points connected by these lines are the familiar ***logic components***: at the lowest level of computation, a typical computer processes ***bits***. Integers, floating-point numbers, characters are all represented as strings of bits, and the usual arithmetical operations can be composed of bit operations.

Definition 4.2 *A* ***Boolean vector function*** *is a mapping* $f : \{0,1\}^n \to \{0,1\}^m$. *Most of the time, we will take* $m = 1$ *and speak of a* ***Boolean function***.

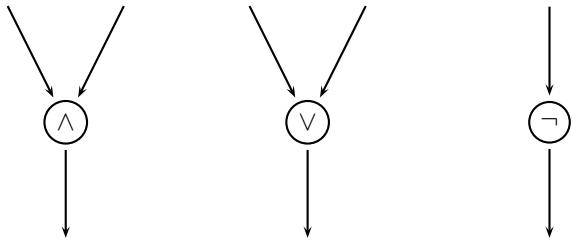


Figure 4.1. AND, OR and NOT gate.

The variables in $f(x_1, \ldots, x_n)$ are sometimes called ***Boolean variables***, ***Boolean variables*** or ***bits***.

Example 4.2 Given an undirected graph G with N nodes, suppose we want to study the question whether it has a Hamiltonian cycle (a sequence $(u_1, \ldots, u_n)$ listing all vertices of G such that (u_i, u_{i+1}) is an edge for each $i < n$ and also (u_n, u_1) is an edge). This question is described by a Boolean function f as follows. The graph can be described with $\binom{N}{2}$ Boolean variables x_{ij} $(1 \leq i < j \leq N)$: x_{ij} is 1 if and only if there is an edge between nodes i and j. We define $f(x_{12}, x_{13}, \ldots, x_{N-1,N}) = 1$ if there is a Hamiltonian cycle in G and 0 otherwise.

Example 4.3 [Boolean vector function] Let $n = m = 2k$, let the input be two integers u, v, written as k-bit strings: $x = (u_1, \ldots, u_k, v_1, \ldots, v_k)$. The output of the function is their product $y = u \cdot v$ (written in binary): if $u = 5 = (101)_2$, $v = 6 = (110)_2$ then $y = u \cdot v = 30 = (11110)_2$.

There are only four one-variable Boolean functions: the identically 0, identically 1, the identity and the ***negation***: $x \to \neg x = 1 - x$. We mention only the following two-variable Boolean functions: the operation of ***conjunction*** (logical AND):

$$x \wedge y = \begin{cases} 1 & \text{if } x = y = 1 \ , \\ 0 & \text{otherwise} \ , \end{cases}$$

this is the same as multiplication. The operation of ***disjunction***, or logical OR:

$$x \vee y = \begin{cases} 0 & \text{if } x = y = 0 \ , \\ 1 & \text{otherwise} \ . \end{cases}$$

It is easy to see that $x \vee y = \neg(\neg x \wedge \neg y)$: in other words, disjunction $x \vee y$ can be expressed using the functions $\neg, \wedge$ and the operation of ***composition***. The following two-argument Boolean functions are also frequently used:

$$\begin{aligned} x \to y &= \neg x \vee y && \text{(implication)}, \\ x \leftrightarrow y &= (x \to y) \wedge (y \to x) && \text{(equivalence)}, \\ x \oplus y &= x + y \bmod 2 = \neg(x \leftrightarrow y) && \text{(binary addition)} \ . \end{aligned}$$

A finite number of Boolean functions is sufficent to express all others: thus, arbitrarily complex Boolean functions can be "computed" by "elementary" operations. In some sense, this is what happens inside computers.

Definition 4.3 *A set of Boolean functions is a* ***complete basis*** *if every other Boolean function can be obtained by repeated composition from its elements.*

Proposition 4.3 *The set* $\{\wedge, \vee, \neg\}$ *forms a complete basis; in other words, every Boolean function can be represented by a Boolean expression using only these connectives.*

The proof can be found in all elementary introductions to propositional logic. Note that since $\vee$ can be expressed using $\{\wedge, \neg\}$, this latter set is also a complete basis (and so is $\{\vee, \neg\}$).

From now on, under a ***Boolean expression*** (formula), we mean an expression built up from elements of some given complete basis. If we do not mention the basis then the complete basis $\{\wedge, \neg\}$ will be meant.

In general, one and the same Boolean function can be expressed in many ways as a Boolean expression. Given such an expression, it is easy to compute the value of the function. However, most Boolean functions can still be expressed only by very large Boolean expression (see Exercise 4.2-4).

4.2.2. Circuits

A Boolean expression is sometimes large since when writing it, there is no possibility for *reusing partial results*. (For example, in the expression

$$((x \vee y \vee z) \wedge u) \vee (\neg(x \vee y \vee z) \wedge v),$$

the part $x \vee y \vee z$ occurs twice.) This deficiency is corrected by the following more general formalism.

A Boolean circuit is essentially an acyclic directed graph, each of whose nodes computes a Boolean function (from some complete basis) of the bits coming into it on its input edges, and sends out the result on its output edges (see Figure 4.2). Let us give a formal definition.

Definition 4.4 *Let* Q *be a complete basis of Boolean functions. For an integer* N *let* $V = \{1, \dots, N\}$ *be a set of* ***nodes****. A* ***Boolean circuit*** *over* Q *is given by the following tuple:*

$$\mathcal{N} = (V, \{\, k_v : v \in V \,\}, \{\, \arg_j(v) : v \in V; j = 1, \dots, k_v \,\}, \{\, b_v : v \in V \,\}) \,. \qquad (4.10)$$

For every node v *there is a natural number* k_v *showing its number of* ***inputs****. The* ***sources****, nodes* v *with* $k_v = 0$*, are called* ***input nodes****: we will denote them, in increasing order, as*

$$\mathrm{inp}_i \quad (i = 1, \dots, n) \,.$$

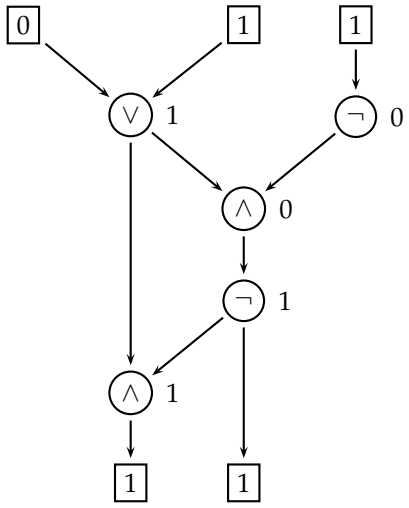


Figure 4.2. The assignment (values on nodes, configuration) gets propagated through all the gates. This is the "computation".

To each non-input node v a Boolean function

$$b_v(y_1, \ldots, y_{k_v})$$

from the complete basis Q is assigned: it is called the ***gate*** *of node v. It has as many arguments as the number of entering edges. The* ***sinks*** *of the graph, nodes without outgoing edges, will be called* ***output nodes****: they can be denoted by*

$$\mathrm{out}_i \quad (i = 1, \ldots, m) \; .$$

(Our Boolean circuits will mostly have just a single output node.) To every non-input node v and every $j = 1, \ldots, k_v$ belongs a node $\arg_j(v) \in V$ (the node sending the value of input variable y_j of the gate of v). The circuit defines a graph $G = (V, E)$ whose set of edges is

$$E = \{\, (\arg_j(v), v) : v \in V,\ j = 1, \ldots, k_v \,\} \; .$$

We require $\arg_j(v) < v$ for each j, v (we identified the with the natural numbers $1, \ldots, N$): this implies that the graph G is acyclic. The ***size***

$$|\mathcal{N}|$$

of the circuit $\mathcal{N}$ is the number of nodes. The ***depth*** *of a node v is the maximal length of directed paths leading from an input node to v. The* ***depth*** *of a circuit is the maximum depth of its output nodes.*

Definition 4.5 *An* ***input assignment****, or* ***input configuration*** *to our circuit*

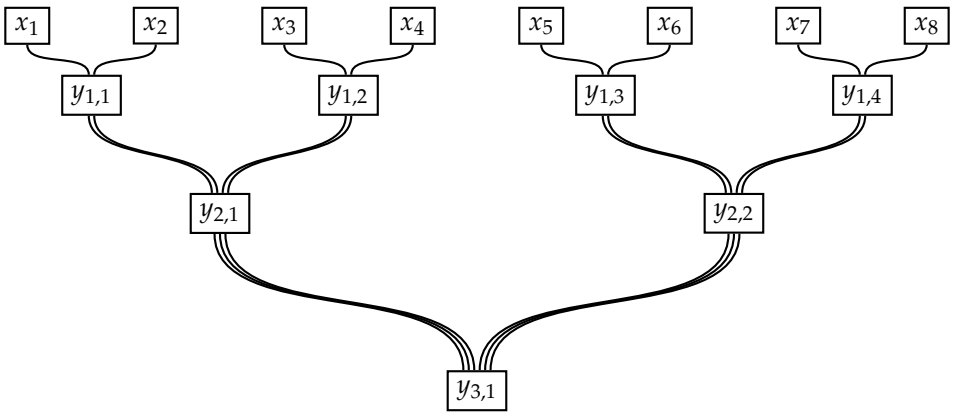


Figure 4.3. Naive parallel addition.

$\mathcal{N}$ is a vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ with $x_i \in \{0,1\}$ giving value x_i to node inp_i:

$$\mathrm{val}_{\boldsymbol{x}}(v) = y_v(\boldsymbol{x}) = x_i$$

*for $v = \mathrm{inp}_i$, $i = 1, \ldots, n$. The function $y_v(\boldsymbol{x})$ can be extended to a unique **configuration** $v \mapsto y_v(\boldsymbol{x})$ on all other nodes of the circuit as follows. If gate b_v has k arguments then*

$$y_v = b_v(y_{\mathrm{arg}_1(v)}, \ldots, y_{\mathrm{arg}_k(v)}) \; . \tag{4.11}$$

*For example, if $b_v(x,y) = x \wedge y$, and $u_j = \mathrm{arg}_j(v)$ $(j = 1,2)$ are the input nodes to v then $y_v = y_{u_1} \wedge y_{u_2}$. The process of extending the configuration by the above equation is also called the **computation** of the circuit. The vector of the values $y_{\mathrm{out}_i}(\boldsymbol{x})$ for $i = 1, \ldots, m$ is the **result** of the computation. We say that the Boolean circuit **computes** the vector function*

$$\boldsymbol{x} \mapsto (y_{\mathrm{out}_1}(\boldsymbol{x}), \ldots, y_{\mathrm{out}_m}(\boldsymbol{x})) \; .$$

*The assignment procedure can be performed in **stages**: in stage t, all nodes of depth t receive their values.*

We assign values to the edges as well: the value assigned to an edge is the one assigned to its start node.

4.2.3. Fast addition by a Boolean circuit

The depth of a Boolean circuit can be viewed as the shortest time it takes to compute the output vector from the input vector by this circuit. Az an example application of Boolean circuits, let us develop a circuit that computes the sum of its input bits very fast. We will need this result later in the present chapter for error-correcting purposes.

Definition 4.6 *We will say that a Boolean circuit computes a* ***near-majority*** *if it outputs a bit y with the following property: if $3/4$ of all input bits is equal to b then $y = b$.*

The depth of our circuit is clearly $\Omega(\log n)$, since the output must have a path to the majority of inputs. In order to compute the majority, we will also solve the task of summing the input bits.

Theorem 4.4

(a) *Over the complete basis consisting of the set of all 3-argument Boolean functions, for each n there is a Boolean circuit of input size n and depth $\leq 3\log(n+1)$ whose output vector represents the sum of the input bits as a binary number.*

(b) *Over this same complete basis, for each n there is a Boolean circuit of input size n and depth $\leq 2\log(n+1)$ computing a near-majority.*

Proof. First we prove (a). For simplicity, assume $n = 2^k - 1$: if n is not of this form, we may add some fake inputs. The naive approach would be proceed according to Figure 4.3: to first compute $y_{1,1} = x_1 + x_2$, $y_{1,2} = x_3 + x_4$, $\ldots$, $y_{1,2^{k-1}} = x_{2^k-1} + x_{2^k}$. Then, to compute $y_{2,1} = y_{1,1} + y_{1,2}$, $y_{2,2} = y_{1,3} + y_{1,4}$, and so on. Then $y_{k,1} = x_1 + \cdots + x_{2^k}$ will indeed be computed in k stages.

It is somewhat troublesome that $y_{i,j}$ here is a number, not a bit, and therefore must be represented by a *bit vector*, that is by group of nodes in the circuit, not just by a single node. However, the general addition operation

$$y_{i+1,j} = y_{i,2j-1} + y_{i,2j} \ ,$$

when performed in the naive way, will typically take more than a constant number of steps: the numbers $y_{i,j}$ have length up to $i+1$ and therefore the addition may add i to the depth, bringing the total depth to $1 + 2 + \cdots + k = \Omega(k^2)$.

The following observation helps to decrease the depth. Let a, b, c be three numbers in binary notation: for example, $a = \sum_{i=0}^{k} a_i 2^i$. There are simple parallel formulas to represent the sum of these three numbers as the sum of two others, that is to compute $a + b + c = d + e$ where d, e are numbers also in binary notation:

$$\begin{aligned} d_i &= a_i + b_i + c_i \bmod 2 \ , \\ e_{i+1} &= \lfloor (a_i + b_i + c_i)/2 \rfloor \ . \end{aligned} \tag{4.12}$$

Since both formulas are computed by a single 3-argument gate, 3 numbers can be reduced to 2 (while preserving the sum) in a single parallel computation step. Two such steps reduce 4 numbers to 2. In $2(k-1)$ steps therefore they reduce a sum of 2^k terms to a sum of 2 numbers of length $\leq k$. Adding these two numbers in the regular way increases the depth by k: we found that 2^k bits can be be added in $3k-2$ steps.

To prove (b), construct the circuit as in the proof of (a), but without the last addition: the output is two k-bit numbers whose sum we are interested in. The highest-order nonzero bit of these numbers is at some position $< k$. If the sum is

more than 2^{k-1} then one these numbers has a nonzero bit at position $(k-1)$ or $(k-2)$. We can determine this in two applications of 3-input gates. ■

Exercises

4.2-1 Show that $\{1, \oplus, \wedge\}$ is a complete basis.

4.2-2 Show that the function $x \,\mathrm{NOR}\, y = \neg(x \vee y)$ forms a complete basis by itself.

4.2-3 Let us fix the complete basis $\{\wedge, \neg\}$. Prove Proposition 4.3 (or look up its proof in a textbook). Use it to give an upper bound for an arbitrary Boolean function f of n variables, on:

(a) the smallest size of a Boolean expression for f;

(b) the smallest size of a Boolean circuit for f;

(c) the smallest depth of a Boolean circuit for f;

4.2-4 Show that for every n there is a Boolean function f of n variables such that every Boolean circuit in the complete basis $\{\wedge, \neg\}$ computing f contains $\Omega(2^n/n)$ nodes. *Hint.* For a constant $c > 0$, upperbound the number of Boolean circuits with at most $c2^n/n$ nodes and compare it with the number of Boolean functions over n variables.]

4.2-5 Consider a circuit $\mathcal{M}_3^r$ with 3^r inputs, whose single output bit is computed from the inputs by r levels of 3-input majority gates. Show that there is an input vector $\boldsymbol{x}$ which is 1 in only $n^{1/\log 3}$ positions but with which $\mathcal{M}_3^r$ outputs 1. Thus a small minority of the inputs, when cleverly arranged, can command the result of this circuit.

4.3. Expensive fault-tolerance in Boolean circuits

Let $\mathcal{N}$ be a Boolean circuit as given in Definition 4.4. When noise is allowed then the values

$$y_v = \mathrm{val}_{\boldsymbol{x}}(v)$$

will not be determined by the formula (4.11) anymore. Instead, they will be random variables Y_v. The random assignment $(Y_v : v \in V)$ will be called a ***random configuration***.

Definition 4.7 *At vertex v, let*

$$Z_v = b_v(Y_{\arg_1(v)}, \ldots, Y_{\arg_k(v)}) \oplus Y_v \ . \tag{4.13}$$

*In other words, $Z_v = 1$ if gate Y_v is not equal to the value computed by the noise-free gate b_v from its inputs $Y_{\arg_j(v)}$. (See Figure 4.4.) The set of vertices where Z_v is non-zero is the set of **faults**.*

*Let us call the difference $\mathrm{val}_{\boldsymbol{x}}(v) \oplus Y_v$ the **deviation** at node v.*

Let us impose conditions on the kind of noise that will be allowed. Each fault should occur only with probability at most ϵ, two specific faults should only occur with probability at most ϵ^2, and so on.

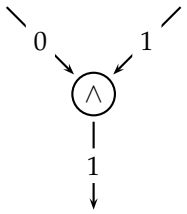


Figure 4.4. Failure at a gate.

Definition 4.8 *For an $\epsilon > 0$, let us say that the random configuration $(Y_v : v \in V)$ is ϵ-**admissible** if*

(a) $Y_{\mathrm{inp}(i)} = x_i$ *for* $i = 1, \ldots, n$.

(b) *For every set C of non-input nodes, we have*

$$\mathbf{P}[\, Z_v = 1 \text{ for all } v \in C \,] \leq \epsilon^{|C|}. \tag{4.14}$$

In other words, in an ϵ-admissible random configuration, the probability of having faults at k different specific gates is at most ϵ^k. This is how we require that not only is the fault probability low but also, faults do not "conspire". The admissibility condition is satisfied if faults occur independently with probability $\leq \epsilon$.

Our goal is to build a circuit that will work correctly, with high probability, despite the ever-present noise: in other words, in which errors *do not accumulate*. This concept is formalized below.

Definition 4.9 *We say that the circuit $\mathcal{N}$ with output node w is (ϵ, δ)-**resilient** if for all inputs $\boldsymbol{x}$, all ϵ-admissible configurations $\mathbf{Y}$, we have $\mathbf{P}[\, Y_w \neq \mathrm{val}_{\boldsymbol{x}}(w) \,] \leq \delta$.*

Let us explore this concept. There is no (ϵ, δ)-resilient circuit with $\delta < \epsilon$, since even the last gate can fail with probability ϵ. So, let us, a little more generously, allow $\delta > 2\epsilon$. Clearly, for each circuit $\mathcal{N}$ and for each $\delta > 0$ we can choose ϵ small enough so that $\mathcal{N}$ is (ϵ, δ)-resilient. But this is not what we are after: hopefully, one does not need more reliable gates every time one builds a larger circuit. So, we hope to find a function

$$F(N, \delta)$$

and an $\epsilon_0 > 0$ with the property that for all $\epsilon < \epsilon_0$, $\delta \geq 2\epsilon$, every Boolean circuit $\mathcal{N}$ of size N there is some (ϵ, δ)-resilient circuit $\mathcal{N}'$ of size $F(N, \delta)$ computing the same function as $\mathcal{N}$. If we achieve this then we can say that we prevented the accumulation of errors. Of course, we want to make $F(N, \delta)$ relatively small, and ϵ_0 large (allowing more noise). The function $F(N, \delta)/N$ can be called the ***redundancy***: the factor by which we need to increase the size of the circuit to make it resilient. Note that the problem is nontrivial even with, say, $\delta = 1/3$. Unless the accumulation of errors is prevented we will lose gradually all information about the desired output, and no

$\delta < 1/2$ could be guaranteed.

How can we correct errors? A simple idea is this: do "everything" 3 times and then continue with the result obtained by majority vote.

Definition 4.10 *For odd natural number d, a d-input majority gate is a Boolean function that outputs the value equal to the majority of its inputs.*

Note that a d-input majority can be computed using $O(d)$ gates of type AND and NOT.

Why should majority voting help? The following *informal discussion* helps understanding the benefits and pitfalls. Suppose for a moment that the output is a single bit. If the probability of each of the three independently computed results failing is δ then the probability that at least 2 of them fails is bounded by $3\delta^2$. Since the majority vote itself can fail with some probability ϵ the total probability of failure is bounded by $3\delta^2 + \epsilon$. We decrease the probability δ of failure, provided the condition $3\delta^2 + \epsilon < \delta$ holds.

We found that if δ is small, then repetition and majority vote can "make it" smaller. Of course, in order to keep the error probability from accumulating, we would have to perform this majority operation repeatedly. Suppose, for example, that our computation has t stages. Our bound on the probability of faulty output after stage i is δ_i. We plan to perform the majority operation after each stage i. Let us perform stage i three times. The probability of failure is now bounded by

$$\delta_{i+1} = \delta_i + 3\delta^2 + \epsilon. \tag{4.15}$$

Here, the error probabilities of the different stages accumulate, and even if $3\delta^2 + \epsilon < \delta$ we only get a bound $\delta_t < (t-1)\delta$. So, this strategy will not work for arbitrarily large computations.

Here is a somewhat mad idea to avoid accumulation: repeat *everything* before the end of stage i three times, not only stage i itself. In this case, the growing bound (4.15) would be replaced with

$$\delta_{i+1} = 3(\delta_i + \delta)^2 + \epsilon\ .$$

Now if $\delta_i < \delta$ and $12\delta^2 + \epsilon < \delta$ then also $\delta_{i+1} < \delta$, so errors do not accumulate. But we paid an enormous price: the fault-tolerant version of the computation reaching stage $(i+1)$ is 3 times larger than the one reaching stage i. To make t stages fault-tolerant this way will cost a factor of 3^t in size. This way, the function $F(N, \delta)$ introduced above may become exponential in N.

The theorem below formalizes the above discussion.

Theorem 4.5 *Let R be a finite and complete basis for Boolean functions. If $2\epsilon \leq \delta \leq 0.01$ then every function can be computed by an (ϵ, δ)-resilient circuit over R.*

Proof. For simplicity, we will prove the result for a complete basis that contains the three-argument majority function and contains not functions with more than three arguments. We also assume that faults occur independently.

Let $\mathcal{N}$ be a noise-free circuit of depth t computing function f. We will prove that there is an (ϵ, δ)-resilient circuit $\mathcal{N}'$ of depth $2t$ computing f. The proof is by

induction on t. The sufficient conditions on ϵ and δ will emerge from the proof.

The statement is certainly true for $t = 1$, so suppose $t > 1$. Let g be the output gate of the circuit $\mathcal{N}$, then $f(\boldsymbol{x}) = g(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), f_3(\boldsymbol{x}))$. The subcircuits $\mathcal{N}_i$ computing the functions f_i have depth $\leq t-1$. By the inductive assumption, there exist (ϵ, δ)-resilient circuits $\mathcal{N}'_i$ of depth $\leq 2t-2$ that compute f_i. Let $\mathcal{M}$ be a new circuit containing copies of the circuits $\mathcal{N}'_i$ (with the corresponding input nodes merged), with a new node in which $f(\boldsymbol{x})$ is computed as g is applied to the outputs of $\mathcal{N}'_i$. Then the probability of error of $\mathcal{M}$ is at most $3\delta + \epsilon < 4\delta$ if $\epsilon < \delta$ since each circuit $\mathcal{N}'_i$ can err with probability δ and the node with gate g can fail with probability ϵ.

Let us now form $\mathcal{N}'$ by taking three copies of $\mathcal{M}$ (with the inputs merged) and adding a new node computing the majority of the outputs of these three copies. The error probability of $\mathcal{N}'$ is at most $3(4\delta)^2 + \epsilon = 48\delta^2 + \epsilon$. Indeed, error will be due to either a fault at the majority gate or an error in at least two of the three independent copies of $\mathcal{M}$. So under condition

$$48\delta^2 + \epsilon \leq \delta \,, \tag{4.16}$$

the circuit $\mathcal{N}'$ is (ϵ, δ)-resilient. This condition will be satisfied by $2\epsilon \leq \delta \leq 0.01$. ■

The circuit $\mathcal{N}'$ constructed in the proof above is at least 3^t times larger than $\mathcal{N}$. So, the redundancy is enormous. Fortunately, we will see a much more economical solution. But there are interesting circuits with small depth, for which the 3^t factor is not extravagant.

Theorem 4.6 *Over the complete basis consisting of all 3-argument Boolean functions, for all sufficiently small $\epsilon > 0$, if $2\epsilon \leq \delta \leq 0.01$ then for each n there is an (ϵ, δ)-resilient Boolean circuit of input size n, depth $\leq 4\log(n+1)$ and size $(n+1)^7$ outputting a near-majority (as given in Definition 4.6).*

Proof. Apply Theorem 4.5 to the circuit from part (a) of Theorem 4.4: it gives a new, $4\log(n+1)$-deep (ϵ, δ)-resilient circuit computing a near-majority. The size of any such circuit with 3-input gates is at most $3^{4\log(n+1)} = (n+1)^{4\log 3} < (n+1)^7$. ■

Exercises

4.3-1 Exercise 4.2-5 suggests that the iterated majority vote $\mathcal{M}_3^r$ is not safe against manipulation. However, it works very well under some circumstances. Let the input to $\mathcal{M}_3^r$ be a vector $\boldsymbol{X} = (X_1, \ldots, X_n)$ of independent Boolean random variables with $\mathbf{P}[\,X_i = 1\,] = p < 1/6$. Denote the (random) output bit of the circuit by Z. Assuming that our majority gates can fail with probability $\leq \epsilon \leq p/2$ independently, prove

$$\mathbf{P}[\,Z = 1\,] \leq \max\{10\epsilon,\ 0.3(p/0.3)^{2^k}\}\,.$$

Hint. Define $g(p) = \epsilon + 3p^2$, $g_0(p) = p$, $g_{i+1}(p) = g(g_i(p))$, and prove $\mathbf{P}[\,Z = 1\,] \leq g_r(p)$.]

4.3-2 We say that a circuit $\mathcal{N}$ computes the function $f(x_1, \ldots, x_n)$ in an (ϵ, δ)-***input-robust*** way, if the following holds: For any input vector $\boldsymbol{x} = (x_1, \ldots, x_n)$, for any vector $\boldsymbol{X} = (X_1, \ldots, X_n)$ of independent Boolean random variables "perturbing it" in the sense $\mathbf{P}[\,X_i \neq x_i\,] \leq \epsilon$, for the output Y of circuit $\mathcal{N}$ on input $\boldsymbol{X}$ we have

$\mathbf{P}[Y = f(\boldsymbol{x})] \geq 1 - \delta$. Show that if the function $x_1 \oplus \cdots \oplus x_n$ is computable on an $(\epsilon, 1/4)$-input-robust circuit then $\epsilon \leq 1/n$.

4.4. Safeguarding intermediate results

In this section, we will see ways to introduce fault-tolerance that scale up better. Namely, we will show:

Theorem 4.7 *There are constants R_0, ϵ_0 such that for*

$$F(n, \delta) = N \log(n/\delta) \ ,$$

for all $\epsilon < \epsilon_0$, $\delta \geq 3\epsilon$, for every deterministic computation of size N there is an (ϵ, δ)-resilient computation of size $R_0 F(N, \delta)$ with the same result.

Let us introduce a concept that will simplify the error analysis of our circuits, making it independent of the input vector x.

Definition 4.11 *In a Boolean circuit $\mathcal{N}$, let us call a majority gate at a node v a **correcting majority gate** if for every input vector $\boldsymbol{x}$ of $\mathcal{N}$, all input wires of node v have the same value. Consider a computation of such a circuit $\mathcal{N}$. This computation will make some nodes and wires of $\mathcal{N}$ **tainted**. We define taintedness by the following rules:*

- *The input nodes are untainted.*
- *If a node is tainted then all of its output wires are tainted.*
- *A correcting majority gate is tainted if either it fails or a majority of its inputs are tainted.*
- *Any other gate is tainted if either it fails or one of its inputs is tainted.*

Clearly, if for all ϵ-admissible random configurations the output is tainted with probability $\leq \delta$ then the circuit is (ϵ, δ)-resilient.

4.4.1. Cables

So far, we have only made use of redundancy idea (ii) of the introduction to the present chapter: repeating computation steps. Let us now try to use idea (i) (keeping information in redundant form) in Boolean circuits. To protect information traveling from gate to gate, we replace each wire of the noiseless circuit by a "cable" of k wires (where k will be chosen appropriately). Each wire within the cable is supposed to carry the same bit of information, and we hope that a majority will carry this bit even if some of the wires fail.

Definition 4.12 *In a Boolean circuit $\mathcal{N}'$, a certain set of edges is allowed to be*

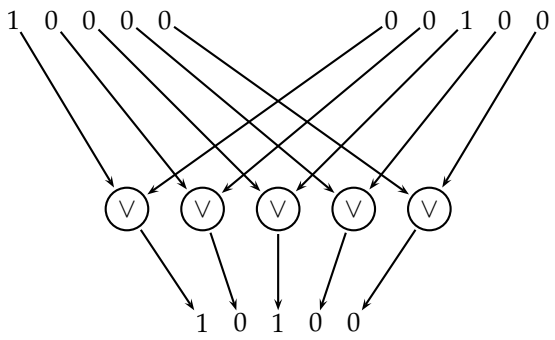


Figure 4.5. An executive organ.

called a ***cable*** *if in a noise-free computation of this circuit, each edge carries the same Boolean value. The* ***width*** *of the cable is its number of elements. Let us fix an appropriate constant threshold ϑ. Consider any possible computation of the noisy version of the circuit $\mathcal{N}'$, and a cable of width k in $\mathcal{N}'$. This cable will be called* ***ϑ-safe*** *if at most ϑk of its wires are tainted.*

Let us take a Boolean circuit $\mathcal{N}$ that we want to make resilient. As we replace wires of $\mathcal{N}$ with cables of $\mathcal{N}'$ containing k wires each, we will replace each noiseless 2-argument gate at a node v by a module called the ***executive organ*** of k gates, which for each $i = 1, \ldots, k$, passes the ith wire both incoming cables into the ith node of the organ. Each of these nodes contains a gate of one and the same type b_v. The wires emerging from these nodes form the ***output cable*** of the executive organ.

The number of tainted wires in this output cable may become too high: indeed, if there were ϑk tainted wires in the x cable and also in the y cable then there could be as many as $2\vartheta k$ such wires in the $g(x, y)$ cable (not even counting the possible new taints added by faults in the executive organ). The crucial part of the construction is to attach to the executive organ a so-called ***restoring organ***: a module intended to decrease the taint in a cable.

4.4.2. Compressors

How to build a restoring organ? Keeping in mind that this organ itself must also work in noise, one solution is to build (for an approriate δ') a special (ϵ, δ')-resilient circuit that computes the near-majority of its k inputs in k independent copies. Theorem 4.6 provides a circuit of size $k(k+1)^7$ to do this.

It turns out that, at least asymptotically, there is a better solution. We will look for a *very simple* restoring organ: one whose own noise we can analyse easily. What could be simpler than a circuit having only *one level* of gates? We fix an odd positive integer constant d (for example, $d = 3$). Each gate of our organ will be a d-input majority gate.

Definition 4.13 *A* ***multigraph*** *is a graph in which between any two vertices there may be several edges, not just 0 or 1. Let us call a bipartite multigraph with k inputs*

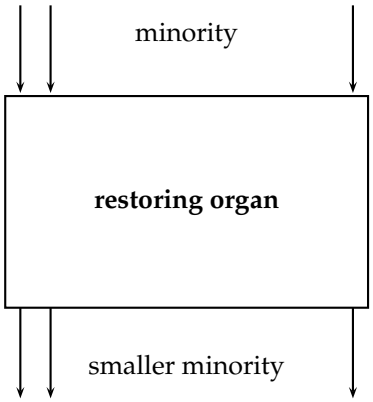


Figure 4.6. A restoring organ.

*and k outputs, d-**half-regular**, if each output node has degree d. Such a graph is a (d, α, γ, k)-**compressor** if it has the following property: for every set E of at most $\leq \alpha k$ inputs, the number of those output points connected to at least $d/2$ elements of E (with multiplicity) is at most $\gamma \alpha k$.*

The compressor property is interesting generally when $\gamma < 1$. For example, in an $(5, 0.1, 0.5, k)$-compressor the outputs have degree 5, and the majority operation in these nodes decreases every error set confined to 10% of all input to just 5% of all outputs. A compressor with the right parameters could serve as our restoring organ: it decreases a minority to a smaller minority and may in this way restore the safety of a cable. But, are there compressors?

Theorem 4.8 *For all $\gamma < 1$, all integers d with*

$$1 < \gamma(d-1)/2 \,, \tag{4.17}$$

there is an α such that for all integer $k > 0$ there is a (d, α, γ, k)-compressor.

Note that for $d = 3$, the theorem does not guarantee a compressor with $\gamma < 1$.

Proof. We will not give an explicit construction for the multigraph, we will just show that it exists. We will select a d-half-regular multigraph randomly (each such multigraph with the same probability), and show that it will be a (d, α, γ, k)-compressor with positive probability. This proof method is called the ***probabilistic method***. Let

$$s = \lfloor d/2 \rfloor \,.$$

Our construction will be somewhat more general, allowing $k' \neq k$ outputs. Let us generate a random bipartite d-half-regular multigraph with k inputs and k' outputs in the following way. To each output, we draw edges from d random input nodes chosen independently and with uniform distribution over all inputs. Let A be an input set of size αk, let v be an output node and let E_v be the event that v has $s+1$

or more edges from A. Then we have

$$\mathbf{P}(E_v) \leq \binom{d}{s+1}\alpha^{s+1} = \binom{d}{s}\alpha^{s+1} =: p\ .$$

On the average (in expected value), the event E_v will occur for pk' different output nodes v. For an input set A, let F_A be the event that the set of nodes v for which E_v holds has size $> \gamma\alpha k'$. By inequality (4.6) we have

$$\mathbf{P}(F_A) \leq \left(\frac{ep}{\gamma\alpha}\right)^{k'\gamma\alpha}\ .$$

The number M of sets A of inputs with $\leq \alpha k$ elements is, using inequality (4.7),

$$M \leq \sum_{i\leq\alpha k}\binom{k}{i} \leq \left(\frac{e}{\alpha}\right)^{\alpha k}\ .$$

The probability that our random graph is not a compressor is at most as large as the probability that there is at least one input set A for which event F_A holds. This can be bounded by

$$M\cdot\mathbf{P}(F_A) \leq e^{-\alpha Dk'}$$

where

$$D = -(\gamma s - k/k')\ln\alpha - \gamma\big(\ln\tbinom{d}{s} - \ln\gamma + 1\big) - k/k'.$$

As we decrease α the first term of this expression dominates. Its coefficient is positive according to the assumption (4.17). We will have $D > 0$ if

$$\alpha < \exp\left(-\frac{\gamma\big(\ln\binom{d}{s} - \ln\gamma + 1\big) + k/k'}{\gamma s - k/k'}\right)\ .$$

■

Example 4.4 Choosing $\gamma = 0.4$, $d = 7$, the value $\alpha = 10^{-7}$ will work.

We turn a (d, α, γ, k)-compressor into a restoring organ $\mathcal{R}$, by placing d-input majority gates into its outputs. If the majority elements sometimes fail then the output of $\mathcal{R}$ is random. Assume that at most αk inputs of $\mathcal{R}$ are tainted. Then $(\gamma+\rho)\alpha k$ outputs can only be tainted if $\alpha\rho k$ majority gates fail. Let

$$p_{\mathcal{R}}$$

be the probability of this event. Assuming that the gates of $\mathcal{R}$ fail independently with probability $\leq \epsilon$, inequality (4.6) gives

$$p_{\mathcal{R}} \leq \left(\frac{e\epsilon}{\alpha\rho}\right)^{\alpha\rho k}\ . \tag{4.18}$$

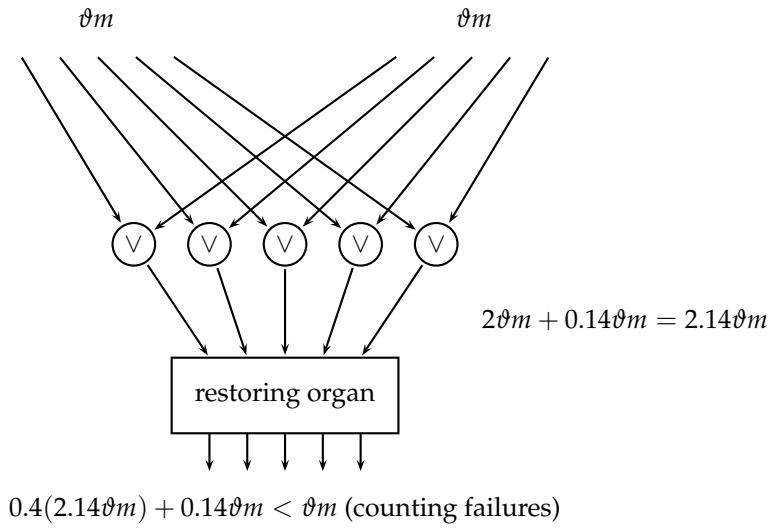


Figure 4.7. An executive organ followed by a restoring organ.

Example 4.5 Choose $\gamma = 0.4$, $d = 7$, $\alpha = 10^{-7}$ as in Example 4.4, further $\rho = 0.14$ (this will satisfy the inequality (4.19) needed later). With $\epsilon = 10^{-9}$, we get $p_{\mathcal{R}} \leq e^{-10^{-8}k}$.

The attractively small degree $d = 7$ led to an extremely unattractive probability bound on the failure of the whole compressor. This bound does decrease exponentially with cable width k, but only an extremely large k would make it small.

Example 4.6 Choosing again $\gamma = 0.4$, but $d = 41$ (voting in each gate of the compressor over 41 wires instead of 7), leads to somewhat more realistic results. This choice allows $\alpha = 0.15$. With $\rho = 0.14$, $\epsilon = 10^{-9}$ again, we get $p_{\mathcal{R}} \leq e^{-0.32k}$.

These numbers look less frightening, but we will still need many scores of wires in the cable to drive down the probability of compression failure. And although in practice our computing components fail with frequency much less than 10^{-9}, we may want to look at the largest ϵ that still can be tolerated.

4.4.3. Propagating safety

Compressors allow us to construct a reliable Boolean circuit all of whose cables are safe.

Definition 4.14 *Given a Boolean circuit $\mathcal{N}$ with a single bit of output (for simplicity), a cable width k and a Boolean circuit $\mathcal{R}$ with k inputs and k outputs, let*

$$\mathcal{N}' = \mathrm{Cab}(\mathcal{N}, \mathcal{R})$$

be the Boolean circuit that we obtain as follows. The input nodes of $\mathcal{N}'$ are the same

as those of $\mathcal{N}$. We replace each wire of $\mathcal{N}$ with a cable of width k, and each gate of $\mathcal{N}$ with an executive organ followed by a restoring organ that is a copy of the circuit $\mathcal{R}$. The new circuit has k outputs: the outputs of the restoring organ of $\mathcal{N}'$ belonging to the last gate of $\mathcal{N}$.

In noise-free computations, on every input, the output of $\mathcal{N}'$ is the same as the output of $\mathcal{N}$, but in k identical copies.

Lemma 4.9 *There are constants $d, \epsilon_0, \vartheta, \rho > 0$ and for every cable width k a circuit $\mathcal{R}$ of size $2k$ and gate size $\leq d$ with the following property. For every Boolean circuit $\mathcal{N}$ of gate size ≤ 2 and number of nodes N, for every $\epsilon < \epsilon_0$, for every ϵ-admissible configuration of $\mathcal{N}' = \mathrm{Cab}(\mathcal{N}, \mathcal{R})$, the probability that not every cable of $\mathcal{N}'$ is ϑ-safe is $< 2N(\frac{e\epsilon}{\vartheta\rho})^{\vartheta\rho k}$.*

Proof. We know that there are d, α and $\gamma < 1/2$ with the property that for all k a (d, α, γ, k)-compressor exists. Let ρ be chosen to satisfy

$$\gamma(2+\rho) + \rho \leq 1, \tag{4.19}$$

and define

$$\vartheta = \alpha/(2+\rho). \tag{4.20}$$

Let $\mathcal{R}$ be a restoring organ built from a (d, α, γ, k)-compressor. Consider a gate v of circuit $\mathcal{N}$, and the corresponding executive organ and restoring organ in $\mathcal{N}'$. Let us estimate the probability of the event E_v that the input cables of this combined organ are ϑ-safe but its output cable is not. Assume that the two incoming cables are safe: then at most $2\vartheta k$ of the outputs of the executive organ are tainted due to the incoming cables: new taint can still occur due to failures. Let E_{v1} be the event that the executive organ taints at least $\rho\vartheta k$ more of these outputs. Then $\mathbf{P}(E_{v1}) \leq (\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$, using the estimate (4.18). The outputs of the executive organ are the inputs of the restoring organ. If no more than $(2+\rho)\vartheta k = \alpha k$ of these are tainted then, in case the organ operates perfectly, it would decrease the number of tainted wires to $\gamma(2+\rho)\vartheta k$. Let E_{v2} be the event that the restoring organ taints an additional $\rho\vartheta k$ of these wires. Then again, $\mathbf{P}(E_{v2}) \leq (\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$. If neither E_{v1} nor E_{v2} occur then at most $\gamma(2+\rho)\vartheta k + \rho\vartheta k \leq \vartheta k$ (see (4.19)) tainted wires emerge from the restoring organ, so the outgoing cable is safe. Therefore $E_v \subset E_{v1} \cup E_{v2}$ and hence $\mathbf{P}(E_v) \leq 2(\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$.

Let $V = \{1, \ldots, N\}$ be the nodes of the circuit $\mathcal{N}$. Since the incoming cables of the whole circuit $\mathcal{N}'$ are safe, the event that there is some cable that is not safe is contained in $E_1 \cup E_2 \cup \cdots \cup E_N$; hence the probability is bounded by $2N(\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$. ■

4.4.4. Endgame

Proof of Theorem 4.7. We will prove the theorem only for the case when our computation is a Boolean circuit with a single bit of output. The generalization with more bits of output is straightforward. The proof of Lemma 4.9 gives us a circuit $\mathcal{N}'$ whose output cable is safe except for an event of probability $< 2N(\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$. Let

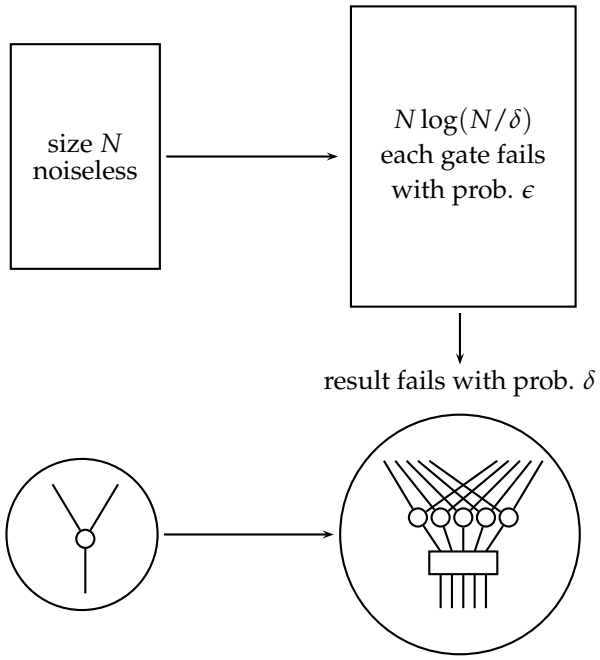


Figure 4.8. Reliable circuit from a fault-free circuit.

us choose k in such a way that this becomes $\leq \delta/3$:

$$k \geq \frac{\log(6N/\delta)}{\rho\vartheta \log \frac{\rho\vartheta}{e\epsilon_0}} \ . \tag{4.21}$$

It remains to add a little circuit to this output cable to extract from it the majority reliably. This can be done using Theorem 4.6, adding a small extra circuit of size $(k+1)^7$ that can be called the ***coda*** to $\mathcal{N}'$. Let us call the resulting circuit $\mathcal{N}''$.

The probability that the output cable is unsafe is $< \delta/3$. The probability that the output cable is safe but the "coda" circuit fails is bounded by 2ϵ. So, the probability that $\mathcal{N}''$ fails is $\leq 2\epsilon + \delta/3 \leq \delta$, by the assumption $\delta \geq 3\epsilon$.

Let us estimate the size of $\mathcal{N}''$. By (4.21), we can choose cable width $k = O(\log(N/\delta))$. We have $|\mathcal{N}'| \leq 2kN$, hence

$$|\mathcal{N}''| \leq 2kN + (k+1)^7 = O(N \log(N/\delta)).$$

■

Example 4.7 Take the constants of Example 4.6, with ϑ defined in equation (4.20): then $\epsilon_0 = 10^{-9}$, $d = 41$, $\gamma = 0.4$, $\rho = 0.14$, $\alpha = 0.15$, $\vartheta = 0.07$, giving

$$\frac{1}{\rho\vartheta \ln \frac{\rho\vartheta}{e\epsilon_0}} \approx 6.75,$$

so making k as small as possible (ignoring that it must be integer), we get $k \approx 6.75 \ln(N/\delta)$. With $\delta = 10^{-8}$, $N = 10^{12}$ this allows $k = 323$. In addition to this truly unpleasant cable size, the size of the "coda" circuit is $(k+1)^7 \approx 4 \cdot 10^{17}$, which dominates the size of the rest of $\mathcal{N}''$ (though as $N \to \infty$ it becomes asymptotically negligible).

As Example 4.7 shows, the actual price in redundancy computable from the proof is unacceptable in practice. The redundancy $O(\lg(N/\delta))$ sounds good, since it is only logarithmic in the size of the computation, and by choosing a rather large majority gate (41 inputs), the factor 6.75 in the $O(\cdot)$ here also does not look bad; still, we do not expect the final price of reliability to be this high. How much can this redundancy improved by optimization or other methods? Problem *4-6* shows that in a slightly more restricted error model (all faults are independent and have the same probability), with more randomization, better constants can be achieved. Exercises 4.4-1, 4.4-2 and 4.4-5 are concerned with an improved construction for the "coda" circuit. Exercise 4.5-2 shows that the coda circuit can be omitted completely. But none of these improvements bring redundancy to acceptable level. Even aside from the discomfort caused by their random choice (this can be helped), concentrators themselves are rather large and unwieldy. The problem is probably with using circuits as a model for computation. There is no natural way to break up a general circuit into subunits of non-constant size in order to deal with the reliability problem in modular style.

4.4.5. The construction of compressors

This subsection is sketchier than the preceding ones, and assumes some knowledge of linear algebra.

We have shown that compressors exist. How expensive is it to find a (d, α, γ, k)-compressor, say, with $d = 41$, $\alpha = 0.15$, $\gamma = 0.4$, as in Example 4.6? In a deterministic algorithm, we could search through all the approximately d^k d-half-regular bipartite graphs. For each of these, we could check all possible input sets of size $\leq \alpha k$: as we know, their number is $\leq (e/\alpha)^{\alpha k} < 2^k$. The cost of checking each subset is $O(k)$, so the total number of operations is $O(k(2d)^k)$. Though this number is exponential in k, recall that in our error-correcting construction, $k = O(\log(N/\delta))$ for the size N of the noiseless circuit: therefore the total number of operations needed to find a compressor is polynomial in N.

The proof of Theorem 4.8 shows that a randomly chosen d-half-regular bipartite graph is a compressor with large probability. Therefore there is a faster, randomized algorithm for finding a compressor. Pick a random d-half-regular bipartite graph, check if it is a compressor: if it is not, repeat. We will be done in a constant expected number of repetititons. This is a faster algorithm, but is still exponential in k, since each checking takes $\Omega(k(e/\alpha)^{\alpha k})$ operations.

Is it possible to construct a compressor explicitly, avoiding any search that takes exponential time in k? The answer is yes. We will show here only, however, that the compressor property is implied by a certain property involving linear algebra, which can be checked in polynomial time. Certain explicitly constructed graphs are known that possess this property. These are generally sought after not so much for

their compressor property as for their *expander* property (see the section on reliable storage).

For vectors $\boldsymbol{v}, \boldsymbol{w}$, let $(\boldsymbol{v}, \boldsymbol{w})$ denote their inner product. A d-half-regular bipartite multigraph with $2k$ nodes can be defined by an ***incidence matrix*** $\boldsymbol{M} = (m_{ij})$, where m_{ij} is the number of edges connecting input j to output i. Let $\boldsymbol{e}$ be the vector $(1, 1, \ldots, 1)^T$. Then $\boldsymbol{Me} = d\boldsymbol{e}$, so $\boldsymbol{e}$ is an ***eigenvector*** of $\boldsymbol{M}$ with ***eigenvalue*** d. Moreover, d is the largest eigenvalue of $\boldsymbol{M}$. Indeed, denoting by $|\boldsymbol{x}|_1 = \sum_i |x_i|$ for any row vector $\boldsymbol{x} = (x_1, \ldots, x_k)$, we have $|\boldsymbol{xM}|_1 \leq |\boldsymbol{x}|_1$.

Theorem 4.10 *Let G be a multigraph defined by the matrix $\boldsymbol{M}$. For all $\gamma > 0$, and*

$$\mu < d\sqrt{\gamma}/2, \tag{4.22}$$

there is an $\alpha > 0$ such that if the second largest eigenvalue of the matrix $\boldsymbol{M}^T\boldsymbol{M}$ is μ^2 then G is a (d, α, γ, k)-compressor.

Proof. The matrix $\boldsymbol{M}^T\boldsymbol{M}$ has largest eigenvalue d^2. Since it is symmetric, it has a basis of orthogonal eigenvectors $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_k$ of unit length with corresponding non-negative eigenvalues

$$\lambda_1^2 \geq \cdots \geq \lambda_k^2$$

where $\lambda_1 = d$ and $\boldsymbol{e}_1 = \boldsymbol{e}/\sqrt{k}$. Recall that in the orthonormal basis $\{\boldsymbol{e}_i\}$, any vector $\boldsymbol{f}$ can be written as $\boldsymbol{f} = \sum_i (\boldsymbol{f}, \boldsymbol{e}_i)\boldsymbol{e}_i$. For an arbitrary vector $\boldsymbol{f}$, we can estimate $|\boldsymbol{Mf}|^2$ as follows.

$$\begin{aligned} |\boldsymbol{Mf}|^2 = (\boldsymbol{Mf}, \boldsymbol{Mf}) = (\boldsymbol{f}, \boldsymbol{M}^T\boldsymbol{Mf}) &= \sum_i \lambda_i^2 (\boldsymbol{f}, \boldsymbol{e}_i)^2 \\ &\leq d^2(\boldsymbol{f}, \boldsymbol{e}_1)^2 + \mu^2 \sum_{i>1} (\boldsymbol{f}, \boldsymbol{e}_i)^2 \leq d^2(\boldsymbol{f}, \boldsymbol{e}_1)^2 + \mu^2(\boldsymbol{f}, \boldsymbol{f}) \\ &= d^2(\boldsymbol{f}, \boldsymbol{e})^2/k + \mu^2(\boldsymbol{f}, \boldsymbol{f}). \end{aligned}$$

Let now $A \subset \{1, \ldots, k\}$ be a set of size αk and $\boldsymbol{f} = (f_1, \ldots, f_k)^T$ where $f_j = 1$ for $j \in A$ and 0 otherwise. Then, coordinate i of $\boldsymbol{Mf}$ counts the number d_i of edges coming from the set A to the node i. Also, $(\boldsymbol{f}, \boldsymbol{e}) = (\boldsymbol{f}, \boldsymbol{f}) = |A|$, the number of elements of A. We get

$$\sum_i d_i^2 = |\boldsymbol{Mf}|^2 \leq d^2(\boldsymbol{f}, \boldsymbol{e})^2/k + \mu^2(\boldsymbol{f}, \boldsymbol{f}) = d^2\alpha^2 k + \mu^2 \alpha k,$$

$$k^{-1} \sum_i (d_i/d)^2 \leq \alpha^2 + (\mu/d)^2 \alpha.$$

Suppose that there are $c\alpha k$ nodes i with $d_i > d/2$, then this says

$$c\alpha \leq 4(\mu/d)^2 \alpha + 4\alpha^2.$$

Since (4.22) implies $4(\mu/d)^2 < \gamma$, it follows that $\boldsymbol{M}$ is a $(d, \alpha, \gamma, k, k)$-compressor for small enough α. ■

It is actually sufficient to look for graphs with large k and $\mu/d < c < 1$ where d, c are constants. To see this, let us define the product of two bipartite multigraphs with $2k$ vertices by the multigraph belonging to the product of the corresponding matrices.

Suppose that $\boldsymbol{M}$ is symmetric: then its second largest eigenvalue is μ and the ratio of the two largest eigenvalues of $\boldsymbol{M}^r$ is $(\mu/d)^r$. Therefore using $\boldsymbol{M}^r$ for a sufficiently large r as our matrix, the condition (4.22) can be satisfied. Unfortunately, taking the power will increase the degree d, taking us probably even farther away from practical realizability.

We found that there is a construction of a compressor with the desired parameters as soon as we find multigraphs with arbitrarily large sizes $2k$, with symmetric matrices $\boldsymbol{M}_k$ and with a ratio of the two largest eigenvalues of $\boldsymbol{M}_k$ bounded by a constant $c < 1$ independent of k. There are various constructions of such multigraphs (see the references in the historical overview). The estimation of the desired eigenvalue quotient is never very simple.

Exercises

4.4-1 The proof of Theorem 4.7 uses a "coda" circuit of size $(k+1)^7$. Once we proved this theorem we could, of course, apply it to the computation of the final majority itself: this would reduce the size of the coda circuit to $O(k \log k)$. Try out this approach on the numerical examples considered above to see whether it results in a significant improvement.

4.4-2 The proof of Theorem 4.8 provided also bipartite graphs with the compressor property, with k inputs and $k' < 0.8k$ outputs. An idea to build a smaller "coda" circuit in the proof of Theorem 4.7 is to concatenate several such compressors, decreasing the number of cables in a geometric series. Explore this idea, keeping in mind, however, that as k decreases, the "exponential" error estimate in inequality (4.18) becomes weaker.

4.4-3 In a noisy Boolean circuit, let $F_v = 1$ if the gate at vertex v fails and 0 otherwise. Further, let $T_v = 1$ if v is tainted, and 0 otherwise. Suppose that the distribution of the random variables F_v does not depend on the Boolean input vector. Show that then the joint distribution of the random variables T_v is also independent of the input vector.

4.4-4 This exercise extends the result of Exercise 4.3-1 to random input vectors: it shows that if a random input vector has only a small number of errors, then the iterated majority vote $\mathcal{M}_3^r$ of Exercise 4.2-5 may still work for it, if we rearrange the input wires randomly. Let $k = 3^r$, and let $\boldsymbol{j} = (j_1, \ldots, j_k)$ be a vector of integers $j_i \in \{1, \ldots, k\}$. We define a Boolean circuit $C(\boldsymbol{j})$ as follows. This circuit takes input vector $\boldsymbol{x} = (x_1, \ldots, x_k)$, computes the vector $\boldsymbol{y} = (y_1, \ldots, y_k)$ where $y_i = x_{j_i}$ (in other words, just leads a wire from input node j_i to an "intermediate node" i) and then inputs $\boldsymbol{y}$ into the circuit $\mathcal{M}_3^r$.

Denote the (possibly random) output bit of $C(\boldsymbol{j})$ by Z. For any fixed input vector $\boldsymbol{x}$, assuming that our majority gates can fail with probability $\leq \epsilon \leq \alpha/2$ independently, denote $q(\boldsymbol{j}, \boldsymbol{x}) := \mathbf{P}[\, Z = 1 \,]$. Assume that the input is a vector $\boldsymbol{X} = (X_1, \ldots, X_k)$ of (not necessarily independent) Boolean random variables, with $p(\boldsymbol{x}) := \mathbf{P}[\, \boldsymbol{X} = \boldsymbol{x} \,]$. Denoting $|\boldsymbol{X}| = \sum_i X_i$, assume $\mathbf{P}[\, |\boldsymbol{X}| > \alpha k \,] \leq \rho < 1$. Prove

that there is a choice of the vector $\boldsymbol{j}$ for which

$$\sum_{\boldsymbol{x}} p(\boldsymbol{x})q(\boldsymbol{j},\boldsymbol{x}) \le \rho + \max\{10\epsilon,\ 0.3(\alpha/0.3)^{2^k}\}\ .$$

The choice may depend on the distribution of the random vector $\boldsymbol{X}$. *Hint.* Choose the vector $\boldsymbol{j}$ (and hence the circuit $C(\boldsymbol{j})$) randomly, as a random vector $\boldsymbol{J} = (J_1, \ldots, J_k)$ where the variables J_i are independent and uniformly distributed over $\{1, \ldots, k\}$, and denote $s(\boldsymbol{j}) := \mathbf{P}[\,\boldsymbol{J} = \boldsymbol{j}\,]$. Then prove

$$\sum_{j} s(\boldsymbol{j}) \sum_{\boldsymbol{x}} p(\boldsymbol{x})q(\boldsymbol{j},\boldsymbol{x}) \le \rho + \max\{10\epsilon,\ 0.3(\alpha/0.3)^{2^k}\}.$$

For this, interchange the averaging over $\boldsymbol{x}$ and $\boldsymbol{j}$. Then note that $\sum_j s(\boldsymbol{j})q(\boldsymbol{j},\boldsymbol{x})$ is the probability of $Z = 1$ when the "wires" J_i are chosen randomly "on the fly" during the computation of the circuit.]

4.4-5 Taking the notation of Exercise 4.4-3 suppose, like there, that the random variables F_v are independent of each other, and their distribution does not depend on the Boolean input vector. Take the Boolean circuit $\mathrm{Cab}(\mathcal{N},\mathcal{R})$ introduced in Definition 4.14, and define the random Boolean vector $\boldsymbol{T} = (T_1, \ldots, T_k)$ where $T_i = 1$ if and only if the ith output node is tainted. Apply Exercise 4.4-4 to show that there is a circuit $C(\boldsymbol{j})$ that can be attached to the output nodes to play the role of the "coda" circuit in the proof of Theorem 4.7. The size of $C(\boldsymbol{j})$ is only linear in k, not $(k+1)^7$ as for the coda circuit in the proof there. But, we assumed a little more about the fault distribution, and also the choice of the "wiring" $\boldsymbol{j}$ depends on the circuit $\mathrm{Cab}(\mathcal{N},\mathcal{R})$.

4.5. The reliable storage problem

There is hardly any simpler computation than not doing anything, just keeping the input unchanged. This task does not fit well, however, into the simple model of Boolean circuits as introduced above.

4.5.1. Clocked circuits

An obvious element of ordinary computations is missing from the above described Boolean circuit model: *repetition*. If we want to repeat some computation steps, then we need to introduce *timing* into the work of computing elements and to *store* the partial results between consecutive steps. Let us look at the drawings of the circuit designer again. We will see components like in Figure 4.9, with one ingoing edge and no operation associated with them; these will be called ***shift registers***. The shift registers are controlled by one central ***clock*** (invisible on the drawing). At each clock pulse, the assignment value on the incoming edge jumps onto the outgoing edges and "stays in" the register. Figure 4.10 shows how shift registers may be used inside a circuit.

Definition 4.15 *A **clocked circuit** over a complete basis Q is given by a tuple*

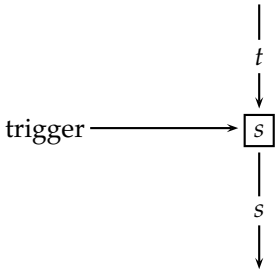


Figure 4.9. A shift register.

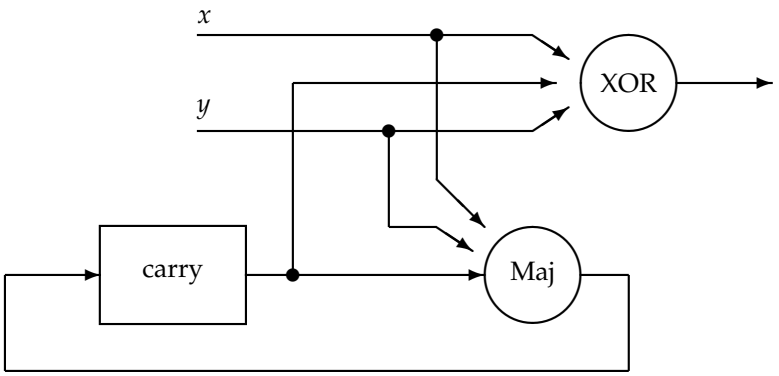


Figure 4.10. Part of a circuit which computes the sum of two binary numbers x, y. We feed the digits of x and y beginning with the lowest-order ones, at the input nodes. The digits of the sum come out on the output edge. A shift register holds the carry.

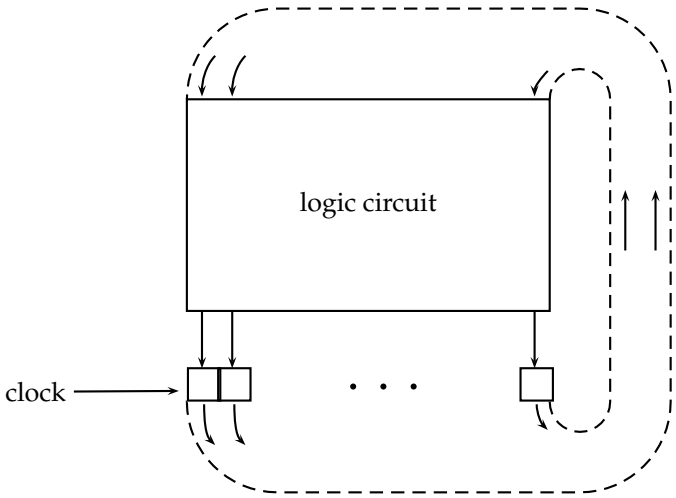


Figure 4.11. A "computer" consists of some memory (shift registers) and a Boolean circuit operating on it. We can define the ***size of computation*** as the size of the computer times the number of steps.

just like a Boolean circuit in (4.10)*. Also, the circuit defines a graph* $G = (V, E)$ *similarly. Recall that we identified nodes with the natural numbers* $1, \ldots, N$*. To each non-input node* v *either a gate* b_v *is assigned as before, or a* ***shift register****: in this case* $k_v = 1$ *(there is only one argument). We do not require the graph to be acyclic, but we do require every directed cycle (if there is any) to pass through at least one shift register.*

The circuit works in a sequence $t = 0, 1, 2, \ldots$ *of* ***clock cycles****. Let us denote the input vector at clock cycle* t *by* $\boldsymbol{x}^t = (x_1^t, \ldots, x_n^t)$*, the shift register states by* $\boldsymbol{s}^t = (s_1^t, \ldots, s_k^t)$*, and the output vector by* $\boldsymbol{y}^t = (y_1^t, \ldots, y_m^t)$*. The part of the circuit going from the inputs and the shift registers to the outputs and the shift registers defines two Boolean vector functions* $\lambda : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^m$ *and* $\tau : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^k$*. The operation of the clocked circuit is described by the following equations (see Figure 4.11, which does not show any inputs and outputs).*

$$\boldsymbol{y}^t = \lambda(\boldsymbol{s}^t, \boldsymbol{x}^t), \quad \boldsymbol{s}^{t+1} = \tau(\boldsymbol{s}^t, \boldsymbol{x}^t) \ . \tag{4.23}$$

Frequently, we have no inputs or outputs during the work of the circuit, so the equations (4.23) can be simplified to

$$\boldsymbol{s}^{t+1} = \tau(\boldsymbol{s}^t) \ . \tag{4.24}$$

How to use a clocked circuit described by this equation for computation? We write some initial values into the shift registers, and propagate the assignment using the gates, for the given clock cycle. Now we send a clock pulse to the register, causing

it to write new values to their output edges (which are identical to the input edges of the circuit). After this, the new assignment is computed, and so on.

How to compute a *function* $f(x)$ with the help of such a circuit? Here is a possible convention. We enter the input x (only in the first step), and then run the circuit, until it signals at an extra output edge when desired result $f(x)$ can be received from the other output nodes.

Example 4.8 This example uses a convention different from the above described one: new input bits are supplied in every step, and the output is also delivered continuously. For the binary adder of Figure 4.10, let u^t and v^t be the two input bits in cycle t, let c^t be the content of the carry, and w^t be the output in the same cycle. Then the equations (4.23) now have the form

$$w^t = u^t \oplus v^t \oplus c^t, \quad c^{t+1} = \mathrm{Maj}(u^t, v^t, c^t) \ ,$$

where Maj is the majority operation.

4.5.2. Storage

A clocked circuit is an interesting parallel computer but let us pose now a task for it that is trivial in the absence of failures: information storage. We would like to store a certain amount of information in such a way that it can be recovered after some time, despite failures in the circuit. For this, the transition function τ introduced in (4.24) cannot be just the identity: it will have to perform some *error-correcting* operations. The restoring organs discussed earlier are natural candidates. Indeed, suppose that we use k memory cells to store a bit of information. We can call the content of this k-tuple ***safe*** when the number of memory cells that dissent from the correct value is under some treshold ϑk. Let the rest of the circuit be a restoring organ built on a (d, α, γ, k)-compressor with $\alpha = 0.9\vartheta$. Suppose that the input cable is safe. Then the probability that after the transition, the new output cable (and therefore the new state) is not safe is $O(e^{-ck})$ for some constant c. Suppose we keep the circuit running for t steps. Then the probability that the state is not safe in some of these steps is $O(te^{-ck})$ which is small as long as t is significantly smaller than e^{ck}. When storing m bits of information, the probability that any of the bits loses its safety in some step is $O(mte^{-cm})$.

To make this discussion rigorous, an error model must be introduced for clocked circuits. Since we will only consider simple transition functions τ like the majority vote above, with a single computation step between times t and $t+1$, we will make the model also very simple.

Definition 4.16 *Consider a clocked circuit described by equation (4.24), where at each time instant $t = 0, 1, 2, \ldots$, the configuration is described by the bit vector $\boldsymbol{s}^t = (s_1^t, \ldots, s_n^t)$. Consider a sequence of random bit vectors $\boldsymbol{Y}^t = (Y_1^t, \ldots, Y_n^t)$ for $t = 0, 1, 2, \ldots$. Similarly to (4.13) we define*

$$Z_{i,t} = \tau(\boldsymbol{Y}^{t-1}) \oplus Y_i^t \ . \tag{4.25}$$

*Thus, $Z_{i,t} = 1$ says that a **failure** occurs at the space-time point (i, t). The sequence*

*$\{\boldsymbol{Y}^t\}$ will be called ϵ-**admissible** if (4.14) holds for every set C of space-time points with $t > 0$.*

By the just described construction, it is possible to keep m bits of information for T steps in

$$O(m \lg(mT)) \tag{4.26}$$

memory cells. More precisely, the cable $\boldsymbol{Y}^T$ will be safe with large probability in any admissible evolution $\boldsymbol{Y}^t$ $(t = 0, \ldots, T)$.

Cannot we do better? The reliable information storage problem is related to the problem of ***information transmission***: given a ***message*** $\boldsymbol{x}$, a ***sender*** wants to transmit it to a ***receiver*** throught a ***noisy channel***. Only now sender and receiver are the same person, and the noisy channel is just the passing of time. Below, we develop some basic concepts of reliable information transmission, and then we will apply them to the construction of a reliable data storage scheme that is more economical than the above seen naive, repetition-based solution.

4.5.3. Error-correcting codes

Error detection To protect information, we can use redundancy in a way more efficient than repetition. We might even add only a single redundant bit to our message. Let $\boldsymbol{x} = (x_1, \ldots, x_6)$, $(x_i \in \{0, 1\})$ be the word we want to protect. Let us create the ***error check bit***

$$x_7 = x_1 \oplus \cdots \oplus x_6 \ .$$

For example, $\boldsymbol{x} = 110010$, $\boldsymbol{x}' = 1100101$. Our ***codeword*** $\boldsymbol{x}' = (x_1, \ldots, x_7)$ will be subject to noise and it turns into a new word, $\boldsymbol{y}$. If $\boldsymbol{y}$ differs from $\boldsymbol{x}'$ in a single changed (not deleted or added) bit then we will ***detect*** this, since then $\boldsymbol{y}$ violates the ***error check relation***

$$y_1 \oplus \cdots \oplus y_7 = 0 \ .$$

We will not be able to correct the error, since we do not know which bit was corrupted.

Correcting a single error To also ***correct*** corrupted bits, we need to add more error check bits. We may try to add two more bits:

$$\begin{aligned} x_8 &= x_1 \oplus x_3 \oplus x_5 \ , \\ x_9 &= x_1 \oplus x_2 \oplus x_5 \oplus x_6 \ . \end{aligned}$$

Then an uncorrupted word $\boldsymbol{y}$ must satisfy the error check relations

$$\begin{aligned} y_1 \oplus \cdots \oplus y_7 &= 0 \ , \\ y_1 \oplus y_3 \oplus y_5 \oplus y_8 &= 0 \ , \\ y_1 \oplus y_2 \oplus y_5 \oplus y_6 \oplus y_9 &= 0 \ , \end{aligned}$$

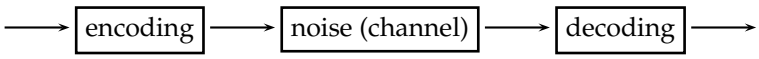


Figure 4.12. Transmission through a noisy channel.

or, in matrix notation $\boldsymbol{Hy} \bmod 2 = 0$, where

$$\boldsymbol{H} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} = (\boldsymbol{h}_1, \ldots, \boldsymbol{h}_9) \ .$$

Note $\boldsymbol{h}_1 = \boldsymbol{h}_5$. The matrix $\boldsymbol{H}$ is called the ***error check matrix***, or ***parity check matrix***.

Another way to write the error check relations is

$$y_1\boldsymbol{h}_1 \oplus \cdots \oplus y_5\boldsymbol{h}_5 \oplus \cdots \oplus y_9\boldsymbol{h}_9 = 0.$$

Now if $\boldsymbol{y}$ is corrupted, even if only in a single position, unfortunately we still cannot correct it: since $\boldsymbol{h}_1 = \boldsymbol{h}_5$, the error could be in position 1 or 5 and we could not tell the difference. If we choose our error-check matrix $\boldsymbol{H}$ in such a way that the colum vectors $\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots$ are *all different* (of course also from 0), then we can always correct an error, provided there is only one. Indeed, if the error was in position 3 then

$$\boldsymbol{Hy} \bmod 2 = \boldsymbol{h}_3.$$

Since all vectors $\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots$ are different, if we see the vector $\boldsymbol{h}_3$ we can imply that the bit y_3 is corrupted. This code is called the ***Hamming code***. For example, the following error check matrix defines the Hamming code of size 7:

$$\boldsymbol{H} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} = (\boldsymbol{h}_1, \ldots, \boldsymbol{h}_7). \tag{4.27}$$

In general, if we have s error check bits then our code can have size $2^s - 1$, hence the number of bits left to store information, the ***information bits*** is $k = 2^s - s - 1$. So, to protect m bits of information from a single error, the Hamming code adds $\approx \log m$ error check bits. This is much better than repeating every bit 3 times.

Codes Let us summarize the error-correction scenario in general terms. In order to fight noise, the sender ***encodes*** the ***message*** $\boldsymbol{x}$ by an ***encoding function*** ϕ_* into a longer string $\phi_*(\boldsymbol{x})$ which, for simplicity, we also assume to be binary. This ***codeword*** will be changed by noise into a string $\boldsymbol{y}$. The receiver gets $\boldsymbol{y}$ and applies to it a ***decoding function*** ϕ^*.

Definition 4.17 *The pair of functions* $\phi_* : \{0,1\}^m \to \{0,1\}^n$ *and* $\phi^* : \{0,1\}^n \to \{0,1\}^m$ *is called a* ***code*** *if* $\phi^*(\phi_*(\boldsymbol{x})) = \boldsymbol{x}$ *holds for all* $\boldsymbol{x} \in \{0,1\}^m$. *The strings*

*$\boldsymbol{x} \in \{0,1\}^m$ are called **messages**, words of the form $\boldsymbol{y} = \phi_*(\boldsymbol{x}) \in \{0,1\}^n$ are called **codewords.** (Sometimes the set of all codewords by itself is also called a code.) For every message $\boldsymbol{x}$, the set of words $C_{\boldsymbol{x}} = \{\, \boldsymbol{y} : \phi^*(\boldsymbol{y}) = \boldsymbol{x} \,\}$ is called the **decoding set** of $\boldsymbol{x}$. (Of course, different decoding sets are disjoint.) The number*

$$R = m/n$$

*is called the **rate** of the code.*

*We say that our code that **corrects** t **errors** if for all possible messages $\boldsymbol{x} \in \{0,1\}^m$, if the received word $\boldsymbol{y} \in \{0,1\}^n$ differs from the codeword $\phi_*(\boldsymbol{x})$ in at most t positions, then $\phi^*(\boldsymbol{y}) = \boldsymbol{x}$.*

If the rate is R then the n-bit codewords carry Rn bits of useful information. In terms of decoding sets, a code corrects t errors if each decoding set $C_{\boldsymbol{x}}$ contains all words that differ from $\phi_*(\boldsymbol{x})$ in at most t symbols (the set of these words is a kind of "ball" of radius t).

The Hamming code corrects a single error, and its rate is close to 1. One of the important questions connected with error-correcting codes is how much do we have to lower the rate in order to correct more errors.

Having a notion of codes, we can formulate the main result of this section about information storage.

Theorem 4.11 (Network information storage). *There are constants $\epsilon, c_1, c_2, R > 0$ with the following property. For all sufficiently large m, there is a code (ϕ_*, ϕ^*) with message length m and codeword length $n \le m/R$, and a Boolean clocked circuit $\mathcal{N}$ of size $O(n)$ with n inputs and n outputs, such that the following holds. Suppose that at time 0, the memory cells of the circuit contain string $\boldsymbol{Y}_0 = \phi_*(\boldsymbol{x})$. Suppose further that the evolution $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \ldots, \boldsymbol{Y}_t$ of the circuit has ϵ-admissible failures. Then we have*

$$\mathbf{P}[\,\phi^*(\boldsymbol{Y}_t) \neq \boldsymbol{x}\,] < t(c_1\epsilon)^{-c_2 n}.$$

This theorem shows that it is possible to store m bits information for time t, in a clocked circuit of size

$$O(\max(\log t, m)) \ .$$

As long as the storage time t is below the exponential bound e^{cm} for a certain constant c, this circuit size is only a constant times larger than the amount m of information it stores. (In contrast, in (4.26) we needed an extra factor $\log m$ when we used a separate restoring organ for each bit.)

The theorem says nothing about how difficult it is to compute the codeword $\phi_*(\boldsymbol{x})$ at the beginning and how difficult it is to carry out the decoding $\phi^*(\boldsymbol{Y}_t)$ at the end. Moreover, it is desireable to perform these two operations also in a noise-tolerant fashion. We will return to the problem of decoding later.

Linear algebra Since we will be dealing more with bit matrices, it is convenient to introduce the algebraic structure

$$\mathbb{F}_2 = (\{0,1\}, +, \cdot),$$

which is a two-element ***field.*** Addition and multiplication in $\mathbb{F}_2$ are defined modulo 2 (of course, for multiplication this is no change). It is also convenient to vest the set $\{0,1\}^n$ of binary strings with the structure $\mathbb{F}_2^n$ of an n-dimensional vector space over the field $\mathbb{F}_2$. Most theorems and algorithms of basic linear algebra apply to arbitrary fields: in particular, one can define the row rank of a matrix as the maximum number of linearly independent rows, and similarly the column rank. Then it is a theorem that the row rank is equal to the colum rank. From now on, in algebraic operations over bits or bit vectors, we will write $+$ in place of $\oplus$ unless this leads to confusion. To save space, we will frequently write column vectors horizontally: we write

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = (x_1, \ldots, x_n)^T,$$

where $\boldsymbol{A}^T$ denotes the transpose of matrix $\boldsymbol{A}$. We will write

$$\boldsymbol{I}_r$$

for the identity matrix over the vector space $\mathbb{F}_2^r$.

Linear codes Let us generalize the idea of the Hamming code.

Definition 4.18 *A code (ϕ_*, ϕ^*) with message length m and codeword length n is **linear** if, when viewing the message and code vectors as vectors over the field $\mathbb{F}_2$, the encoding function can be computed according to the formula*

$$\phi_*(\boldsymbol{x}) = \boldsymbol{G}\boldsymbol{x},$$

*with an $m \times n$ matrix $\boldsymbol{G}$ called the **generator matrix** of the code. The number m is called the the number of **information bits** in the code, the number*

$$k = n - m$$

*the number of **error-check bits**.*

Example 4.9 The matrix $\boldsymbol{H}$ in (4.27) can be written as $\boldsymbol{H} = (\boldsymbol{K}, \boldsymbol{I}_3)$, with

$$\boldsymbol{K} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}.$$

Then the error check relation can be written as

$$\boldsymbol{y} = \begin{matrix} \boldsymbol{I}_4 \\ -\boldsymbol{K} \end{matrix} \begin{pmatrix} y_1 \\ \vdots \\ y_4 \end{pmatrix}.$$

This shows that the bits $y_1, \ldots, y_4$ can be taken to be the message bits, or "information bits", of the code, making the Hamming code a linear code with the generator matrix

$(\boldsymbol{I}_4, -\boldsymbol{K})^T$. (Of course, $-\boldsymbol{K} = \boldsymbol{K}$ over the field $\mathbb{F}_2$.)

The following statement is proved using standard linear algebra, and it generalizes the relation between error check matrix and generator matrix seen in Example 4.9.

Proposition 4.12 *Let $k, m > 0$ be given with $n = m + k$.*

(a) *For every $n \times m$ matrix $\boldsymbol{G}$ of rank m over $\mathbb{F}_2$ there is a $k \times n$ matrix $\boldsymbol{H}$ of rank k with the property*

$$\{\, \boldsymbol{G}\boldsymbol{x} : \boldsymbol{x} \in \mathbb{F}_2^m \,\} = \{\, \boldsymbol{y} \in \mathbb{F}_2^n : \boldsymbol{H}\boldsymbol{y} = 0 \,\}. \tag{4.28}$$

(b) *For every $k \times n$ matrix $\boldsymbol{H}$ of rank k over $\mathbb{F}_2$ there is an $n \times m$ matrix $\boldsymbol{G}$ of rank m with property (4.28).*

Definition 4.19 *For a vector $\boldsymbol{x}$, let $|\boldsymbol{x}|$ denote the number of its nonzero elements: we will also call it the **weight** of $\boldsymbol{x}$.*

In what follows it will be convenient to define a code starting from an error-check matrix $\boldsymbol{H}$. If the matrix has rank k then the code has rate

$$R = 1 - k/n.$$

We can fix any subset S of k linearly independent columns, and call the indices $i \in S$ ***error check bits*** and the indices $i \notin S$ the ***information bits***. (In Example 4.9, we chose $S = \{5, 6, 7\}$.) Important operations can performed over a code, however, without fixing any separation into error-check bits and information bits.

4.5.4. Refreshers

Correcting a single error was not too difficult; finding a similar scheme to correct 2 errors is much harder. However, in storing n bits, typically ϵn (much more than 2) of those bits will be corrupted in every step. There are ingenious and quite efficient codes of positive rate (independent of n) correcting even this many errors. When applied to information storage, however, the error-correction mechanism itself must also work in noise, so we are looking for a particularly simple one. It works in our favor, however, that not all errors need to be corrected: it is sufficient to cut down their number, similarly to the restoring organ in reliable Boolean circuits above.

For simplicity, as gates of our circuit we will allow certain Boolean functions with a large, but constant, number of arguments. On the other hand, our Boolean circuit will have just depth 1, similarly to a restoring organ of Section 4.4. The output of each gate is the input of a memory cell (shift register). For simplicity, we identify the gate and the memory cell and call it a ***cell.*** At each clock tick, a cell reads its inputs from other cells, computes a Boolean function on them, and stores the result (till the next clock tick). But now, instead of majority vote among the input values cells, the Boolean function computed by each cell will be slightly more complicated.

Our particular restoring operations will be defined, with the help of a certain $k \times n$ parity check matrix $\boldsymbol{H} = (h_{ij})$. Let $\boldsymbol{x} = (x_1, \ldots, x_n)^T$ be a vector of bits. For some $j = 1, \ldots, n$, let V_j (from "vertical") be the set of those indices i with $h_{ij} = 1$. For integer $i = 1, \ldots, k$, let H_i (from "horizontal") be the set of those indices j with $h_{ij} = 1$. Then the condition $\boldsymbol{Hx} = 0$ can also be expressed by saying that for all i, we have $\sum_{j \in H_i} x_j \equiv 0 \pmod 2$. The sets H_i are called the ***parity check sets*** belonging to the matrix $\boldsymbol{H}$. From now on, the indices i will be called ***checks***, and the indices j ***locations.***

Definition 4.20 *A linear code $\boldsymbol{H}$ is a **low-density parity-check code** with bounds $K, N > 0$ if the following conditions are satisfied:*

(a) *For each j we have $|V_j| \leq K$;*

(b) *For each i we have $|H_i| \leq N$.*

In other words, the weight of each row is at most N and the weight of each column is at most K.

In our constructions, we will keep the bounds K, N constant while the length n of codewords grows. Consider a situation when $\boldsymbol{x}$ is a codeword corrupted by some errors. To check whether bit x_j is incorrect we may check all the sums

$$s_i = \sum_{j \in H_i} x_j$$

for all $i \in V_j$. If all these sums are 0 then we would not suspect x_j to be in error. If only one of these is nonzero, we will know that $\boldsymbol{x}$ has some errors but we may still think that the error is not in bit x_j. But if a significant number of these sums is nonzero then we may suspect that x_j is a culprit and may want to change it. This idea suggests the following definition.

Definition 4.21 *For a low-density parity-check code $\boldsymbol{H}$ with bounds K, N, the **refreshing operation** associated with the code is the following, to be performed simultaneously for all locations j:*

> *Find out whether more than $\lfloor K/2 \rfloor$ of the sums s_i are nonzero among the ones for $i \in V_j$. If this is the case, flip x_j.*

*Let $\boldsymbol{x}^{\boldsymbol{H}}$ denote the vector obtained from $\boldsymbol{x}$ by this operation. For parameters $0 < \vartheta, \gamma < 1$, let us call $\boldsymbol{H}$ a $(\vartheta, \gamma, K, N, k, n)$-**refresher** if for each vector $\boldsymbol{x}$ of length n with weight $|\boldsymbol{x}| \leq \vartheta n$ the weight of the resulting vector decreases thus: $|\boldsymbol{x}^{\boldsymbol{H}}| \leq \gamma \vartheta n$.*

Notice the similarity of refreshers to compressors. The following lemma shows the use of refreshers, and is an example of the advantages of linear codes.

Lemma 4.13 *For an $(\vartheta, \gamma, K, N, k, n)$-refresher $\boldsymbol{H}$, let $\boldsymbol{x}$ be an n-vector and $\boldsymbol{y}$ a codeword of length n with $|\boldsymbol{x} - \boldsymbol{y}| \leq \vartheta n$. Then $|\boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{y}| \leq \gamma \vartheta n$.*

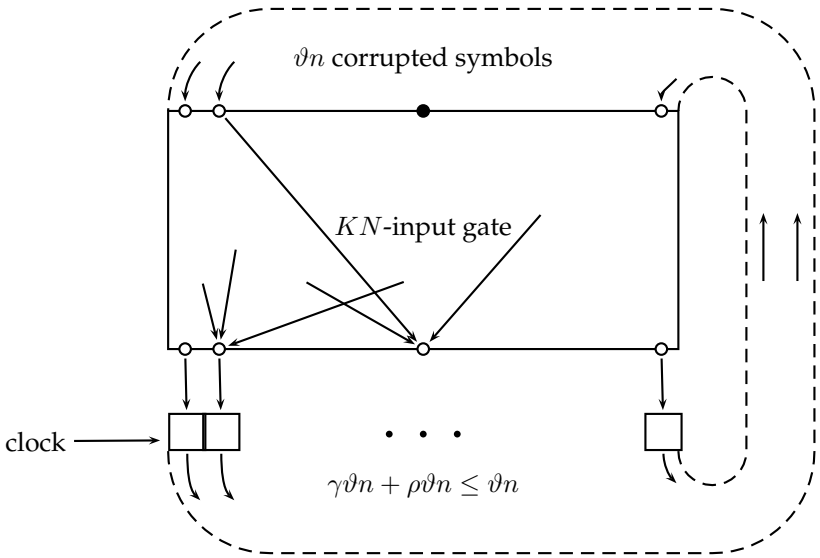


Figure 4.13. Using a refresher

Proof. Since $\boldsymbol{y}$ is a codeword, $\boldsymbol{Hy} = 0$, implying $\boldsymbol{H}(\boldsymbol{x}-\boldsymbol{y}) = \boldsymbol{Hx}$. Therefore the error correction flips the same bits in $\boldsymbol{x}-\boldsymbol{y}$ as in $\boldsymbol{x}$: $(\boldsymbol{x}-\boldsymbol{y})^{\boldsymbol{H}} - (\boldsymbol{x}-\boldsymbol{y}) = \boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{x}$, giving $\boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{y} = (\boldsymbol{x}-\boldsymbol{y})^{\boldsymbol{H}}$. So, if $|\boldsymbol{x}-\boldsymbol{y}| \le \vartheta n$, then $|\boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{y}| = |(\boldsymbol{x}-\boldsymbol{y})^{\boldsymbol{H}}| \le \gamma\vartheta n$. ■

Theorem 4.14 *There is a parameter $\vartheta > 0$ and integers $K > N > 0$ such that for all sufficiently large codelength n and $k = Nn/K$ there is a $(\vartheta, 1/2, K, N, k, n)$-refresher with at least $n - k = 1 - N/K$ information bits.*

In particular, we can choose $N = 100$, $K = 120$, $\vartheta = 1.31 \cdot 10^{-4}$.

We postpone the proof of this theorem, and apply it first.

Proof of Theorem 4.11. Theorem 4.14 provides us with a device for information storage. Indeed, we can implement the operation $\boldsymbol{x} \to \boldsymbol{x}^{\boldsymbol{H}}$ using a single gate g_j of at most KN inputs for each bit j of $\boldsymbol{x}$. Now as long as the inequality $|\boldsymbol{x}-\boldsymbol{y}| \le \vartheta n$ holds for some codeword $\boldsymbol{y}$, the inequality $|\boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{y}| \le \gamma\vartheta n$ follows with $\gamma = 1/2$. Of course, some gates will fail and introduce new deviations resulting in some $\boldsymbol{x}'$ rather than $\boldsymbol{x}^{\boldsymbol{H}}$. Let $e\epsilon < \vartheta/2$ and $\rho = 1 - \gamma (= 1/2)$. Then just as earlier, the probability that there are more than $\rho\vartheta n$ failures is bounded by the exponentially decreasing expression $(e\epsilon/\rho\vartheta)^{\rho\vartheta n}$. With fewer than $\rho\vartheta n$ new deviations, we will still have $|\boldsymbol{x}' - \boldsymbol{y}| < (\gamma+\rho)\vartheta n < \vartheta n$. The probability that at any time $\le t$ the number of failures is more than $\rho\vartheta n$ is bounded by

$$t(e\epsilon/\rho\vartheta)^{\rho\vartheta n} < t(6\epsilon/\vartheta)^{(1/2)\vartheta n}.$$

■

Example 4.10 Let $\epsilon = 10^{-9}$. Using the sample values in Theorem 4.14 we can take $N = 100$, $K = 120$, so the information rate is $1 - N/K = 1/6$. With the corresponding values of ϑ, and $\gamma = \rho = 1/2$, we have $\rho\vartheta = 6.57 \cdot 10^{-5}$. The probability that there are more than $\rho\vartheta n$ failures is bounded by

$$(e\epsilon/\rho\vartheta)^{\rho\vartheta n} = (10^{-4}e/6.57)^{6.57\cdot 10^{-5}n} \approx e^{-6.63\cdot 10^{-4}n}.$$

This is exponentially decreasing with n, albeit initially very slowly: it is not really small until $n = 10^4$. Still, for $n = 10^6$, it gives $e^{-663} \approx 1.16 \cdot 10^{-288}$.

Decoding? In order to use a refresher for information storage, first we need to enter the encoded information into it, and at the end, we need to decode the information from it. How can this be done in a noisy environment? We have nothing particularly smart to say here about encoding besides the reference to the general reliable computation scheme discussed earlier. On the other hand, it turns out that if ϵ is sufficiently small then *decoding can be avoided* altogether.

Recall that in our codes, it is possible to designate certain symbols as information symbols. So, in principle it is sufficient to read out these symbols. The question is only how likely it is that any one of these symbols will be corrupted. The following theorem upperbounds the probability for any symbol to be corrupted, at any time.

Theorem 4.15 *For parameters $\vartheta, \gamma > 0$, integers $K > N > 0$, codelength n, with $k = Nn/K$, consider a $(\vartheta, 1/2, K, N, k, n)$-refresher. Build a Boolean clocked circuit $\mathcal{N}$ of size $O(n)$ with n inputs and n outputs based on this refresher, just as in the proof of Theorem 4.11. Suppose that at time 0, the memory cells of the circuit contain string $\boldsymbol{Y}_0 = \phi_*(\boldsymbol{x})$. Suppose further that the evolution $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \ldots, \boldsymbol{Y}_t$ of the circuit has ϵ-admissible failures. Let $\boldsymbol{Y}_t = (Y_t(1), \ldots, Y_t(n))$ be the bits stored at time t. Then $\epsilon < (2.1KN)^{-10}$ implies*

$$\mathbf{P}[\, Y_t(j) \neq Y_0(j) \,] \leq c\epsilon + t(6\epsilon/\vartheta)^{(1/2)\vartheta n}$$

for some c depending on N, K.

Remark 4.16 *What we are bounding is only the probability of a corrupt symbol in the particular position j. Some of the symbols will certainly be corrupt, but any one symbol one points to will be corrupt only with probability $\leq c\epsilon$.*

The upper bound on ϵ required in the condition of the theorem is very severe, underscoring the theoretical character of this result.

Proof. As usual, it is sufficient to assume $\boldsymbol{Y}_0 = 0$. Let $D_t = \{\, j : Y_t(j) = 1 \,\}$, and let E_t be the set of circuit elements j which fail at time t. Let us define the following sequence of integers:

$$b_0 = 1, \quad b_{u+1} = \lceil (4/3)b_u \rceil, \quad c_u = \lceil (1/3)b_u \rceil.$$

It is easy to see by induction

$$b_0 + \cdots + b_{u-1} \leq 3b_u \leq 9c_u. \tag{4.29}$$

The first members of the sequence b_u are 1,2,3,4,6,8,11,15,18,24,32, and for c_u they are 1,1,1,2,2,3,4,5,6,8,11.

Lemma 4.17 *Suppose that $Y_t(j_0) \neq 0$. Then either there is a time $t' < t$ at which $\geq (1/2)\vartheta n$ circuit elements failed, or there is a sequence of sets $B_u \subseteq D_{t-u}$ for $0 \leq u < v$ and $C \subseteq E_{t-v}$ with the following properties.*

(a) For $u > 0$, every element of B_u shares some error-check with some element of B_{u-1}. Also every element of C shares some error-check with some element of B_{v-1}.

(b) We have $|E_{t-u} \cap B_u| < |B_u|/3$ for $u < v$, on the other hand $C \subseteq E_{t-v}$.

(c) We have $B_0 = \{j_0\}$, $|B_u| = b_u$, for all $u < v$, and $|C| = c_v$.

Proof. We will define the sequence B_u recursively, and will say when to stop. If $j_0 \in E_t$ then we set $v = 0$, $C = \{0\}$, and stop. Suppose that B_u is already defined. Let us define B_{u+1} (or C if $v = u + 1$). Let B'_{u+1} be the set of those j which share some error-check with an element of B_u, and let $B''_{u+1} = B'_{u+1} \cap D_{t-u-1}$. The refresher property implies that either $|B''_{u+1}| > \vartheta n$ or

$$|B_u \smallsetminus E_{t-u}| \leq (1/2)|B''_{u+1}| \; .$$

In the former case, there must have been some time $t' < t - u$ with $|E_{t'}| > (1/2)\vartheta n$, otherwise D_{t-u-1} could never become larger than ϑn. In the latter case, the property $|E_{t-u} \cap B_u| < (1/3)|B_u|$ implies

$$\begin{aligned} (2/3)|B_u| < |B_u \smallsetminus E_{t-u}| \leq (1/2)|B''_{u+1}| \; , \\ (4/3)b_u < |B''_{u+1}|. \end{aligned}$$

Now if $|E_{t-u-1} \cap B''_{u+1}| < (1/3)|B''_{u+1}|$ then let B_{u+1} be any subset of B''_{u+1} with size b_{u+1} (there is one), else let $v = u + 1$ and $C \subseteq E_{t-u-1} \cap B''_{u+1}$ a set of size c_v (there is one). This construction has the required properties. ■

For a given B_u, the number of different choices for B_{u+1} is bounded by

$$\binom{|B'_{u+1}|}{b_{u+1}} \leq \binom{KNb_u}{b_{u+1}} \leq \left(\frac{eKNb_u}{b_{u+1}}\right)^{b_{u+1}} \leq ((3/4)eKN)^{b_{u+1}} \leq (2.1KN)^{b_{u+1}},$$

where we used (4.9). Similarly, the number of different choices for C is bounded by

$$\binom{KNb_{v-1}}{c_v} \leq \mu^{c_v} \text{ with } \mu = 2.1KN.$$

It follows that the number of choices for the whole sequence $B_1, \ldots, B_{v-1}, C$ is bounded by

$$\mu^{b_1 + \cdots + b_{v-1} + c_v}.$$

On the other hand, the probability for a fixed C to have $C \subseteq E_v$ is $\leq \epsilon^{c_v}$. This way, we can bound the probability that the sequence ends exactly at v by

$$p_v \leq \epsilon^{c_v} \mu^{b_1 + \cdots + b_{v-1} + c_v} \leq \epsilon^{c_v} \mu^{10 c_v},$$

where we used (4.29). For small v this gives

$$p_0 \le \epsilon, \quad p_1 \le \epsilon\mu, \quad p_2 \le \epsilon\mu^3, \quad p_3 \le \epsilon^2\mu^6, \quad p_4 \le \epsilon^2\mu^{10}, \quad p_5 \le \epsilon^3\mu^{16}.$$

Therefore

$$\sum_{v=0}^{\infty} p_v \le \sum_{v=0}^{5} p_v + \sum_{v=6}^{\infty} (\mu^{10}\epsilon)^{c_v} \le \epsilon(1+\mu+\mu^3) + \epsilon^2(\mu^6+\mu^{10}) + \frac{\epsilon^3\mu^{16}}{1-\epsilon\mu^{10}},$$

where we used $\epsilon\mu^{10} < 1$ and the property $c_{v+1} > c_v$ for $v \ge 5$. We can bound the last expression by $c\epsilon$ with an appropriate constantb c.

We found that the event $Y_t(j) \ne Y_0(j)$ happens either if there is a time $t' < t$ at which $\ge (1/2)\vartheta n$ circuit elements failed (this has probability bound $t(2e\epsilon/\vartheta)^{(1/2)\vartheta n}$) or an event of probability $\le c\epsilon$ occurs. ■

Expanders We will construct our refreshers from bipartite multigraphs with a property similar to compressors: expanders.

Definition 4.22 *Here, we will distinguish the two parts of the bipartite (multi) graphs not as inputs and outputs but as* ***left nodes*** *and* ***right nodes.*** *A bipartite multigraph B is (N,K)-**regular** if the points of the left set have degree N and the points in the right set have degree K. Consider such a graph, with the left set having n nodes (then the right set has nN/K nodes). For a subset E of the left set of B, let $\mathrm{Nb}(E)$ consist of the points connected by some edge to some element of E. We say that the graph B **expands** E by a factor λ if we have $|\mathrm{Nb}(E)| \ge \lambda|E|$. For $\alpha, \lambda > 0$, our graph B is an (N,K,α,λ,n)-expander if B expands every subset E of size $\le \alpha n$ of the left set by a factor λ.*

Definition 4.23 *Given an (N,K)-regular bipartite multigraph B, with left set $\{u_1,\dots,u_n\}$ and right set $\{v_1,\dots,v_k\}$, we assign to it a parity-check code $\boldsymbol{H}(B)$ as follows: $h_{ij} = 1$ if v_i is connected to u_j, and 0 otherwise.*

Now for every possible error set E, the set $\mathrm{Nb}(E)$ describes the set of parity checks that the elements of E participate in. Under some conditions, the lower bound on the size of $\mathrm{Nb}(E)$ guarantees that a sufficient number of errors will be corrected.

Theorem 4.18 *Let B be an $(N,K,\alpha,(7/8)N,n)$-expander with integer αn. Let $k = Nn/K$. Then $\boldsymbol{H}(B)$ is a $((3/4)\alpha, 1/2, K, N, k, n)$-refresher.*

Proof. More generally, for any $\epsilon > 0$, let B be an $(N,K,\alpha,(3/4+\epsilon)N,n)$-expander with integer αn. We will prove that $\boldsymbol{H}(B)$ is a $(\alpha(1+4\epsilon)/2, (1-4\epsilon), K, N, k, n)$-refresher. For an n-dimensional bit vector $\boldsymbol{x}$ with $A = \{\, j : x_j = 1 \,\}$, $a = |A| = |\boldsymbol{x}|$, assume

$$a \le n\alpha(1+4\epsilon)/2\ . \tag{4.30}$$

Our goal is to show $|\boldsymbol{x}^{\boldsymbol{H}}| \le a(1-4\epsilon)$: in other words, that in the corrected vector the number of errors decreases at least by a factor of $(1-4\epsilon)$.

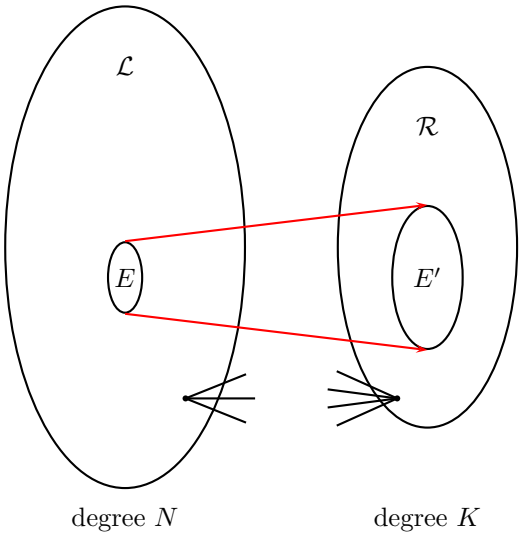


Figure 4.14. A regular expander.

Let F be the set of bits in A that the error correction operation fails to flip, with $f = |F|$, and G the set of of bits that were 0 but the operation turns them to 1, with $g = |G|$. Our goal is to bound $|F \cup G| = f + g$. The key observation is that each element of G shares at least half of its neighbors with elements of A, and similarly, each element of F shares at least half of its neighbors with other elements of A. Therefore both F and G contribute relatively weakly to the expansion of $A \cup G$. Since this expansion is assumed strong, the size of $|F \cup G|$ must be limited.

Let

$$\delta = |\mathrm{Nb}(A)|/(Na) \ .$$

By expansion, $\delta \geq 3/4 + \epsilon$.

First we show $|A \cup G| \leq \alpha n$. Assume namely that, on the contrary, $|A \cup G| > \alpha n$, and let G' be a subset of G such that $|A \cup G'| = \alpha n =: p$ (an integer, according to the assumptions of the theorem). By expansion,

$$(3/4 + \epsilon)Np \leq \mathrm{Nb}(A \cup G').$$

Each bit in G has at most $N/2$ neighbors that are not neighbors of A; so,

$$|\mathrm{Nb}(A \cup G')| \leq \delta Na + N(p - a)/2.$$

Combining these:

$$\begin{aligned} \delta a + (p - a)/2 &\geq (3/4 + \epsilon)p, \\ a &\geq p(1 + 4\epsilon)/(4\delta - 2) \geq \alpha n(1 + 4\epsilon)/2, \end{aligned}$$

since $\delta \leq 1$. This contradiction with (4.30) shows $|A \cup G| \leq \alpha n$.

Now $|A \cup G| \leq \alpha n$ implies (recalling that each element of G contributes at most $N/2$ new neighbors):

$$\begin{aligned}(3/4+\epsilon)N(a+g) &\leq |\mathrm{Nb}(A \cup G)| \leq \delta N a + (N/2)g,\\ (3/4+\epsilon)(a+g) &\leq \delta a + g/2,\\ (3/4+\epsilon)a + (1/4+\epsilon)g &\leq \delta a. \end{aligned} \tag{4.31}$$

Each $j \in F$ must share at least half of its neighbors with others in A. Therefore j contributes at most $N/2$ neighbors on its own; the contribution of the other $N/2$ must be divided by 2, so the the total contribution of j to the neighbors of A is at most $(3/4)N$:

$$\begin{aligned}\delta N a = \mathrm{Nb}(A) &\leq N(a-f) + (3/4)Nf = N(a - f/4),\\ \delta a &\leq a - f/4.\end{aligned}$$

Combining with (4.31):

$$\begin{aligned}(3/4+\epsilon)a + (1/4+\epsilon)g &\leq a - f/4,\\ (1-4\epsilon)a &\geq f + (1+4\epsilon)g \geq f + g.\end{aligned}$$

■

Random expanders Are there expanders good enough for Theorem 4.18? The maximum expansion factor is the degree N and we require a factor of $(7/8)N$. It turns out that random choice works here, too, similarly to the one used in the "construction" of compressors.

The choice has to be done in a way that the result is an (N, K)-regular bipartite multigraph of left size n. We will start with Nn left nodes $u_1, \ldots, u_{Nn}$ and Nn right nodes $v_1, \ldots, v_{Nn}$. Now we choose a random ***matching***, that is a set of Nn edges with the property that every left node is connected by an edge to exactly one right node. Let us call the resulting graph M. We obtain B now as follows: we collapse each group of N left nodes into a single node: $u_1, \ldots, u_N$ into one node, $u_{N+1}, \ldots, u_{2N}$ into another node, and so on. Similarly, we collapse each group of K right nodes into a single node: $v_1, \ldots, v_K$ into one node, $v_{K+1}, \ldots, v_{2K}$ into another node, and so on. The edges between any pair of nodes in B are inherited from the ancestors of these nodes in M. This results in a graph B with n left nodes of degree N and nN/K right nodes of degree K. The process may give multiple edges between nodes of B, this is why B is a multigraph. Two nodes of M will be called ***cluster neighbors*** if they are collapsed to the same node of B.

Theorem 4.19 *Suppose*

$$0 < \alpha \leq e^{\frac{-1}{N/8-1}} \cdot (22K)^{\frac{-1}{1-8/N}}.$$

Then the above random choice gives an $(N, K, \alpha, (7/8)N, n)$-expander with positive probability.

Example 4.11 If $N = 48$, $K = 60$ then the inequality in the condition of the theorem becomes

$$\alpha \leq 1/6785.$$

Proof. Let E be a set of size αn in the left set of B. We will estimate the probability that E has too few neighbors. In the above choice of the graph B we might as well start with assigning edges to the nodes of E, in some fixed order of the $N|E|$ nodes of the preimage of E in M. There are $N|E|$ edges to assign. Let us call a node of the right set of M ***occupied*** if it has a cluster neighbor already reached by an earlier edge. Let X_i be a random variable that is 1 if the ith edge goes to an occupied node and 0 otherwise. There are

$$Nn - i + 1 \geq Nn - N\alpha n = Nn(1-\alpha)$$

choices for the ith edge and at most $KN|E|$ of these are occupied. Therefore

$$\mathbf{P}[\, X_i = 1 \mid X_1, \ldots, X_{i-1} \,] \leq \frac{KN|E|}{Nn(1-\alpha)} = \frac{K\alpha}{1-\alpha} =: p.$$

Using the large deviations theorem in the generalization given in Exercise 4.1-3, we have, for $f > 0$:

$$\mathbf{P}[\, \sum_{i=1}^{N\alpha n} X_i \geq fN\alpha n \,] \leq e^{-N\alpha n D(f,p)} \leq \left(\frac{ep}{f}\right)^{fN\alpha n}.$$

Now, the number of different neighbors of E is $N\alpha n - \sum_i X_i$, hence

$$\mathbf{P}[\, N(E) \leq N\alpha n(1-f) \,] \leq \left(\frac{ep}{f}\right)^{fN\alpha n} = \left(\frac{eK\alpha}{f(1-\alpha)}\right)^{fN\alpha n}.$$

Let us now multiply this with the number

$$\sum_{i \leq \alpha n} \binom{n}{\alpha n} \leq (e/\alpha)^{\alpha n}$$

of sets E of size $\leq \alpha n$:

$$\left(\frac{e}{\alpha}\right)^{\alpha n} \left(\frac{eK\alpha}{f(1-\alpha)}\right)^{fN\alpha n} = \left(\alpha^{fN-1} e \left(\frac{eK}{f(1-\alpha)}\right)^{fN}\right)^{\alpha n} \leq \left(\alpha^{fN-1} e \left(\frac{eK}{0.99f}\right)^{fN}\right)^{\alpha n},$$

where in the last step we assumed $\alpha \leq 0.01$. This is < 1 if

$$\alpha \leq e^{\frac{-1}{fN-1}} \left(\frac{eK}{0.99f}\right)^{\frac{-1}{1-1/(fN)}}.$$

Substituting $f = 1/8$ gives the formula of the theorem. ■

Proof of Theorem 4.14. Theorem 4.18 shows how to get a refresher from an expander, and Theorem 4.19 shows the existence of expanders for certain parameters. Example 4.11 shows that the parameters can be chosen as needed for the refreshers. ■

Exercises

4.5-1 Prove Proposition 4.12.

4.5-2 Apply the ideas of the proof of Theorem 4.15 to the proof of Theorem 4.7, showing that the "coda" circuit is not needed: each wire of the output cable carries the correct value with high probability.

Problems

4-1 Critical value

Consider a circuit $\mathcal{M}_k$ like in Exercise 4.2-5, assuming that each gate fails with probability $\leq \epsilon$ independently of all the others and of the input. Assume that the input vector is all 0, and let $p_k(\epsilon)$ be the probability that the circuit outputs a 1. Show that there is a value $\epsilon_0 < 1/2$ with the property that for all $\epsilon < \epsilon_0$ we have $\lim_{k\to\infty} p_k(\epsilon) = 0$, and for $\epsilon_0 < \epsilon \leq 1/2$, we have have $\lim_{k\to\infty} p_k(\epsilon) = 1/2$. Estimate also the speed of convergence in both cases.

4-2 Regular compressor

We defined a compressor as a d-halfregular bipartite multigraph. Let us call a compressor ***regular*** if it is a d-regular multigraph (the input nodes also have degree d). Prove a theorem similar to Theorem 4.8: for each $\gamma < 1$ there is an integer $d > 1$ and an $\alpha > 0$ such that for all integer $k > 0$ there is a regular $(\dot{,}\alpha, \gamma, k)$-compressor. *Hint.* Choose a random d-regular bipartite multigraph by the following process: (1. Replace each vertex by a group of d vertices. 2. Choose a random complete matching betwen the new input and output vertices. 3. Merge each group of d vertices into one vertex again.) Prove that the probability, over this choice, that a d-regular multigraph is a not a compressor is small. For this, express the probability with the help of factorials and estimate the factorials using Stirling's formula.

4-3 Two-way expander

Recall the definition of expanders. Call a $(d, \alpha, \lg, k)$-expander ***regular*** if it is a d-regular multigraph (the input nodes also have degree d). We will call this multigraph a ***two-way expander*** if it is an expander in both directions: from A to B and from B to A. Prove a theorem similar to the one in Problem *4-2*: for all $\lg < d$ there is an $\alpha > 0$ such that for all integers $k > 0$ there is a two-way regular $(d, \alpha, \lg, k)$-expander.

4-4 Restoring organ from 3-way voting

The proof of Theorem 4.8 did not guarantee a $(\dot{,}\alpha, \gamma, k)$-compressor with any $\gamma < 1/2$, $\dot{<}7$. If we only want to use 3-way majority gates, consider the following construction. First create a 3-halfregular bipartite graph G with inputs $u_1, \ldots, u_k$ and outputs $v_1, \ldots, v_{3k}$, with a 3-input majority gate in each v_i. Then create new nodes $w_1, \ldots, w_k$, with a 3-input majority gate in each w_j. The gate of w_1 computes

the majority of v_1, v_2, v_3, the gate of w_2 computes the majority of v_4, v_5, v_6, and so on. Calculate whether a random choice of the graph G will turn the circuit with inputs $(u_1, \ldots, u_k)$ and outputs $(w_1, \ldots, w_k)$ into a restoring organ. Then consider three stages instead of two, where G has $9k$ outputs and see what is gained.

4-5 Restoring organ from NOR gates

The majority gate is not the only gate capable of strengthening the majority. Recall the NOR gate introduced in Exercise 4.2-2, and form $\mathrm{NOR}_2(x_1, x_2, x_3, x_4) = (x_1 \mathrm{NOR} x_2)\mathrm{NOR}(x_3 \mathrm{NOR} x_4)$. Show that a construction similar to Problem *4-4* can be carried out with NOR_2 used in place of 3-way majority gates.

4-6 More randomness, smaller restoring organs

Taking the notation of Exercise 4.4-3, suppose like there, that the random variables F_v are independent of each other, and their distribution does not depend on the Boolean input vector. Apply the idea of Exercise 4.4-5 to the construction of each restoring organ. Namely, construct a different restoring organ for each position: the choice depends on the circuit preceding this position. Show that in this case, our error estimates can be significantly improved. The improvement comes, just as in Exercise 4.4-5, since now we do not have to multiply the error probability by the number of all possible sets of size $\leq \alpha k$ of tainted wires. Since we know the distribution of this set, we can average over it.

4-7 Near-sorting with expanders

In this problem, we show that expanders can be used for "near-sorting". Let G be a regular two-way $(d, \alpha, \lg, k)$-expander, whose two parts of size k are A and B. According to a theorem of Kőnig, (the edge-set of) every d-regular bipartite multigraph is the disjoint union of (the edge-sets of) d complete matchings $M_1, \ldots, M_d$. To such an expander, we assign a Boolean circuit of depth d as follows. The circuit's nodes are subdivide into levels $i = 0, 1, \ldots, d$. On level i we have two disjoint sets A_i, B_i of size k of nodes a_{ij}, b_{ij} $(j = 1, \ldots, k)$. The Boolean value on a_{ij}, b_{ij} will be x_{ij} and y_{ij} respectively. Denote the vector of $2k$ values at stage i by $\boldsymbol{z}_i = (x_{i1}, \ldots, y_{ik})$. If (p, q) is an edge in the matching M_i, then we put an $\wedge$ gate into a_{ip}, and a $\vee$ gate into b_{iq}:

$$x_{ip} = x_{(i-1)p} \wedge y_{(i-1)q}, \quad y_{iq} = x_{(i-1)p} \vee y_{(i-1)q} \ .$$

This network is trying to "sort" the 0's to A_i and the 1's to B_i in d stages. More generally, the values in the vectors $\boldsymbol{z}_i$ could be arbitrary numbers. Then if $x \wedge y$ still means $\min(x, y)$ and $x \vee y$ means $\max(x, y)$ then each vector $\boldsymbol{z}_i$ is a permutation of the vector $\boldsymbol{z}_0$. Let $\mathbf{G} = (1 + \lambda)\alpha$. Prove that $\boldsymbol{z}_d$ is **G-*sorted*** in the sense that for all m, at least $\mathbf{G}m$ among the m smallest values of $\boldsymbol{z}_d$ is in its left half and at least $\mathbf{G}m$ among the m largest values are in its right half.

4-8 Restoring organ from near-sorters

Develop a new restoring organ using expanders, as follows. First, split each wire of the input cable A, to get two sets A'_0, B'_0. Attach the **G**-sorter of Problem *4-7*, getting outputs A'_d, B'_d. Now split the wires of B'_d into two sets A''_0, B''_0. Attach the **G**-sorter again, getting outputs A''_d, B''_d. Keep only $B = A''_d$ for the output cable. Show that the Boolean vector circuit leading from A to B can be used as a restoring organ.

Chapter notes

The large deviation theorem (Theorem 4.1), or theorems similar to it, are sometimes attributed to Chernoff or Bernstein. One of its frequently used variants is given in Exercise 4.1-2.

The problem of reliable computation with unreliable components was addressed by John von Neumann in [179] on the model of logic circuits. A complete proof of the result of that paper (with a different restoring organ) appeare first in the paper [60] of R. L. Dobrushin and S. I. Ortyukov. Our presentation relied on parts of the paper [191] of N. Pippenger.

The lower-bound result of Dobrushin and Ortyukov in the paper [59] (corrected in [189], [197] and [80]), shows that reduncancy of $\log n$ is unavoidable for a general reliable computation whose complexity is n. However, this lower bound only shows the necessity of putting the input into a redundantly encoded form (otherwise critical information may be lost in the first step). As shown in [191], for many important function classes, linear redundancy is achievable.

It seems natural to separate the cost of the initial encoding: it might be possible to perform the rest of the computation with much less redundancy. An important step in this direction has been made by D. Spielman in the paper [232] in (essentially) the clocked-circuit model. Spielman takes a parallel computation with time t running on w elementary components and makes it reliable using only $(\log w)^c$ times more processors and running it $(\log w)^c$ times longer. The failure probability will be $t\exp(-w^{1/4})$. This is small as long as t is not much larger than $\exp(w^{1/4})$. So, the redundancy is bounded by some power of the logarithm of the *space requirement*; the time requirement does not enter explictly. In Boolean circuits no time- and space-complexity is defined separately. The size of the circuit is analogous to the quantity obtained in other models by taking the product of space and time complexity.

Questions more complex than Problem *4-1* have been studied in [190]. The method of Problem *4-2*, for generating random d-regular multigraphs is analyzed for example in [25]. It is much harder to generate simple regular graphs (not multigraphs) uniformly. See for example [133].

The result of Exercise 4.2-4 is due to C. Shannon, see [221]. The asymptotically best circuit size for the worst functions was found by Lupanov in [156]. Exercise 4.3-1 is based on [60], and Exercise 4.3-2 is based on [59] (and its corrections).

Problem *4-7* is based on the starting idea of the $\lg n$ depth sorting networks in [8].

For storage in Boolean circuits we partly relied on A. V. Kuznietsov's paper [144] (the main theorem, on the existence of refreshers is from M. Pinsker). Low density parity check codes were introduced by R. G. Gallager in the book [74], and their use in reliable storage was first suggested by M. G. Taylor in the paper [242]. New, constructive versions of these codes were developed by M. Sipser and D. Spielman in the paper [233], with superfast coding and decoding.

Expanders, invented by Pinsker in [188] have been used extensively in theoretical computer science: see for example [174] for some more detail. This book also gives references on the construction of graphs with large eigenvalue-gap. Exercise 4.4-4 and Problem *4-6* are based on [60].

The use of expanders in the role of refreshers was suggested by Pippenger (private communication): our exposition follows Sipser and Spielman in [?]. Random expanders were found for example by Pinsker. The needed expansion rate ($> 3/4$ times the left degree) is larger than what can be implied from the size of the eigenvalue gap. As shown in [188] (see the proof in Theorem 4.19) random expanders have the needed expansion rate. Lately, constructive expanders with nearly maximal expansion rate were announced by Capalbo, Reingold, Vadhan and Wigderson in [?].

Reliable computation is also possible in a model of parallel computation that is much more regular than logic circuits: in cellular automata. We cannot present those results here: see for example the papers [79] and [81].

II. COMPUTER ALGEBRA

5. Algebra

First, in this chapter, we will discuss some of the basic concepts of algebra, such as fields, vector spaces and polynomials (Section 5.1). Our main focus will be the study of polynomial rings in one variable. These polynomial rings play a very important role in *constructive applications*. After this, we will outline the theory of finite fields, putting a strong emphasis on the problem of constructing them (Section 5.2) and on the problem of factoring polynomials over such fields (Section 5.3). Then we will study lattices and discuss the Lenstra-Lenstra-Lovász algorithm which can be used to find short lattice vectors (Section 5.4). We will present a polynomial time algorithm for the factorisation of polynomials with rational coefficients; this was the first notable application of the Lenstra-Lenstra-Lovász algorithm (Section 5.5).

5.1. Fields, vector spaces, and polynomials

In this section we will overview some important concepts related to rings and polynomials.

5.1.1. Ring theoretic concepts

We recall some definitions introduced in Chapters 31–33 of the textbook *Introduction to Algorithms*. In the sequel all cross references to Chapters 31–33 refer to results in that book.

A set S with at least two elements is called a ***ring***, if it has two binary operations, the addition, denoted by the $+$ sign, and the multiplication, denoted by the $\cdot$ sign. The elements of S form an Abelian group with respect to the addition, and they form a monoid (that is, a semigroup with an identity), whose identity element is denoted by 1, with respect to the multiplication. We assume that $1 \neq 0$. Further, the distributive properties also hold: for arbitrary elements $a,\ b,\ c \in S$ we have

$$a \cdot (b + c) = a \cdot b + a \cdot c \quad \text{and}$$

$$(b + c) \cdot a = b \cdot a + c \cdot a\ .$$

Being an Abelian group with respect to the addition means that the operation is associative, commutative, it has an identity element (denoted by 0), and every element has an inverse with respect to this identity. More precisely, these requirements are the following:
associative property: for all triples $a,\ b,\ c \in S$ we have $(a+b)+c = a+(b+c)$;
commutative property: for all pairs $a,\ b \in S$ we have $a+b = b+a$;
existence of the identity element: for the zero element 0 of S and for all elements a of S, we have $a+0 = 0+a = a$;
existence of the additive inverse: for all $a \in S$ there exists $b \in S$, such that $a+b = 0$.
It is easy to show that each of the elements a in S has a unique inverse. We usually denote the inverse of an element a by $-a$.

Concerning the multiplication, we require that it must be associative and that the multiplicative identity should exist. The ***identity*** of a ring S is the multiplicative identity of S. The usual name of the additive identity is ***zero***. We usually omit the $\cdot$ sign when writing the multiplication, for example we usually write ab instead of $a \cdot b$.

Example 5.1 *Rings.*
(i) The set $\mathbf{Z}$ of integers with the usual operations $+$ and $\cdot$.
(ii) The set $\mathbf{Z}_m$ of residue classes modulo m with respect to the addition and multiplication modulo m. (iii) The set $\mathbb{R}^{n\times n}$ of $(n \times n)$-matrices with real entries with respect to the addition and multiplication of matrices.

Let S_1 and S_2 be rings. A map $\phi : S_1 \to S_2$ is said to be a ***homomorphism***, if ϕ preserves the operations, in the sense that $\phi(a \pm b) = \phi(a) \pm \phi(b)$ and $\phi(ab) = \phi(a)\phi(b)$ holds for all pairs $a,\ b \in S_1$. A homomorphism ϕ is called an ***isomorphism***, if ϕ is a one-to-one correspondence, and the inverse is also a homomorphism. We say that the rings S_1 and S_2 are ***isomorphic***, if there is an isomorphism between them. If S_1 and S_2 are isomorphic rings, then we write $S_1 \cong S_2$. From an algebraic point of view, isomorphic rings can be viewed as identical.

For example the map $\phi : \mathbf{Z} \to \mathbf{Z}_6$ which maps an integer to its residue modulo 6 is a homomorphism: $\phi(13) = 1,\ \phi(5) = 5,\ \phi(22) = 4$, etc.

A useful and important ring theoretic construction is the ***direct sum.*** The direct sum of the rings S_1 and S_2 is denoted by $S_1 \oplus S_2$. The underlying set of the direct sum is $S_1 \times S_2$, that is, the set of ordered pairs (s_1, s_2) where $s_i \in S_i$. The operations are defined componentwise: for $s_i,\ t_i \in S_i$ we let

$$(s_1, s_2) + (t_1, t_2) := (s_1 + t_1, s_2 + t_2) \quad \text{and}$$

$$(s_1, s_2) \cdot (t_1, t_2) := (s_1 \cdot t_1, s_2 \cdot t_2)\ .$$

Easy calculation shows that $S_1 \oplus S_2$ is a ring with respect to the operations above. This construction can easily be generalised to more than two rings. In this case, the elements of the direct sum are the k-tuples, where k is the number of rings in the direct sum, and the operations are defined componentwise.

Fields A ring $\mathbb{F}$ is said to be a ***field***, if its non-zero elements form an Abelian group with respect to the multiplication. The multiplicative inverse of a non-zero element a is usually denoted a^{-1}.

The best-known examples of fields are the the sets of rational numbers, real numbers, and complex numbers with respect to the usual operations. We usually denote these fields by $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, respectively.

Another important class of fields consists of the fields $\mathbb{F}_p$ of p-elements where p is a prime number. The elements of $\mathbb{F}_p$ are the residue classes modulo p, and the operations are the addition and the multiplication defined on the residue classes. The distributive property can easily be derived from the distributivity of the integer operations. By Theorem 33.12, $\mathbb{F}_p$ is a group with respect to the addition, and, by Theorem 33.13, the set $\mathbb{F}_p^*$ of non-zero elements of $\mathbb{F}_p$ is a group with respect to the multiplication. In order to prove this latter claim, we need to use that p is a prime number.

Characteristic, prime field In an arbitrary field, we may consider the set of elements of the form $m \cdot 1$, that is, the set of elements that can be written as the sum $1 + \cdots + 1$ of m copies of the multiplicative identity where m is a positive integer. Clearly, one of the two possibilities must hold:
(a) none of the elements $m \cdot 1$ is zero;
(b) $m \cdot 1$ is zero for some $m \geq 1$.

In case (a) we say that $\mathbb{F}$ is a field with ***characteristic zero***. In case (b) the ***characteristic*** of $\mathbb{F}$ is the smallest $m \geq 1$ such that $m \cdot 1 = 0$. In this case, the number m must be a prime, for, if $m = rs$, then $0 = m \cdot 1 = rs \cdot 1 = (r \cdot 1)(s \cdot 1)$, and so either $r \cdot 1 = 0$ or $s \cdot 1 = 0$.

Suppose that P denotes the smallest subfield of $\mathbb{F}$ that contains 1. Then P is said to be the ***prime field*** of $\mathbb{F}$. In case (a) the subfield P consists of the elements $(m \cdot 1)(s \cdot 1)^{-1}$ where m is an integer and s is a positive integer. In this case, P is isomorphic to the field $\mathbb{Q}$ of rational numbers. The identification is obvious: $(m \cdot 1)(s \cdot 1)^{-1} \leftrightarrow m/s$.

In case (b) the characteristic is a prime number, and P is the set of elements $m \cdot 1$ where $0 \leq m < p$. In this case, P is isomorphic to the field $\mathbb{F}_p$ of residue classes modulo p.

Vector spaces Let $\mathbb{F}$ be a field. An additively written Abelian group V is said to be a ***vector space*** over $\mathbb{F}$, or simply an $\mathbb{F}$-vector space, if for all elements $a \in \mathbb{F}$ and $v \in V$, an element $av \in V$ is defined (in other words, $\mathbb{F}$ acts on V) and the following hold:

$$a(u+v) = au + av, \quad (a+b)u = au + bu \ ,$$

$$a(bu) = (ab)u, \quad 1u = u \ .$$

Here a, b are arbitrary elements of $\mathbb{F}$, the elements u, v are arbitrary in V, and the element 1 is the multiplicative identity of $\mathbb{F}$.

The space of $(m \times n)$-matrices over $\mathbb{F}$ is an important example of vector spaces. Their properties are studied in Chapter 31.

A vector space V over a field $\mathbb{F}$ is said to be ***finite-dimensional*** if there is a

collection $\{v_1, \ldots, v_n\}$ of finitely many elements in V such that each of the elements $v \in V$ can be written as a ***linear combination*** $v = a_1 v_1 + \cdots + a_n v_n$ for some $a_1, \ldots, a_n \in \mathbb{F}$. Such a set $\{v_i\}$ is called a ***generating set*** of V. The cardinality of the smallest generating set of V is referred to as the ***dimension*** of V over $\mathbb{F}$, denoted $\dim_{\mathbb{F}} V$. In a finite-dimensional vector space, a generating system containing $\dim_{\mathbb{F}} V$ elements is said to be a ***basis***.

A set $\{v_1, \ldots, v_k\}$ of elements of a vector space V is said to be ***linearly independent***, if, for $a_1, \ldots, a_k \in \mathbb{F}$, the equation $0 = a_1 v_1 + \cdots + a_k v_k$ implies $a_1 = \cdots = a_k = 0$. It is easy to show that a basis in V is a linearly independent set. An important property of linearly independent sets is that such a set can be extended to a basis of the vector space. The dimension of a vector space coincides with the cardinality of its largest linearly independent set.

A non-empty subset U of a vector space V is said to be a ***subspace*** of V, if it is an (additive) subgroup of V, and $au \in U$ holds for all $a \in \mathbb{F}$ and $u \in U$. It is obvious that a subspace can be viewed as a vector space.

The concept of homomorphisms can be defined for vector spaces, but in this context we usually refer to them as ***linear maps***. Let V_1 and V_2 be vector spaces over a common field $\mathbb{F}$. A map $\phi : V_1 \to V_2$ is said to be linear, if, for all $a,\ b \in \mathbb{F}$ and $u,\ v \in V_1$, we have

$$\phi(au + bv) = a\phi(u) + b\phi(v) \ .$$

The linear mapping ϕ is an ***isomorphism*** if ϕ is a one-to-one correspondence and its inverse is also a homomorphism. Two vector spaces are said to be isomorphic if there is an isomorphism between them.

Lemma 5.1 *Suppose that $\phi : V_1 \to V_2$ is a linear mapping. Then $U = \phi(V_1)$ is a subspace in V_2. If ϕ is one-to-one, then $\dim_{\mathbb{F}} U = \dim_{\mathbb{F}} V_1$. If, in this case, $\dim_{\mathbb{F}} V_1 = \dim_{\mathbb{F}} V_2 < \infty$, then $U = V_2$ and the mapping ϕ is an isomorphism.*

Proof As

$$\phi(u) \pm \phi(v) = \phi(u \pm v) \quad \text{and} \quad a\phi(u) = \phi(au),$$

we obtain that U is a subspace. Further, it is clear that the images of the elements of a generating set of V_1 form a generating set for U. Let us now suppose that ϕ is one-to-one. In this case, the image of a linearly independent subset of V_1 is linearly independent in V_2. It easily follows from these observations that the image of a basis of V_1 is a basis of U, and so $\dim_{\mathbb{F}} U = \dim_{\mathbb{F}} V_1$. If we assume, in addition, that $\dim_{\mathbb{F}} V_2 = \dim_{\mathbb{F}} V_1$, then a basis of U is also a basis of V_2, as it is a linearly independent set, and so it can be extended to a basis of V_2. Thus $U = V_2$ and the mapping ϕ must be a one-to-one correspondence. It is easy to see, and is left to the reader, that ϕ^{-1} is a linear mapping. ∎

The ***direct sum*** of vector spaces can be defined similarly to the direct sum of rings. The direct sum of the vector spaces V_1 and V_2 is denoted by $V_1 \oplus V_2$. The underlying set of the direct sum is $V_1 \times V_2$, and the addition and the action of the field $\mathbb{F}$ are defined componentwise. It is easy to see that

$$\dim_{\mathbb{F}} (V_1 \oplus V_2) = \dim_{\mathbb{F}} V_1 + \dim_{\mathbb{F}} V_2 \ .$$

Finite multiplicative subgroups of fields Let $\mathbb{F}$ be a field and let $G \subseteq \mathbb{F}$ be a finite multiplicative subgroup of $\mathbb{F}$. That is, the set G contains finitely many elements of $\mathbb{F}$, each of which is non-zero, G is closed under multiplication, and the multiplicative inverse of an element of G also lies in G. We aim to show that the group G is cyclic, that is, G can be generated by a single element. The main concepts related to cyclic groups can be found in Section 33.3.4. $\mathrm{ord}(a)$ of an element $a \in G$ is the smallest positive integer k such that $a^k = 1$.

The cyclic group generated by an element a is denoted by $\langle a \rangle$. Clearly, $|\langle a \rangle| = \mathrm{ord}(a)$, and an element a^i generates the group $\langle a \rangle$ if and only if i and n are relatively prime. Hence the group $\langle a \rangle$ has exactly $\phi(n)$ generators where ϕ is Euler's totient function (see Subsection 33.3.2).

The following identity is valid for an arbitrary integer n:

$$\sum_{d|n} \phi(d) = n.$$

Here the summation index d runs through all positive divisors of n. In order to verify this identity, consider all the rational numbers i/n with $1 \leq i \leq n$. The number of these is exactly n. After simplifying these fractions, they will be of the form j/d where d is a positive divisor of n. A fixed denominator d will occur exactly $\phi(d)$ times.

Theorem 5.2 *Suppose that $\mathbb{F}$ is a field and let G be a finite multiplicative subgroup of $\mathbb{F}$. Then there exists an element $a \in G$ such that $G = \langle a \rangle$.*

Proof Suppose that $|G| = n$. Lagrange's theorem (Theorem 33.15) implies that the order of an element $b \in G$ is a divisor of n. We claim, for an arbitrary d, that there are at most $\phi(d)$ elements in $\mathbb{F}$ with order d. The elements with order d are roots of the polynomial $x^d - 1$. If $\mathbb{F}$ has an element b with order d, then, by Lemma 5.5, $x^d - 1 = (x - b)(x - b^2) \cdots (x - b^d)$ (the lemma will be verified later). Therefore all the elements of $\mathbb{F}$ with order d are contained in the group $\langle b \rangle$, which, in turn, contains exactly $\phi(d)$ elements of order d.

If G had no element of order n, then the order of each of the elements of G would be a proper divisor of n. In this case, however, using the identity above and the fact that $\phi(n) > 0$, we obtain

$$n = |G| \leq \sum_{d|n,\ d<n} \phi(d) < n\ ,$$

which is a contradiction. ∎

5.1.2. Polynomials

Suppose that $\mathbb{F}$ is a field and that $a_0, \ldots, a_n$ are elements of $\mathbb{F}$. Recall that an expression of the form

$$f = f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n,$$

where x is an indeterminate, is said to be a ***polynomial*** over $\mathbb{F}$ (see Chapter 32). The scalars a_i are the ***coefficients*** of the polynomial f. The degree of the zero

polynomial is zero, while the ***degree*** of a non-zero polynomial f is the largest index j such that $a_j \neq 0$. The degree of f is denoted by $\deg f$.

The set of all polynomials over $\mathbb{F}$ in the indeterminate x is denoted by $\mathbb{F}[x]$. If

$$f = f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

and

$$g = g(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_n x^n$$

are polynomials with degree not larger than n, then their sum is defined as the polynomial

$$h = h(x) = f + g = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$$

whose coefficients are $c_i = a_i + b_i$.

The product fg of the polynomials f and g is defined as the polynomial

$$fg = d_0 + d_1 x + d_2 x^2 + \cdots + d_{2n} x^{2n}$$

with degree at most $2n$ whose coefficients are given by the equations $d_j = \sum_{k=0}^{j} a_k b_{j-k}$. On the right-hand side of these equations, the coefficients with index greater than n are considered zero. Easy computation shows that $\mathbb{F}[x]$ is a commutative ring with respect to these operations. It is also straightforward to show that $\mathbb{F}[x]$ has no ***zero divisors***, that is, whenever $fg = 0$, then either $f = 0$ or $g = 0$.

Division with remainder and divisibility The ring $\mathbb{F}[x]$ of polynomials over $\mathbb{F}$ is quite similar, in many ways, to the ring $\mathbf{Z}$ of integers. One of their similar features is that the procedure of division with remainder can be performed in both rings.

Lemma 5.3 *Let $f(x),\ g(x) \in \mathbb{F}[x]$ be polynomials such that $g(x) \neq 0$. Then there there exist polynomials $q(x)$ and $r(x)$ such that*

$$f(x) = q(x)g(x) + r(x)\ ,$$

and either $r(x) = 0$ or $\deg r(x) < \deg g(x)$. Moreover, the polynomials q and r are uniquely determined by these conditions.

Proof We verify the claim about the existence of the polynomials q and r by induction on the degree of f. If $f = 0$ or $\deg f < \deg g$, then the assertion clearly holds. Let us suppose, therefore, that $\deg f \geq \deg g$. Then subtracting a suitable multiple $q^*(x)g(x)$ of g from f, we obtain that the degree of $f_1(x) = f(x) - q^*(x)g(x)$ is smaller than $\deg f(x)$. Then, by the induction hypothesis, there exist polynomials q_1 and r_1 such that

$$f_1(x) = q_1(x)g(x) + r_1(x)$$

and either $r_1 = 0$ or $\deg r_1 < \deg g$. It is easy to see that, in this case, the polynomials $q(x) = q_1(x) + q^*(x)$ and $r(x) = r_1(x)$ are as required.

It remains to show that the polynomials q and r are unique. Let Q and R be polynomials, possibly different from q and r, satisfying the assertions of the lemma. That is, $f(x) = Q(x)g(x) + R(x)$, and so $(q(x) - Q(x))g(x) = R(x) - r(x)$. If the

polynomial on the left-hand side is non-zero, then its degree is at least $\deg g$, while the degree of the polynomial on the right-hand side is smaller than $\deg g$. This, however, is not possible. ∎

Let R be a commutative ring with a multiplicative identity and without zero divisors, and set $R^* := R \setminus \{0\}$. The ring R is said to be a ***Euclidean ring*** if there is a function $\phi : R^* \to \mathbb{N}$ such that $\phi(ab) \geq \phi(a)\phi(b)$, for all $a,\ b \in R^*$; and, further, if $a \in R$, $b \in R^*$, then there are elements $q,\ r \in R$ such that $a = qb + r$, and if $r \neq 0$, then $\phi(r) < \phi(b)$. The previous lemma shows that $\mathbb{F}[x]$ is a Euclidean ring where the rôle of the function ϕ is played by the degree function.

The concept of ***divisibility*** in $\mathbb{F}[x]$ can be defined similarly to the definition of the corresponding concept in the ring of integers. A polynomial $g(x)$ is said to be a ***divisor*** of a polynomial $f(x)$ (the notation is $g \mid f$), if there is a polynomial $q(x) \in \mathbb{F}[x]$ such that $f(x) = q(x)g(x)$. The non-zero elements of $\mathbb{F}$, which are clearly divisors of each of the polynomials, are called the ***trivial divisors*** or ***units***. A non-zero polynomial $f(x) \in \mathbb{F}[x]$ is said to be ***irreducible***, if whenever $f(x) = q(x)g(x)$ with $q(x),\ g(x) \in \mathbb{F}[x]$, then either q or g is a unit.

Two polynomials $f,\ g \in \mathbb{F}[x]$ are called ***associates***, if there is some $u \in \mathbb{F}^*$ such that $f(x) = ug(x)$.

Using Lemma 5.3, one can easily prove the unique factorisation theorem in the ring of polynomials following the argument of the proof of the corresponding theorem in the ring of integers (see Section 33.1). The role of the absolute value of integers is played by the degree of polynomials.

Theorem 5.4 *An arbitrary polynomial $0 \neq f \in \mathbb{F}[x]$ can be written in the form*

$$f(x) = uq_1(x)^{e_1} \cdots q_r(x)^{e_r},$$

where $u \in \mathbb{F}^*$ is a unit, the polynomials $q_i \in \mathbb{F}[x]$ are pairwise non-associate and irreducible, and, further, the numbers e_i are positive integers. Furthermore, this decomposition is essentially unique in the sense that whenever

$$f(x) = UQ_1(x)^{d_1} \cdots Q_s(x)^{d_s}$$

is another such decomposition, then $r = s$, and, after possibly reordering the factors Q_i, the polynomials q_i and Q_i are associates, and moreover $d_i = e_i$ for all $1 \leq i \leq r$.

Two polynomials are said to be ***relatively prime,*** if they have no common irreducible divisors.

A scalar $a \in \mathbb{F}$ is a ***root*** of a polynomial $f \in \mathbb{F}[x]$, if $f(a) = 0$. Here the value $f(a)$ is obtained by substituting a into the place of x in $f(x)$.

Lemma 5.5 *Suppose that $a \in \mathbb{F}$ is a root of a polynomial $f(x) \in \mathbb{F}[x]$. Then there exists a polynomial $g(x) \in \mathbb{F}[x]$ such that $f(x) = (x - a)g(x)$. Hence the polynomial f may have at most $\deg f$ roots.*

Proof By Lemma 5.3, there exists $g(x) \in \mathbb{F}[x]$ and $r \in \mathbb{F}$ such that $f(x) = (x - a)g(x) + r$. Substituting a for x, we find that $r = 0$. The second assertion now follows by induction on $\deg f$ from the fact that the roots of g are also roots of f. ∎

The cost of the operations with polynomials Suppose that $f(x),\ g(x) \in \mathbb{F}[x]$ are polynomials of degree at most n. Then the polynomials $f(x) \pm g(x)$ can obviously be computed using $O(n)$ field operations. The product $f(x)g(x)$ can be obtained, using its definition, by $O(n^2)$ field operations. If the Fast Fourier Transform can be performed over $\mathbb{F}$, then the multiplication can be computed using only $O(n \lg n)$ field operations (see Theorem 32.2). For general fields, the cost of the fastest known multiplication algorithms for polynomials (for instance the Schönhage–Strassen-method) is $O(n \lg n \lg \lg n)$, that is, $\widetilde{O}(n)$ field operations.

The division with remainder, that is, determining the polynomials $q(x)$ and $r(x)$ for which $f(x) = q(x)g(x) + r(x)$ and either $r(x) = 0$ or $\deg r(x) < \deg g(x)$, can be performed using $O(n^2)$ field operations following the straightforward method outlined in the proof of Lemma 5.3. There is, however, an algorithm (the Sieveking–Kung algorithm) for the same problem using only $\widetilde{O}(n)$ steps. The details of this algorithm are, however, not discussed here.

Congruence, residue class ring Let $f(x) \in \mathbb{F}[x]$ with $\deg f = n > 0$, and let $g,\ h \in \mathbb{F}[x]$. We say that g is ***congruent*** to h modulo f, or simply $g \equiv h \pmod{f}$, if f divides the polynomial $g - h$. This concept of congruence is similar to the corresponding concept introduced in the ring of integers (see 33.3.2). It is easy to see from the definition that the relation $\equiv$ is an equivalence relation on the set $\mathbb{F}[x]$. Let $[g]_f$ (or simply $[g]$ if f is clear from the context) denote the equivalence class containing g. From Lemma 5.3 we obtain immediately, for each g, that there is a unique $r \in \mathbb{F}[x]$ such that $[g] = [r]$, and either $r = 0$ (if f divides g) or $\deg r < n$. This polynomial r is called the ***representative*** of the class $[g]$. The set of equivalence classes is traditionally denoted by $\mathbb{F}[x]/(f)$.

Lemma 5.6 *Let $f,\ f_1,\ f_2,\ g_1,\ g_2 \in \mathbb{F}[x]$ and let $a \in \mathbb{F}$. Suppose that $f_1 \equiv f_2 \pmod{f}$ and $g_1 \equiv g_2 \pmod{f}$. Then*

$$f_1 + g_1 \equiv f_2 + g_2 \pmod{f} ,$$

$$f_1 g_1 \equiv f_2 g_2 \pmod{f} ,$$

and

$$af_1 \equiv af_2 \pmod{f} .$$

Proof The first congruence is valid, as

$$(f_1 + g_1) - (f_2 + g_2) = (f_1 - f_2) + (g_1 - g_2) ,$$

and the right-hand side of this is clearly divisible by f. The second and the third congruences follow similarly from the identities

$$f_1 g_1 - f_2 g_2 = (f_1 - f_2)g_1 + (g_1 - g_2)f_2$$

and

$$af_1 - af_2 = a(f_1 - f_2) ,$$

respectively. ∎

The previous lemma makes it possible to define the sum and the product of two congruence classes $[g]_f$ and $[h]_f$ as $[g]_f + [h]_f := [g+h]_f$ and $[g]_f[h]_f := [gh]_f$, respectively. The lemma claims that the sum and the product are independent of the choice of the congruence class representatives. The same way, we may define the action of $\mathbb{F}$ on the set of congruence classes: we set $a[g]_f := [ag]_f$.

Theorem 5.7 *Suppose that* $f(x) \in \mathbb{F}[x]$ *and that* $\deg f = n > 0$.
(i) The set of residue classes $\mathbb{F}[x]/(f)$ *is a commutative ring with an identity under the operations* $+$ *and* $\cdot$ *defined above.*
(ii) The ring $\mathbb{F}[x]/(f)$ *contains the field* $\mathbb{F}$ *as a subring, and it is an* n*-dimensional vector space over* $\mathbb{F}$*. Further, the residue classes* $[1],\ [x], \ldots, [x^{n-1}]$ *form a basis of* $\mathbb{F}[x]/(f)$.
(iii) If f *is an irreducible polynomial in* $\mathbb{F}[x]$*, then* $\mathbb{F}[x]/(f)$ *is a field.*

Proof (i) The fact that $\mathbb{F}[x]/(f)$ is a ring follows easily from the fact that $\mathbb{F}[x]$ is a ring. Let us, for instance, verify the distributive property:

$$[g]([h_1]+[h_2]) = [g][h_1+h_2] = [g(h_1+h_2)] = [gh_1+gh_2] = [gh_1]+[gh_2] = [g][h_1]+[g][h_2] \ .$$

The zero element of $\mathbb{F}[x]/(f)$ is the class $[0]$, the additive inverse of the class $[g]$ is the class $[-g]$, while the multiplicative identity element is the class $[1]$. The details are left to the reader.

(ii) The set $\{[a] \mid a \in \mathbb{F}\}$ is a subring isomorphic to $\mathbb{F}$. The correspondence is obvious: $a \leftrightarrow [a]$. By part (i), $\mathbb{F}[x]/(f)$ is an additive Abelian group, and the action of $\mathbb{F}$ satisfies the vector space axioms. This follows from the fact that the polynomial ring is itself a vector space over $\mathbb{F}$. Let us, for example, verify the distributive property:

$$a([h_1]+[h_2]) = a[h_1+h_2] = [a(h_1+h_2)] = [ah_1+ah_2] = [ah_1]+[ah_2] = a[h_1]+a[h_2] \ .$$

The other properties are left to the reader.

We claim that the classes $[1],\ [x], \ldots, [x^{n-1}]$ are linearly independent. For, if

$$[0] = a_0[1] + a_1[x] + \cdots + a_{n-1}[x^{n-1}] = [a_0 + a_1x + \cdots + a_{n-1}x^{n-1}] \ ,$$

then $a_0 = \cdots = a_{n-1} = 0$, as the zero polynomial is the unique polynomial with degree less than n that is divisible by f. On the other hand, for a polynomial g, the degree of the class representative of $[g]$ is less than n. Thus the class $[g]$ can be expressed as a linear combination of the classes $[1], [x], \ldots, [x^{n-1}]$. Hence the classes $[1], [x], \ldots, [x^{n-1}]$ form a basis of $\mathbb{F}[x]/(f)$, and so $\dim_{\mathbb{F}} \mathbb{F}[x]/(f) = n$.

(iii) Suppose that f is irreducible. First we show that $\mathbb{F}[x]/(f)$ has no zero divisors. If $[0] = [g][h] = [gh]$, then f divides gh, and so f divides either g or h. That is, either $[g] = 0$ or $[h] = 0$. Suppose now that $g \in \mathbb{F}[x]$ with $[g] \neq [0]$. We claim that the classes $[g][1], [g][x], \ldots, [g][x^{n-1}]$ are linearly independent. Indeed, an equation $[0] = a_0[g][1]+\cdots+a_{n-1}[g][x^{n-1}]$ implies $[0] = [g][a_0+\cdots+a_{n-1}x^{n-1}]$, and, in turn, it also yields that $a_0 = \cdots = a_{n-1} = 0$. Therefore the classes $[g][1], [g][x], \ldots, [g][x^{n-1}]$ form a basis of $\mathbb{F}[x]/(f)$. Hence there exist coefficients $b_i \in \mathbb{F}$ for which

$$[1] = b_0[g][1] + \cdots + b_{n-1}[g][x^{n-1}] = [g][b_0 + \cdots + b_{n-1}x^{n-1}] \ .$$

Thus we find that the class $[0] \neq [g]$ has a multiplicative inverse, and so $\mathbb{F}[x]/(f)$ is a field, as required. ■

We note that the converse of part (iii) of the previous theorem is also true, and its proof is left to the reader (Exercise 5.1-1).

Example 5.2 We usually represent the elements of the residue class ring $\mathbb{F}[x]/(f)$ by their representatives, which are polynomials with degree less than $\deg f$.

1. Suppose that $\mathbb{F} = \mathbb{F}_2$ is the field of two elements, and let $f(x) = x^3 + x + 1$. Then the ring $\mathbb{F}[x]/(f)$ has 8 elements, namely

$$[0],\ [1],\ [x],\ [x+1],\ [x^2],\ [x^2+1],\ [x^2+x],\ [x^2+x+1].$$

Practically speaking, the addition between the classes is the is addition of polynomials. For instance

$$[x^2+1] + [x^2+x] = [x+1]\ .$$

When computing the product, we compute the product of the representatives, and substitute it (or reduce it) with its remainder after dividing by f. For instance,

$$[x^2+1] \cdot [x^2+x] = [x^4+x^3+x^2+x] = [x+1]\ .$$

The polynomial f is irreducible over $\mathbb{F}_2$, since it has degree 3, and has no roots. Hence the residue class ring $\mathbb{F}[x]/(f)$ is a field.

2. Let $\mathbb{F} = \mathbb{R}$ and let $f(x) = x^2 - 1$. The elements of the residue class ring are the classes of the form $[ax + b]$ where $a,\ b \in \mathbb{R}$. The ring $\mathbb{F}[x]/(f)$ is not a field, since f is not irreducible. For instance, $[x+1][x-1] = [0]$.

Lemma 5.8 *Let $\mathbb{L}$ be a field containing a field $\mathbb{F}$ and let $\alpha \in \mathbb{L}$.*
(i) If $\mathbb{L}$ is finite-dimensional as a vector space over $\mathbb{F}$, then there is a non-zero polynomial $f \in \mathbb{F}[x]$ such that α is a root of f.
(ii) Assume that there is a polynomial $f \in \mathbb{F}[x]$ with $f(\alpha) = 0$, and let g be such a polynomial with minimal degree. Then the polynomial g is irreducible in $\mathbb{F}[x]$. Further, if $h \in \mathbb{F}[x]$ with $h(\alpha) = 0$ then g is a divisor of h.

Proof (i) For a sufficiently large n, the elements $1, \alpha, \ldots, \alpha^n$ are linearly dependent over $\mathbb{F}$. A linear dependence gives a polynomial $0 \neq f \in \mathbb{F}[x]$ such that $f(\alpha) = 0$.

(ii) If $g = g_1 g_2$, then, as $0 = g(\alpha) = g_1(\alpha) g_2(\alpha)$, the element α is a root of either g_1 or g_2. As g was chosen to have minimal degree, one of the polynomials $g_1,\ g_2$ is a unit, and so g is irreducible. Finally, let $h \in \mathbb{F}[x]$ such that $h(\alpha) = 0$. Let $q,\ r \in \mathbb{F}[x]$ be the polynomials as in Lemma 5.3 for which $h(x) = q(x)g(x) + r(x)$. Substituting α for x into the last equation, we obtain $r(\alpha) = 0$, which is only possible if $r = 0$. ■

Definition 5.9 *The polynomial $g \in \mathbb{F}[x]$ in the last lemma is said to be a* ***minimal polynomial*** *of α.*

It follows from the previous lemma that the minimal polynomial is unique up to a scalar multiple. It will often be helpful to assume that the leading coefficient (the coefficient of the term with the highest degree) of the minimal polynomial g is 1.

Corollary 5.10 *Let $\mathbb{L}$ be a field containing $\mathbb{F}$, and let $\alpha \in \mathbb{L}$. Suppose that $f \in \mathbb{F}[x]$ is irreducible and that $f(\alpha) = 0$. Then f is a minimal polynomial of α.*

Proof Suppose that g is a minimal polynomial of α. By the previous lemma, $g \mid f$ and g is irreducible. This is only possible if the polynomials f and g are associates. ■

Let $\mathbb{L}$ be a field containing $\mathbb{F}$ and let $\alpha \in \mathbb{L}$. Let $\mathbb{F}(\alpha)$ denote the smallest subfield of $\mathbb{L}$ that contains $\mathbb{F}$ and α.

Theorem 5.11 *Let $\mathbb{L}$ be a field containing $\mathbb{F}$ and let $\alpha \in \mathbb{L}$. Suppose that $f \in \mathbb{F}[x]$ is a minimal polynomial of α. Then the field $\mathbb{F}(\alpha)$ is isomorphic to the field $\mathbb{F}[x]/(f)$. More precisely, there exists an isomorphism $\phi : \mathbb{F}[x]/(f) \to \mathbb{F}(\alpha)$ such that $\phi(a) = a$, for all $a \in \mathbb{F}$, and $\phi([x]_f) = \alpha$. The map ϕ is also an isomorphism of vector spaces over $\mathbb{F}$, and so $\dim_{\mathbb{F}} \mathbb{F}(\alpha) = \deg f$.*

Proof Let us consider the map $\psi : \mathbb{F}[x] \to \mathbb{L}$, which maps a polynomial $g \in \mathbb{F}[x]$ into $g(\alpha)$. This is clearly a ring homomorphism, and $\psi(\mathbb{F}[x]) \subseteq \mathbb{F}(\alpha)$. We claim that $\psi(g) = \psi(h)$ if and only if $[g]_f = [h]_f$. Indeed, $\psi(g) = \psi(h)$ holds if and only if $\psi(g - h) = 0$, that is, if $g(\alpha) - h(\alpha) = 0$, which, by Lemma 5.8, is equivalent to $f \mid g - h$, and this amounts to saying that $[g]_f = [h]_f$. Suppose that ϕ is the map $\mathbb{F}[x]/(f) \to \mathbb{F}(\alpha)$ induced by ψ, that is, $\phi([g]_f) := \psi(g)$. By the argument above, the map ϕ is one-to-one. Routine computation shows that ϕ is a ring, and also a vector space, homomorphism. As $\mathbb{F}[x]/(f)$ is a field, its homomorphic image $\phi(\mathbb{F}[x]/(f))$ is also a field. The field $\phi(\mathbb{F}[x]/(f))$ contains $\mathbb{F}$ and α, and so necessarily $\phi(\mathbb{F}[x]/(f)) = \mathbb{F}(\alpha)$. ■

Euclidean algorithm and the greatest common divisor Let $f(x),\ g(x) \in \mathbb{F}[x]$ be polynomials such that $g(x) \neq 0$. Set $f_0 = f$, $f_1 = g$ and define the polynomials q_i and f_i using division with reminder as follows:

$$f_0(x) = q_1(x)f_1(x) + f_2(x) \ ,$$

$$f_1(x) = q_2(x)f_2(x) + f_3(x) \ ,$$

$$\vdots$$

$$f_{k-2}(x) = q_{k-1}(x)f_{k-1}(x) + f_k(x) \ ,$$

$$f_{k-1}(x) = q_k(x)f_k(x) + f_{k+1}(x) \ .$$

Note that if $1 < i < k$ then $\deg f_{i+1}$ is smaller than $\deg f_i$. We form this sequence of polynomials until we obtain that $f_{k+1} = 0$. By Lemma 5.3, this defines a finite process. Let n be the maximum of $\deg f$ and $\deg g$. As, in each step, we decrease the degree of the polynomials, we have $k \leq n + 1$. The computation outlined above is usually referred to as the ***Euclidean algorithm***. A version of this algorithm for the ring of integers is described in Section 33.2.

We say that the polynomial $h(x)$ is the ***greatest common divisor*** of the polynomials $f(x)$ and $g(x)$, if $h(x) \mid f(x)$, $h(x) \mid g(x)$, and, if a polynomial $h_1(x)$ is a divisor of f and g, then $h_1(x)$ is a divisor of $h(x)$. The usual notation for the greatest common divisor of $f(x)$ and $g(x)$ is $\gcd(f(x), g(x))$. It follows from Theorem 5.4 that $\gcd(f(x), g(x))$ exists and it is unique up to a scalar multiple.

Theorem 5.12 *Suppose that* $f(x),\ g(x) \in \mathbb{F}[x]$ *are polynomials, that* $g(x) \neq 0$*, and let* n *be the maximum of* $\deg f$ *and* $\deg g$*. Assume, further, that the number* k *and the polynomial* f_k *are defined by the procedure above. Then*
(i) $\gcd(f(x), g(x)) = f_k(x)$*.*
(ii) There are polynomials $F(x),\ G(x)$ *with degree at most* n *such that*

$$f_k(x) = F(x)f(x) + G(x)g(x)\ . \tag{5.1}$$

(iii) With a given input $f,\ g$*, the polynomials* $F(x),\ G(x),\ f_k(x)$ *can be computed using* $O(n^3)$ *field operations in* $\mathbb{F}$*.*

Proof (i) Going backwards in the Euclidean algorithm, it is easy to see that the polynomial f_k divides each of the f_i, and so it divides both f and g. The same way, if a polynomial $h(x)$ divides f and g, then it divides f_i, for all i, and, in particular, it divides f_k. Thus $\gcd(f(x), g(x)) = f_k(x)$.

(ii) The claim is obvious if $f = 0$, and so we may assume without loss of generality that $f \neq 0$. Starting at the beginning of the Euclidean sequence, it is easy to see that there are polynomials $F_i(x),\ G_i(x) \in \mathbb{F}[x]$ such that

$$F_i(x)f(x) + G_i(x)g(x) = f_i(x)\ . \tag{5.2}$$

We observe that (5.2) also holds if we substitute $F_i(x)$ by its remainder $F_i^*(x)$ after dividing by g and substitute $G_i(x)$ by its remainder $G_i^*(x)$ after dividing by f. In order to see this, we compute

$$F_i^*(x)f(x) + G_i^*(x)g(x) \equiv f_i(x) \pmod{f(x)g(x)}\ ,$$

and notice that the degree of the polynomials on both sides of this congruence is smaller than $(\deg f)(\deg g)$. This gives

$$F_i^*(x)f(x) + G_i^*(x)g(x) = f_i(x)\ .$$

(iii) Once we determined the polynomials f_{i-1}, f_i, F_i^* and G_i^*, the polynomials f_{i+1}, F_{i+1}^* and G_{i+1}^* can be obtained using $O(n^2)$ field operations in $\mathbb{F}$. Initially we have $F_1^* = 1$ and $G_2^* = -q_1$. As $k \leq n+1$, the claim follows. ∎

Remark. Traditionally, the Euclidean algorithm is only used to compute the greatest common divisor. The version that also computes the polynomials $F(x)$ and $G(x)$ in (5.1) is usually called the extended Euclidean algorithm. In Chapter **??** the reader can find a discussion of the Euclidean algorithm for polynomials. It is relatively easy to see that the polynomials $f_k(x)$, $F(x)$, and $G(x)$ in (5.1) can, in fact, be computed using $O(n^2)$ field operations. The cost of the asymptotically best method is $\widetilde{O}(n)$.

The derivative of a polynomial is often useful when investigating multiple factors. The ***derivative*** of the polynomial

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n \in \mathbb{F}[x]$$

is the polynomial

$$f'(x) = a_1 + 2a_2x + \cdots + na_nx^{n-1}\ .$$

It follows immediately from the definition that the map $f(x) \mapsto f'(x)$ is an $\mathbb{F}$-linear mapping $\mathbb{F}[x] \to \mathbb{F}[x]$. Further, for $f(x),\ g(x) \in \mathbb{F}[x]$ and $a \in \mathbb{F}$, the equations $(f(x)+g(x))' = f'(x)+g'(x)$ and $(af(x))' = af'(x)$ hold. The derivative of a product can be computed using the ***Leibniz rule***: for all $f(x),\ g\ (x) \in \mathbb{F}[x]$ we have $(f(x)g(x))' = f'(x)g(x)+f(x)g'(x)$. As the derivation is a linear map, in order to show that the Leibniz rule is valid, it is enough to verify it for polynomials of the form $f(x) = x^i$ and $g(x) = x^j$. It is easy to see that, for such polynomials, the Leibniz rule is valid.

The derivative $f'(x)$ is sensitive to multiple factors in the irreducible factorisation of $f(x)$.

Lemma 5.13 *Let $\mathbb{F}$ be an arbitrary field, and assume that $f(x) \in \mathbb{F}[x]$ and $f(x) = u^k(x)v(x)$ where $u(x),\ v(x) \in \mathbb{F}[x]$. Then $u^{k-1}(x)$ divides the derivative $f'(x)$ of the polynomial $f(x)$.*

Proof Using induction on k and the Leibniz rule, we find $(u^k(x))' = ku^{k-1}(x)u'(x)$. Thus, applying the Leibniz rule again, $f'(x) = u^{k-1}(x)(ku'(x)v(x) + u^k(x)v'(x))$. Hence $u^{k-1}(x) \mid f'(x)$. ∎

In many cases the converse of the last lemma also holds.

Lemma 5.14 *Let $\mathbb{F}$ be an arbitrary field, and assume that $f(x) \in \mathbb{F}[x]$ and $f(x) = u(x)v(x)$ where the polynomials $u(x)$ and $v(x)$ are relatively prime. Suppose further that $u'(x) \neq 0$ (for instance $\mathbb{F}$ has characteristic 0 and $u(x)$ is non-constant). Then the derivative $f'(x)$ is not divisible by $u(x)$.*

Proof By the Leibniz rule, $f'(x) = u(x)v'(x) + u'(x)v(x) \equiv u'(x)v(x) \pmod{u(x)}$. Since $\deg u'(x)$ is smaller than $\deg u(x)$, we obtain that $u'(x)$ is not divisible by $u(x)$, and neither is the product $u'(x)v(x)$, as $u(x)$ and $v(x)$ are relatively prime. ∎

The Chinese remainder theorem for polynomials Using the following theorem, the ring $\mathbb{F}[x]/(f)$ can be assembled from rings of the form $\mathbb{F}[x]/(g)$ where $g \mid f$.

Theorem 5.15 *(Chinese remainder theorem for polynomials) Let $f_1, \ldots, f_k \in \mathbb{F}[x]$ pairwise relatively prime polynomials with positive degree and set $f = f_1 \cdots f_k$. Then the rings $\mathbb{F}[x]/(f)$ and $\mathbb{F}[x]/(f_1) \oplus \cdots \oplus \mathbb{F}[x]/(f_k)$ are isomorphic. The mapping realizing the isomorphism is*

$$\phi : [g]_f \mapsto ([g]_{f_1}, \ldots, [g]_{f_k}), \quad g \in \mathbb{F}[x]\ .$$

Proof First we note that the map ϕ is well-defined. If $h \in [g]_f$, then $h = g + f^*f$, which implies that h and g give the same remainder after division by the polynomial f_i, that is, $[h]_{f_i} = [g]_{f_i}$.

The mapping ϕ is clearly a ring homomorphism, and it is also a linear mapping between two vector spaces over $\mathbb{F}$. The mapping ϕ is one-to-one; for, if $\phi([g]) = \phi([h])$, then $\phi([g-h]) = (0, \ldots, 0)$, that is, $f_i \mid g-h$ $(1 \leq i \leq k)$, which gives $f \mid g-h$ and $[g] = [h]$.

The dimensions of the vector spaces $\mathbb{F}[x]/(f)$ and $\mathbb{F}[x]/(f_1) \oplus \cdots \oplus \mathbb{F}[x]/(f_k)$ coincide: indeed, both spaces have dimension $\deg f$. Lemma 5.1 implies that ϕ is an

isomorphism between vector spaces. It only remains to show that ϕ^{-1} preserves the multiplication; this, however, is left to the reader. ∎

Exercises

5.1-1 Let $f \in \mathbb{F}[x]$ be polynomial. Show that the residue class ring $\mathbb{F}[x]/(f)$ has no zero divisors if and only if f is irreducible.

5.1-2 Let R be a commutative ring with an identity. A subset $I \subseteq R$ is said to be an ***ideal***, if I is an additive subgroup, and $a \in I$, $b \in R$ imply $ab \in I$. Show that R is a field if and only if its ideals are exactly $\{0\}$ and R.

5.1-3 Let $a_1, \ldots, a_k \in R$. Let $(a_1, \ldots, a_k)$ denote the smallest ideal in R that contains the elements a_i. Show that $(a_1, \ldots, a_k)$ always exists, and it consists of the elements of the form $b_1a_1 + b_2a_2 + \cdots + b_ka_k$ where $b_1, \ldots, b_k \in R$.

5.1-4 A commutative ring R with an identity and without zero divisors is said to be a ***principal ideal domain*** if, for each ideal I of R, there is an element $a \in I$ such that (using the notation of the previous exercise) $I = (a)$. Show that $\mathbf{Z}$ and $\mathbb{F}[x]$ where $\mathbb{F}$ is a field, are principal ideal domains.

5.1-5 Suppose that S is a commutative ring with an identity, that I an ideal in S, and that $a,\ b \in S$. Define a relation on S as follows: $a \equiv b \pmod{I}$ if and only if $a - b \in I$. Verify the following:

a.) The relation $\equiv$ is an equivalence relation on S.

b.) Let $[a]_I$ denote the equivalence class containing an element a, and let S/I denote the set of equivalence classes. Set $[a]_I + [b]_I := [a + b]_I$, and $[a]_I[b]_I := [ab]_I$. Show that, with respect to these operations, S/I is a commutative ring with an identity. *Hint.* Follow the argument in the proof of Theorem 5.7.

5.1-6 Let $\mathbb{F}$ be a field and let $f(x),\ g(x) \in \mathbb{F}[x]$ such that $\gcd(f(x), g(x)) = 1$. Show that there exists a polynomial $h(x) \in \mathbb{F}[x]$ such that $h(x)g(x) \equiv 1 \pmod{f(x)}$. *Hint.* Use the Euclidean algorithm.

5.2. Finite fields

Finite fields, that is, fields with a finite number of elements, play an important rôle in mathematics and in several of its application areas, for instance, in computing. They are also fundamental in many important constructions. In this section we summarise the most important results in the theory of finite fields, putting an emphasis on the problem of their construction.

In this section p denotes a prime number, and q denotes a power of p with a positive integer exponent.

Theorem 5.16 *Suppose that $\mathbb{F}$ is a finite field. Then there is a prime number p such that the prime field of $\mathbb{F}$ is isomorphic to $\mathbb{F}_p$ (the field of residue classes modulo p). Further, the field $\mathbb{F}$ is a finite dimensional vector space over $\mathbb{F}_p$, and the number of its elements is a power of p. In fact, if $\dim_{\mathbb{F}_p} \mathbb{F} = d$, then $|\mathbb{F}| = p^d$.*

Proof The characteristic of $\mathbb{F}$ must be a prime, say p, as a field with characteristic zero must have infinitely many elements. Thus the prime field P of $\mathbb{F}$ is isomorphic to $\mathbb{F}_p$. Since P is a subfield, the field $\mathbb{F}$ is a vector space over P. Let $\alpha_1, \ldots, \alpha_d$ be a

basis of $\mathbb{F}$ over P. Then each $\alpha \in \mathbb{F}$ can be written uniquely in the form $\sum_{j=1}^{d} a_i \alpha_i$ where $a_i \in P$. Hence $|\mathbb{F}| = p^d$. ■

In a field $\mathbb{F}$, the set of non-zero elements (the multiplicative group of $\mathbb{F}$) is denoted by $\mathbb{F}^*$. From Theorem 5.2 we immediately obtain the following result.

Theorem 5.17 *If $\mathbb{F}$ is a finite field, then its multiplicative group $\mathbb{F}^*$ is cyclic.*

A generator of the group $\mathbb{F}^*$ is said to be a ***primitive element***. If $|\mathbb{F}| = q$ and α is a primitive element of $\mathbb{F}$, then the elements of $\mathbb{F}$ are $0, \alpha, \alpha^2, \ldots, \alpha^{q-1} = 1$.

Corollary 5.18 *Suppose that $\mathbb{F}$ is a finite field with order p^d and let α be a primitive element of $\mathbb{F}$. Let $g \in \mathbb{F}_p[x]$ be a minimal polynomial of α over $\mathbb{F}_p$. Then g is irreducible in $\mathbb{F}_p[x]$, the degree of g is d, and $\mathbb{F}$ is isomorphic to the field $\mathbb{F}_p[x]/(g)$.*

Proof Since the element α is primitive in $\mathbb{F}$, we have $\mathbb{F} = \mathbb{F}_p(\alpha)$. The rest of the lemma follows from Lemma 5.8 and from Theorem 5.11. ■

Theorem 5.19 *Let $\mathbb{F}$ be a finite field with order q. Then*
(i) (Fermat's little theorem) *If $\beta \in \mathbb{F}^*$, then $\beta^{q-1} = 1$.*
(ii) If $\beta \in \mathbb{F}$, then $\beta^q = \beta$.

Proof (i) Suppose that $\alpha \in \mathbb{F}^*$ is a primitive element. Then we may choose an integer i such that $\beta = \alpha^i$. Therefore

$$\beta^{q-1} = (\alpha^i)^{q-1} = (\alpha^{q-1})^i = 1^i = 1.$$

(ii) Clearly, if $\beta = 0$ then this claim is true, while, for $\beta \neq 0$, the claim follows from part (i). ■

Theorem 5.20 *Let $\mathbb{F}$ be a field with q elements. Then*

$$x^q - x = \prod_{\alpha \in \mathbb{F}} (x - \alpha) \ .$$

Proof By Theorem 5.19 and Lemma 5.5, the product on the right-hand side is a divisor of the polynomial $x^q - x \in \mathbb{F}[x]$. Now the assertion follows, as the degrees and the leading coefficients of the two polynomials in the equation coincide. ■

Corollary 5.21 *Arbitrary two finite fields with the same number of elements are isomorphic.*

Proof Suppose that $q = p^d$, and that both $\mathbb{K}$ and $\mathbb{L}$ are fields with q elements. Let β be a primitive element in $\mathbb{L}$. Then Corollary 5.18 implies that a minimal polynomial $g(x) \in \mathbb{F}_p[x]$ of β over $\mathbb{F}_p$ is irreducible (in $\mathbb{F}_p[x]$) with degree d. Further, $\mathbb{L} \cong \mathbb{F}_p[x]/(g(x))$. By Lemma 5.8 and Theorem 5.19, the minimal polynomial g is a divisor of the polynomial $x^q - x$. Applying Theorem 5.20 to $\mathbb{K}$, we find that the polynomial $x^q - x$, and also its divisor $g(x)$, can be factored as a product of linear terms in $\mathbb{K}[x]$, and so $g(x)$ has at least one root α in $\mathbb{K}$. As $g(x)$ is irreducible in $\mathbb{F}_p[x]$, it must be a minimal polynomial of α (see Corollary 5.10), and so $\mathbb{F}_p(\alpha)$ is isomorphic to the field $\mathbb{F}_p[x]/(g(x))$. Comparing the number of elements in $\mathbb{F}_p(\alpha)$

and in $\mathbb{K}$, we find that $\mathbb{F}_p(\alpha) = \mathbb{K}$, and further, that $\mathbb{K}$ and $\mathbb{L}$ are isomorphic. ∎

In the sequel, we let $\mathbb{F}_q$ denote the field with q elements, provided it exists. In order to prove the existence of such a field for each prime-power q, the following two facts will be useful.

Lemma 5.22 *If p is a prime number and j is an integer such that $0 < j < p$, then $p \mid \binom{p}{j}$.*

Proof On the one hand, the number $\binom{p}{j}$ is an integer. On the other hand, $\binom{p}{j} = p(p-1)\cdots(p-j+1)/j!$ is a fraction such that, for $0 < j < p$, its numerator is divisible by p, but its denominator is not. ∎

Lemma 5.23 *Let R be a commutative ring and let p be a prime such that $pr = 0$ for all $r \in R$. Then the map $\Phi_p : R \to R$ mapping $r \mapsto r^p$ is a ring homomorphism.*

Proof Suppose that $r,\ s \in R$. Clearly,

$$\Phi_p(rs) = (rs)^p = r^p s^p = \Phi_p(r)\Phi_p(s) \ .$$

By the previous lemma,

$$\Phi_p(r+s) = (r+s)^p = \sum_{j=0}^{p} \binom{p}{j} r^{p-j} s^j = r^p + s^p = \Phi_p(r) + \Phi_p(s) \ .$$

We obtain in the same way that $\Phi_p(r-s) = \Phi_p(r) - \Phi_p(s)$. ∎

The homomorphism Φ_p in the previous lemma is called the ***Frobenius endomorphism***.

Theorem 5.24 *Assume that the polynomial $g(x) \in \mathbb{F}_q[x]$ is irreducible, and, for a positive integer d, it is a divisor of the polynomial $x^{q^d} - x$. Then the degree of $g(x)$ divides d.*

Proof Let n be the degree of $g(x)$, and suppose, by contradiction, that $d = tn + s$ where $0 < s < n$. The assumption that $g(x) \mid x^{q^d} - x$ can be rephrased as $x^{q^d} \equiv x \pmod{g(x)}$. However, this means that, for an arbitrary polynomial $u(x) = \sum_{i=0}^{N} u_i x^i \in \mathbb{F}_q[x]$, we have

$$u(x)^{q^d} = \sum_{i=0}^{N} u_i^{q^d} x^{iq^d} = \sum_{i=0}^{N} u_i (x^{q^d})^i \equiv \sum_{i=0}^{N} u_i x^i = u(x) \pmod{g(x)} \ .$$

Note that we applied Lemma 5.23 to the ring $R = \mathbb{F}_q[x]/(g(x))$, and Theorem 5.19 to $\mathbb{F}_q$. The residue class ring $\mathbb{F}_q[x]/(g(x))$ is isomorphic to the field $\mathbb{F}_{q^n}$, which has q^n elements. Let $u(x) \in \mathbb{F}_q[x]$ be a polynomial for which $u(x) \pmod{g(x)}$ is a primitive element in the field $\mathbb{F}_{q^n}$. That is, $u(x)^{q^n-1} \equiv 1 \pmod{g(x)}$, but $u(x)^j \not\equiv 1 \pmod{g(x)}$ for $j = 1, \ldots, q^n - 2$. Therefore,

$$u(x) \equiv u(x)^{q^d} = u(x)^{q^{tn+s}} = (u(x)^{q^{nt}})^{q^s} \equiv u(x)^{q^s} \pmod{g(x)} \ ,$$

and so $u(x)(u(x)^{q^s-1} - 1) \equiv 0 \pmod{g(x)}$. Since the residue class ring $\mathbb{F}_q[x]/(g(x))$ is a field, $u(x) \not\equiv 0 \pmod{g(x)}$, but we must have $u(x)^{q^s-1} \equiv 1 \pmod{g(x)}$. As $0 \le q^s - 1 < q^n - 1$, this contradicts to the primitivity of $u(x) \pmod{g(x)}$. ∎

Theorem 5.25 *For an arbitrary prime p and positive integer d, there exists a field with p^d elements.*

Proof We use induction on d. The claim clearly holds if $d = 1$. Now let $d > 1$ and let r be a prime divisor of d. By the induction hypothesis, there is a field with $q = p^{(d/r)}$ elements. By Theorem 5.24, each of the irreducible factors, in $\mathbb{F}_q[x]$, of the the polynomial $f(x) = x^{q^r} - x$ has degree either 1 or r. Further, $f'(x) = (x^{q^r} - x)' = -1$, and so, by Lemma 5.13, $f(x)$ is square-free. Over $\mathbb{F}_q$, the number of linear factors of $f(x)$ is at most q, and so is the degree of their product. Hence there exist at least $(q^r - q)/r \geq 1$ polynomials with degree r that are irreducible in $\mathbb{F}_q[x]$. Let $g(x)$ be such a polynomial. Then the field $\mathbb{F}_q[x]/(g(x))$ is isomorphic to the field with $q^r = p^d$ elements. ■

Corollary 5.26 *For each positive integer d, there is an irreducible polynomial $f \in \mathbb{F}_p[x]$ with degree d.*

Proof Take a minimal polynomial over $\mathbb{F}_p$ of a primitive element in $\mathbb{F}_{p^d}$. ■

A little bit later, in Theorem 5.31, we will prove a stronger statement: a random polynomial in $\mathbb{F}_p[x]$ with degree d is irreducible with high probability.

Subfields of finite fields The following theorem describes all subfields of a finite field.

Theorem 5.27 *The field $\mathbb{F} = \mathbb{F}_{p^n}$ contains a subfield isomorphic to $\mathbb{F}_{p^k}$, if and only if $k \mid n$. In this case, there is exactly one subfield in $\mathbb{F}$ that is isomorphic to $\mathbb{F}_{p^k}$.*

Proof The condition that $k \mid n$ is necessary, since the larger field is a vector space over the smaller field, and so $p^n = (p^k)^l$ must hold with a suitable integer l.

Conversely, suppose that $k \mid n$, and let $f \in \mathbb{F}_p[x]$ be an irreducible polynomial with degree k. Such a polynomial exists by Corollary 5.26. Let $q = p^k$. Applying Theorem 5.19, we obtain, in $\mathbb{F}_p[x]/(f)$, that $x^q \equiv x \pmod{f}$, which yields $x^{p^n} = x^{q^l} \equiv x \pmod{f}$. Thus f must be a divisor of the polynomial $x^{p^n} - x$. Using Theorem 5.20, we find that f has a root α in $\mathbb{F}$. Now we may prove in the usual way that the subfield $\mathbb{F}_p(\alpha)$ is isomorphic to $\mathbb{F}_{p^k}$.

The last assertion is valid, as the elements of $\mathbb{F}_q$ are exactly the roots of $x^q - x$ (Theorem 5.20), and this polynomial can have, in an arbitrary field, at most q roots. ■

The structure of irreducible polynomials Next we prove an important property of the irreducible polynomials over finite fields.

Theorem 5.28 *Assume that $\mathbb{F}_q \subseteq \mathbb{F}$ are finite fields, and let $\alpha \in \mathbb{F}$. Let $f \in \mathbb{F}_q[x]$ be the minimal polynomial of α over $\mathbb{F}_q$ with leading coefficient 1, and suppose that* $\deg f = d$. *Then*

$$f(x) = (x - \alpha)(x - \alpha^q) \cdots (x - \alpha^{q^{d-1}}) .$$

Moreover, the elements $\alpha, \alpha^q, \ldots, \alpha^{q^{d-1}}$ are pairwise distinct.

Proof Let $f(x) = a_0 + a_1 x + \cdots + x^d$. If $\beta \in \mathbb{F}$ with $f(\beta) = 0$, then, using Lemma 5.23 and Theorem 5.19, we obtain

$$0 = f(\beta)^q = (a_0 + a_1\beta + \cdots + \beta^d)^q = a_0^q + a_1^q \beta^q + \cdots + \beta^{dq} = a_0 + a_1\beta^q + \cdots + \beta^{qd} = f(\beta^q) \ .$$

Thus β^q is also a root of f.

As α is a root of f, the argument in the previous paragraph shows that so are the elements $\alpha, \alpha^q, \ldots, \alpha^{q^{d-1}}$. Hence it suffices to show, that they are pairwise distinct. Suppose, by contradiction, that $\alpha^{q^i} = \alpha^{q^j}$ and that $0 \le i < j < d$. Let $\beta = \alpha^{q^i}$ and let $l = j - i$. By assumption, $\beta = \beta^{q^l}$, which, by Lemma 5.8, means that $f(x) \mid x^{q^l} - x$. From Theorem 5.24, we obtain, in this case, that $d \mid l$, which is a contradiction, as $l < d$. ∎

This theorem shows that a polynomial f which is irreducible over a finite field cannot have multiple roots. Further, all the roots of f can be obtained from a single root taking q-th powers repeatedly.

Automorphisms In this section we characterise certain automorphisms of finite fields.

Definition 5.29 *Suppose that $\mathbb{F}_q \subseteq \mathbb{F}$ are finite fields. The map $\Psi : \mathbb{F} \to \mathbb{F}$ is an $\mathbb{F}_q$-**automorphism** of the field $\mathbb{F}$, if it is an isomorphism between rings, and $\Psi(a) = a$ holds for all $a \in \mathbb{F}_q$.*

Recall that the map $\Phi = \Phi_q : \mathbb{F} \to \mathbb{F}$ is defined as follows: $\Phi(\alpha) = \alpha^q$ where $\alpha \in \mathbb{F}$.

Theorem 5.30 *The set of $\mathbb{F}_q$-automorphisms of the field $\mathbb{F} = \mathbb{F}_{q^d}$ is formed by the maps $\Phi, \Phi^2, \ldots, \Phi^d = id$.*

Proof By Lemma 5.23, the map $\Phi : \mathbb{F} \to \mathbb{F}$ is a ring homomorphism. The map Φ is obviously one-to-one, and hence it is also an isomorphism. It follows from Theorem 5.19, that Φ leaves the elements $\mathbb{F}_q$ fixed. Thus the maps Φ^j are $\mathbb{F}_q$-automorphisms of $\mathbb{F}$.

Suppose that $f(x) = a_0 + a_1 x + \cdots + x^d \in \mathbb{F}_q[x]$, and $\beta \in \mathbb{F}$ with $f(\beta) = 0$, and that Ψ is an $\mathbb{F}_q$-automorphism of $\mathbb{F}$. We claim that $\Psi(\beta)$ is a root of f. Indeed,

$$0 = \Psi(f(\beta)) = \Psi(a_0) + \Psi(a_1)\Psi(\beta) + \cdots + \Psi(\beta)^d = f(\Psi(\beta)) \ .$$

Let β be a primitive element of $\mathbb{F}$ and assume now that $f \in \mathbb{F}_q[x]$ is a minimal polynomial of β. By the observation above and by Theorem 5.28, $\Psi(\beta) = \beta^{q^j}$, with some $0 \le j < d$, that is, $\Psi(\beta) = \Phi^j(\beta)$. Hence the images of a generating element of $\mathbb{F}$ under the automorphisms Ψ and Φ^j coincide, which gives $\Psi = \Phi^j$. ∎

The construction of finite fields Let $q = p^n$. By Theorem 5.7 and Corollary 5.26, the field $\mathbb{F}_q$ can be written in the form $\mathbb{F}[x]/(f)$, where $f \in \mathbb{F}[x]$ is an irreducible polynomial with degree n. In practical applications of field theory, for example in computer science, this is the most common method of constructing a finite field. Using, for instance, the polynomial $f(x) = x^3 + x + 1$ in Example 5.2, we may construct the field $\mathbb{F}_8$. The following theorem shows that we have a good chance of obtaining an irreducible polynomial by a random selection.

Theorem 5.31 *Let $f(x) \in \mathbb{F}_q[x]$ be a uniformly distributed random polynomial with degree $k > 1$ and leading coefficient 1. (Being uniformly distributed means that the probability of choosing f is $1/q^k$.) Then f is irreducible over $\mathbb{F}_q$ with probability at least $1/k - 1/q^{k/2}$.*

Proof First we estimate the number of elements $\alpha \in \mathbb{F}_{q^k}$ for which $\mathbb{F}_q(\alpha) = \mathbb{F}_{q^k}$. We claim that the number of such elements is at least

$$|\mathbb{F}_{q^k}| - \sum_{r|k} |\mathbb{F}_{q^{k/r}}| \,,$$

where the summation runs for the distinct prime divisors r of k. Indeed, if α does not generate, over $\mathbb{F}_q$, the field $\mathbb{F}_{q^k}$, then it is contained in a maximal subfield of $\mathbb{F}_{q^k}$, and these maximal subfields are, by Theorem 5.27, exactly the fields of the form $\mathbb{F}_{q^{k/r}}$. The number of distinct prime divisors of k are at most $\lg k$, and so the number of such elements α is at least $q^k - (\lg k)q^{k/2}$. The minimal polynomials with leading coefficients 1 over $\mathbb{F}_q$ of such elements α have degree k and they are irreducible. Such a polynomial is a minimal polynomial of exactly k elements α (Theorem 5.28). Hence the number of distinct irreducible polynomials with degree k and leading coefficient 1 in $\mathbb{F}_q[x]$ is at least

$$\frac{q^k}{k} - \frac{(\lg k)q^{k/2}}{k} \geq \frac{q^k}{k} - q^{k/2} \,,$$

from which the claim follows. ■

If, having $\mathbb{F}_q$, we would like to construct one of its extensions $\mathbb{F}_{q^k}$, then it is worth selecting a ***random*** polynomial

$$f(x) = a_0 + a_1 x + \cdots + a_{k-1}x^{k-1} + x^k \in \mathbb{F}_q[x].$$

In other words, we select uniformly distributed random coefficients $a_0, \ldots, a_{k-1} \in \mathbb{F}_q$ independently. The polynomial so obtained is irreducible with a high probability (in fact, with probability at least $1/k - \epsilon$ if q^k is large). Further, in this case, we also have $\mathbb{F}_q[x]/(f) \cong \mathbb{F}_{q^k}$. We expect that we will have to select about k polynomials before we find an irreducible one.

We have seen in Theorem 5.2 that field extensions can be obtained using irreducible polynomials. It is often useful if these polynomials have some further nice properties. The following lemma claims the existence of such polynomials.

Lemma 5.32 *Let r be a prime. In a finite field $\mathbb{F}_q$ there exists an element which is not an r-th power if and only if $q \equiv 1 \pmod{r}$. If $b \in \mathbb{F}_q$ is such an element, then the polynomial $x^r - b$ is irreducible in $\mathbb{F}_q[x]$, and so $\mathbb{F}_q[x]/(x^r - b)$ is a field with q^r elements.*

Proof Suppose first that $r \nmid q - 1$ and let s be a positive integer such that $sr \equiv 1 \pmod{q-1}$. If $b \in \mathbb{F}_q$ such that $b \neq 0$, then $(b^s)^r = b^{sr} = bb^{sr-1} = b$, while if $b = 0$, then $b = 0^r$. Hence, in this case, each of the elements of $\mathbb{F}_q$ is an r-th power.

Next we assume that $r \mid q - 1$, and we let a be a primitive element in $\mathbb{F}_q$. Then, in $\mathbb{F}_q$, the r-th powers are exactly the following $1 + (q-1)/r$ elements: $0, (a^r)^0, (a^r)^1, \ldots, (a^r)^{(q-1)/r-1}$. Suppose now that $r^s \mid q - 1$, but $r^{s+1} \nmid q - 1$.

Then the order of an element $b \in \mathbb{F}_q \setminus \{0\}$ is divisible by r^s if and only if b is not an r-th power. Let b be such an element, and let $g(x) \in \mathbb{F}_q[x]$ be an irreducible factor of the polynomial $x^r - b$. Suppose that the degree of $g(x)$ is d; clearly, $d \le r$. Then $\mathbb{K} = \mathbb{F}_q[x]/(g(x))$ is a field with q^d elements and, in $\mathbb{K}$, the equation $[x]^r = b$ holds. Therefore the order of $[x]$ is divisible by r^{s+1}. Consequently, $r^{s+1} \mid q^d - 1$. As $q-1$ is not divisible by r^{s+1}, we have $r \mid (q^d-1)/(q-1) = 1 + q + \cdots + q^{d-1}$. In other words $1+q+\ldots+q^{d-1} \equiv 0 \pmod{r}$. On the other hand, as $q \equiv 1 \pmod{r}$, we find $1 + q + \cdots + q^{d-1} \equiv d \pmod{r}$, and hence $d \equiv 0 \pmod{r}$, which, since $0 < d \le r$, can only happen if $d = r$. ∎

In certain cases, we can use the previous lemma to boost the probability of finding an irreducible polynomial.

Proposition 5.33 *Let r be a prime such that $r \mid q-1$. Then, for a random element $b \in \mathbb{F}_q^*$, the polynomial $x^r - b$ is irreducible in $\mathbb{F}_q[x]$ with probability at least $1 - 1/r$.*

Proof Under the conditions, the r-th powers in $\mathbb{F}_q^*$ constitute the cyclic subgroup with order $(q-1)/r$. Thus a random element $b \in \mathbb{F}_q^*$ is an r-th power with probability $1/r$, and hence the assertion follows from Lemma 5.32. ∎

Remark. Assume that $r \mid (q-1)$, and, if $r = 2$, then assume also that $4 \mid (q-1)$. In this case there is an element b in $\mathbb{F}_q$ that is not an r-th power. We claim that that the residue class $[x]$ is not an r-th power in $\mathbb{F}_q[x]/(x^r - b) \cong \mathbb{F}_q^r$. Indeed, by the argument in the proof of Lemma 5.32, it suffices to show that $r^2 \nmid (q^r-1)/(q-1)$. By our assumptions, this is clear if $r = 2$. Now assume that $r > 2$, and write $q \equiv 1 + rt \pmod{r^2}$. Then, for all integers $i \ge 0$, we have $q^i \equiv 1 + irt \pmod{r^2}$, and so, by the assumptions,

$$\frac{q^r - 1}{q-1} = 1 + q + \cdots + q^{r-1} \equiv r + \frac{r(r-1)}{2} rt \equiv r \pmod{r^2} .$$

Exercises

5.2-1 Show that the polynomial $x^{q+1} - 1$ can be factored as a product of linear factors over the field $\mathbb{F}_{q^2}$.

5.2-2 Show that the polynomial $f(x) = x^4 + x + 1$ is irreducible over $\mathbb{F}_2$, that is, $\mathbb{F}_2[x]/(f) \cong \mathbb{F}_{16}$. What is the order of the element $[x]_f$ in the residue class ring? Is it true that the element $[x]_f$ is primitive in $\mathbb{F}_{16}$?

5.2-3 Determine the irreducible factors of $x^{31} - 1$ over the field $\mathbb{F}_2$.

5.2-4 Determine the subfields of $\mathbb{F}_{3^6}$.

5.2-5 Let a and b be positive integers. Show that there exists a finite field $\mathbb{K}$ containing $\mathbb{F}_q$ such that $\mathbb{F}_{q^a} \subseteq \mathbb{K}$ and $\mathbb{F}_{q^b} \subseteq \mathbb{K}$. What can we say about the number of elements in $\mathbb{K}$?

5.2-6 Show that the number of irreducible polynomials with degree k and leading coefficient 1 over $\mathbb{F}_q$ is at most q^k/k.

5.2-7 *(a)* Let $\mathbb{F}$ be a field, let V be an n-dimensional vector space over $\mathbb{F}$, and let $A : V \to V$ be a linear transformation whose minimal polynomial coincides with its characteristic polynomial. Show that there exists a vector $v \in V$ such that the images $v, Av, \ldots, A^{n-1}v$ are linearly independent.

(b) A set $S = \{\alpha, \alpha^q, \ldots, \alpha^{q^{d-1}}\}$ is said to be a ***normal basis*** of $\mathbb{F}_{q^d}$ over $\mathbb{F}_q$, if $\alpha \in \mathbb{F}_{q^d}$ and S is a linearly independent set over $\mathbb{F}_q$. Show that $\mathbb{F}_{q^d}$ has a normal basis over $\mathbb{F}_q$. *Hint.* Show that a minimal polynomial of the $\mathbb{F}_q$-linear map $\Phi : \mathbb{F}_{q^d} \to \mathbb{F}_{q^d}$ is $x^d - 1$, and use part (a).

5.3. Factoring polynomials over finite fields

One of the problems that we often have to solve when performing symbolic computation is the ***factorisation*** problem. Factoring an algebraic expression means writing it as a product of simpler expressions. Experience shows that this can be very helpful in the solution of a large variety of algebraic problems. In this section, we consider a class of factorisation algorithms that can be used to factor polynomials in one variable over finite fields.

The input of the ***polynomial factorisation problem*** is a polynomial $f(x) \in \mathbb{F}_q[x]$. Our aim is to compute a factorisation

$$f = f_1^{e_1} f_2^{e_2} \cdots f_s^{e_s} \tag{5.3}$$

of f where the polynomials $f_1, \ldots, f_s$ are pairwise relatively prime and irreducible over $\mathbb{F}_q$, and the exponents e_i are positive integers. By Theorem 5.4, f determines the polynomials f_i and the exponents e_i essentially uniquely.

Example 5.3 Let $p = 23$ and let

$$f(x) = x^6 - 3x^5 + 8x^4 - 11x^3 + 8x^2 - 3x + 1 \, .$$

Then it is easy to compute modulo 23 that

$$f(x) = (x^2 - x + 10)(x^2 + 5x + 1)(x^2 - 7x + 7) \, .$$

None of the factors $x^2 - x + 10$, $x^2 + 5x + 1$, $x^2 - 7x + 7$ has a root in $\mathbb{F}_{23}$, and so they are necessarily irreducible in $\mathbb{F}_{23}[x]$.

The factorisation algorithms are important computational tools, and so they are implemented in most of the computer algebra systems (Mathematica, Maple, etc). These algorithms are often used in the area of error-correcting codes and in cryptography.

Our aim in this section is to present some of the basic ideas and building blocks that can be used to factor polynomials over finite fields. We will place an emphasis on the existence of polynomial time algorithms. The discussion of the currently best known methods is, however, outside the scope of this book.

5.3.1. Square-free factorisation

The factorisation problem in the previous section can efficiently be reduced to the special case when the polynomial f to be factored is square-free; that is, in (5.3), $e_i = 1$ for all i. The basis of this reduction is Lemma 5.13 and the following simple result. Recall that the derivative of a polynomial $f(x)$ is denoted by $f'(x)$.

Lemma 5.34 *Let $f(x) \in \mathbb{F}_q[x]$ be a polynomial. If $f'(x) = 0$, then there exists a polynomial $g(x) \in \mathbb{F}_q[x]$ such that $f(x) = g(x)^p$.*

Proof Suppose that $f(x) = \sum_{i=0}^{n} a_i x^i$. Then $f'(x) = \sum_{i=1}^{n} a_i i x^{i-1}$. If the coefficient $a_i i$ is zero in $\mathbb{F}_q$ then either $a_i = 0$ or $p \mid i$. Hence, if $f'(x) = 0$ then $f(x)$ can be written as $f(x) = \sum_{j=0}^{k} b_j x^{pj}$. Let $q = p^d$; then choosing $c_j = b_j^{p^{d-1}}$, we have $c_j^p = b_j^{p^d} = b_j$, and so $f(x) = (\sum_{j=0}^{k} c_j x^j)^p$. ∎

If $f'(x) = 0$, then, using the previous lemma, a factorisation of $f(x)$ into square-free factors can be obtained from that of the polynomial $g(x)$, which has smaller degree. On the other hand, if $f'(x) \neq 0$, then, by Lemma 5.13, the polynomial $f(x)/\gcd(f(x), f'(x))$ is already square-free and we only have to factor $\gcd(f(x), f'(x))$ into square-free factors. The division of polynomials and computing the greatest common divisor can be performed in polynomial time, by Theorem 5.12. In order to compute the polynomial $g(x)$, we need the solutions, in $\mathbb{F}_q$, of equations of the form $y^p = a$ with $a \in \mathbb{F}_q$. If $q = p^s$, then $y = a^{p^{s-1}}$ is a solution of such an equation, which, using ***fast exponentiation*** (repeated squaring, see 33.6.1), can be obtained in polynomial time.

One of the two reduction steps can always be performed if f is divisible by a square of a polynomial with positive degree.

Usually a polynomial can be written as a product of square-free factors in many different ways. For the sake of uniqueness, we define the ***square-free factorisation*** of a polynomial $f \in \mathbb{F}[x]$ as the factorisation

$$f = f_1^{e_1} \cdots f_s^{e_s} ,$$

where $e_1 < \cdots < e_s$ are integers, and the polynomials f_i are relatively prime and square-free. Hence we collect together the irreducible factors of f with the same multiplicity. The following algorithm computes a square-free factorisation of f. Besides the observations we made in this section, we also use Lemma 5.14. This lemma, combined with Lemma 5.13, guarantees that the product of the irreducible factors with multiplicity one of a polynomial f over a finite field is $f/\gcd(f, f')$.

SQUARE-FREE-FACTORISATION(f)

1 $g \leftarrow f$
2 $S \leftarrow \emptyset$
3 $m \leftarrow 1$
4 $i \leftarrow 1$

```
5  while deg g ≠ 0
6        do if g' = 0
7              then g ← ∛g
8                   i ← i · p
9              else h ← g/gcd(g, g')
10                  g ← g/h
11                  if deg h ≠ 0
12                     then S ← S ∪ (h, m)
13                  m ← m + i
14 return S
```

The degree of the polynomial g decreases after each execution of the main loop, and the subroutines used in this algorithm run in polynomial time. Thus the method above can be performed in polynomial time.

5.3.2. Distinct degree factorisation

Suppose that f is a square-free polynomial. Now we factor f as

$$f(x) = h_1(x)h_2(x)\cdots h_t(x)\ , \tag{5.4}$$

where, for $i = 1, \ldots, t$, the polynomial $h_i(x) \in \mathbb{F}_q[x]$ is a product of irreducible polynomials with degree i. Though this step is not actually necessary for the solution of the factorisation problem, it is worth considering, as several of the known methods can efficiently exploit the structure of the polynomials h_i. The following fact serves as the starting point of the distinct degree factorisation.

Theorem 5.35 *The polynomial $x^{q^d} - x$ is the product of all the irreducible polynomials $f \in \mathbb{F}_q[x]$, each of which is taken with multiplicity 1, that have leading coefficient 1 and whose degree divides d.*

Proof As $(x^{q^d} - x)' = -1$, all the irreducible factors of this polynomial occur with multiplicity one. If $f \in \mathbb{F}_q[x]$ is irreducible and divides $x^{q^d} - x$, then, by Theorem 5.24, the degree of f divides d.

Conversely, let $f \in \mathbb{F}_q[x]$ be an irreducible polynomial with degree k such that $k \mid d$. Then, by Theorem 5.27, f has a root in $\mathbb{F}_{q^d}$, which implies $f \mid x^{q^d} - x$. ∎

The theorem offers an efficient method for computing the polynomials $h_i(x)$. First we separate h_1 from f, and then, step by step, we separate the product of the factors with higher degrees.

Distinct-Degree-Factorisation(f)

```
1  F ← f
3  for i ← 1 to deg f
4       do h_i ← gcd(F, x^{q^i} − x)
7          F ← F/h_i
8  return h_1, ..., h_{deg f}
```

If, in this algorithm, the polynomial $F(x)$ is constant, then we may stop, as the further steps will not give new factors. As the polynomial $x^{q^i} - x$ may have large degree, computing $\gcd(F(x), x^{q^i} - x)$ must be performed with particular care. The important idea here is that the residue $x^{q^i} \pmod{F(x)}$ can be computed using fast exponentiation.

The algorithm outlined above is suitable for testing whether a polynomial is irreducible, which is one of the important problems that we encounter when constructing finite fields. The algorithm presented here for distinct degree factorisation can solve this problem efficiently. For, it is obvious that a polynomial f with degree k is irreducible, if, in the factorisation (5.4), we have $h_k(x) = f(x)$.

The following algorithm for testing whether a polynomial is irreducible is somewhat more efficient than the one sketched in the previous paragraph and handles correctly also the inputs that are not square-free.

IRREDUCIBILITY-TEST(f)

```
1  n ← deg f
2  if x^(p^n) ≢ x (mod f)
3     then return "no"
4  for the prime divisors r of n
5        do if x^(p^(n/r)) ≡ x (mod f)
6              then return "no"
7  return "yes"
```

In lines 2 and 5, we check whether n is the smallest among the positive integers k for which f divides $x^{q^k} - x$. By Theorem 5.35, this is equivalent to the irreducibility of f. If f survives the test in line 2, then, by Theorem 5.35, we know that f is square-free and k must divide n. Using at most $\lg n + 1$ fast exponentiations modulo f, we can thus decide if f is irreducible.

Theorem 5.36 *If the field $\mathbb{F}_q$ is given and $k > 1$ is an integer, then the field $\mathbb{F}_{q^k}$ can be constructed using a randomised Las Vegas algorithm which runs in time polynomial in $\lg q$ and k.*

Proof The algorithm is the following.

FINITE-FIELD-CONSTRUCTION(q^k)

```
1  for i ← 0 to k − 1
2        do a_i ← a random element (uniformly distributed) of F_q
3  f ← x^k + Σ_{i=0}^{k−1} a_i x^i
4  if IRREDUCIBILITY-TEST(f) = "yes"
5     then return F_q[x]/(f)
6     else  return "fail"
```

In lines 1–3, we choose a uniformly distributed random polynomial with leading coefficient 1 and degree k. Then, in line 4, we efficiently check if $f(x)$ is irreducible. By Theorem 5.31, the polynomial f is irreducible with a reasonably high probability.

■

5.3.3. The Cantor-Zassenhaus algorithm

In this section we consider the special case of the factorisation problem in which q is odd and the polynomial $f(x) \in \mathbb{F}_q[x]$ is of the form

$$f = f_1 f_2 \cdots f_s, \tag{5.5}$$

where the f_i are pairwise relatively prime irreducible polynomials in $\mathbb{F}_q[x]$ with the same degree d, and we also assume that $s \geq 2$. Our motivation for investigating this special case is that a square-free distinct degree factorisation reduces the general factorisation problem to such a simpler problem. If q is even, then Berlekamp's method, presented in Section 5.3.4, gives a deterministic polynomial time solution. There is a variation of the method discussed in the present section that works also for even q; see Exercise *5-2*

Lemma 5.37 *Suppose that q is odd. Then there are $(q^2-1)/2$ pairs $(c_1, c_2) \in \mathbb{F}_q \times \mathbb{F}_q$ such that exactly one of $c_1^{(q-1)/2}$ and $c_2^{(q-1)/2}$ is equal to 1.*

Proof Suppose that a is a primitive element in $\mathbb{F}_q$; that is, $a^{q-1} = 1$, but $a^k \neq 1$ for $0 < k < q-1$. Then $\mathbb{F}_q \setminus \{0\} = \{a^s | s = 0, \ldots, q-2\}$, and further, as $\left(a^{(q-1)/2}\right)^2 = 1$, but $a^{(q-1)/2} \neq 1$, we obtain that $a^{(q-1)/2} = -1$. Therefore $a^{s(q-1)/2} = (-1)^s$, and so half of the element $c \in \mathbb{F}_q \setminus \{0\}$ give $c^{(q-1)/2} = 1$, while the other half give $c^{(q-1)/2} = -1$. If $c = 0$ then clearly $c^{(q-1)/2} = 0$. Thus there are $((q-1)/2)((q+1)/2)$ pairs (c_1, c_2) such that $c_1^{(q-1)/2} = 1$, but $c_2^{(q-1)/2} \neq 1$, and, obviously, we have the same number of pairs for which the converse is valid. Thus the number of pairs that satisfy the condition is $(q-1)(q+1)/2 = (q^2-1)/2$. ■

Theorem 5.38 *Suppose that q is odd and the polynomial $f(x) \in \mathbb{F}_q[x]$ is of the form (5.5) and has degree n. Choose a uniformly distributed random polynomial $u(x) \in \mathbb{F}_q[x]$ with degree less than n. (That is, choose pairwise independent, uniformly distributed scalars $u_0, \ldots, u_{n-1}$, and consider the polynomial $u(x) = \sum_{i=0}^{n-1} u_i x^i$.) Then, with probability at least $(q^{2d}-1)/(2q^{2d}) \geq 4/9$, the greatest common divisor*

$$\gcd(u(x)^{\frac{q^d-1}{2}} - 1, f(x))$$

is a proper divisor of $f(x)$.

Proof The element $u(x) \pmod{f_i(x)}$ corresponds to an element of the residue class field $\mathbb{F}[x]/(f_i(x)) \cong \mathbb{F}_{q^d}$. By the Chinese remainder theorem (Theorem 5.15), choosing the polynomial $u(x)$ uniformly implies that the residues of $u(x)$ modulo the factors $f_i(x)$ are independent and uniformly distributed random polynomials. By Lemma 5.37, the probability that exactly one of the residues of the polynomial $u(x)^{(q^d-1)/2} - 1$ modulo $f_1(x)$ and $f_2(x)$ is zero is precisely $(q^{2d}-1)/(2q^{2d})$. In this case the greatest common divisor in the theorem is indeed a divisor of f. For, if $u(x)^{(q^d-1)/2} - 1 \equiv 0 \pmod{f_1(x)}$, but this congruence is not valid modulo $f_2(x)$, then the polynomial $u(x)^{(q^d-1)/2} - 1$ is divisible by the factor $f_1(x)$, but not divisible

by $f_2(x)$, and so its greatest common divisor with $f(x)$ is a proper divisor of $f(x)$. The function

$$\frac{q^{2d}-1}{2q^{2d}} = \frac{1}{2} - \frac{1}{2q^{2d}}$$

is strictly increasing in q^d, and it takes its smallest possible value if q^d is the smallest odd prime-power, namely 3. The minimum is, thus, $1/2 - 1/18 = 4/9$. ∎

The previous theorem suggests the following randomised Las Vegas polynomial time algorithm for factoring a polynomial of the form (5.5) to a product of two factors.

CANTOR-ZASSENHAUS-ODD(f, d)

```
1  n ← deg f
2  for i ← 0 to n − 1
3       do u_i ← a random element (uniformly distributed) of F_q
4  u ← Σ_{i=0}^{n−1} u_i x^i
5  g ← gcd(u^{(q^d−1)/2} − 1, f)
6  if 0 < deg g < deg f
7     then return(g, f/g)
8     else  return "fail"
```

If one of the polynomials in the output is not irreducible, then, as it is of the form (5.5), it can be fed, as input, back into the algorithm. This way we obtain a polynomial time randomised algorithm for factoring f.

In the computation of the greatest common divisor, the residue $u(x)^{(q^d-1)/2} \pmod{f(x)}$ should be computed using fast exponentiation.

Now we can conclude that the general factorisation problem (5.3) over a field with odd order can be solved using a randomised polynomial time algorithm.

5.3.4. Berlekamp's algorithm

Here we will describe an algorithm that reduces the problem of factoring polynomials to the problem of searching through the underlying field or its prime field. We assume that

$$f(x) = f_1^{e_1}(x) \cdots f_s^{e_s}(x) \ ,$$

where the $f_i(x)$ are pairwise non-associate, irreducible polynomials in $\mathbb{F}_q[x]$, and also that $\deg f(x) = n$. The Chinese remainder theorem (Theorem 5.15) gives an isomorphism between the rings $\mathbb{F}_q[x]/(f)$ and

$$\mathbb{F}_q[x]/(f_1^{e_1}) \oplus \cdots \oplus \mathbb{F}_q[x]/(f_s^{e_s}).$$

The isomorphism is given by the following map:

$$[u(x)]_f \leftrightarrow ([u(x)]_{f_1^{e_1}}, \ldots, [u(x)]_{f_s^{e_s}}) \ ,$$

where $u(x) \in \mathbb{F}_q[x]$.

The most important technical tools in Berlekamp's algorithm are the p-th and

q-th power maps in the residue class ring $\mathbb{F}_q[x]/(f(x))$. Taking p-th and q-th powers on both sides of the isomorphism above given by the Chinese remainder theorem, we obtain the following maps:

$$[u(x)]^p \quad \leftrightarrow \quad ([u(x)^p]_{f_1^{e_1}}, \ldots, [u(x)^p]_{f_s^{e_s}}) \ , \tag{5.6}$$

$$[u(x)]^q \quad \leftrightarrow \quad ([u(x)^q]_{f_1^{e_1}}, \ldots, [u(x)^q]_{f_s^{e_s}}) \ . \tag{5.7}$$

The ***Berlekamp subalgebra*** B_f of the polynomial $f = f(x)$ is the subring of the residue class ring $\mathbb{F}_q[x]/(f)$ consisting of the fixed points of the q-th power map. Further, the ***absolute Berlekamp subalgebra*** A_f of f consists of the fixed points of the p-th power map. In symbols,

$$B_f = \{[u(x)]_f \in \mathbb{F}_q[x]/(f) : \ [u(x)^q]_f = [u(x)]_f\} \ ,$$

$$A_f = \{[u(x)]_f \in \mathbb{F}_q[x]/(f) : \ [u(x)^p]_f = [u(x)]_f\} \ .$$

It is easy to see that $A_f \subseteq B_f$. The term subalgebra is used here, because both types of Berlekamp subalgebras are subrings in the residue class ring $\mathbb{F}_q[x]/(f(x))$ (that is they are closed under addition and multiplication modulo $f(x)$), and, in addition, B_f is also linear subspace over $\mathbb{F}_q$, that is, it is closed under multiplication by the elements of $\mathbb{F}_q$. The absolute Berlekamp subalgebra A_f is only closed under multiplication by the elements of the prime field $\mathbb{F}_p$.

The Berlekamp subalgebra B_f is a subspace, as the map $u \mapsto u^q - u \pmod{f(x)}$ is an $\mathbb{F}_q$-linear map of $\mathbb{F}_q[x]/g(x)$ into itself, by Lemma 5.23 and Theorem 5.19. Hence a basis of B_f can be computed as a solution of a homogeneous system of linear equations over $\mathbb{F}_q$, as follows.

For all $i \in \{0, \ldots, n-1\}$, compute the polynomial $h_i(x)$ with degree at most $n-1$ that satisfies $x^{iq} - x^i \equiv h_i(x) \pmod{f(x)}$. For each i, such a polynomial h_i can be determined by fast exponentiation using $O(\lg q)$ multiplications of polynomials and divisions with remainder. Set $h_i(x) = \sum_{j=0}^{n} h_{ij} x^j$. The class $[u]_f$ of a polynomial $u(x) = \sum_{i=0}^{n-1} u_i x^i$ with degree less than n lies in the Berlekamp subalgebra if and only if

$$\sum_{i=0}^{n-1} u_i h_i(x) = 0 \ ,$$

which, considering the coefficient of x^j for $j = 0, \ldots, n-1$, leads to the following system of n homogeneous linear equations in n variables:

$$\sum_{i=0}^{n-1} h_{ij} u_i = 0, \quad (j = 0, \ldots, n-1) \ .$$

Similarly, computing a basis of the absolute Berlekamp subalgebra over $\mathbb{F}_p$ can be carried out by solving a system of nd homogeneous linear equations in nd variables over the prime field $\mathbb{F}_p$, as follows. We represent the elements of $\mathbb{F}_q$ in the usual way, namely using polynomials with degree less than d in $\mathbb{F}_p[y]$. We perform the

operations modulo $g(y)$, where $g(y) \in \mathbb{F}_p[y]$ is an irreducible polynomial with degree d over the prime field $\mathbb{F}_p$. Then the polynomial $u[x] \in \mathbb{F}_q[x]$ of degree less than n can be written in the form

$$\sum_{i=0}^{n-1} \sum_{j=0}^{d-1} u_{ij} y^j x^i,$$

where $u_{ij} \in \mathbb{F}_p$. Let, for all $i \in \{0, \ldots, n-1\}$ and for all $j \in \{0, \ldots, d-1\}$, $h_{ij}(x) \in \mathbb{F}_q[x]$ be the unique polynomial with degree at most $(n-1)$ for which $h_{ij}(x) \equiv (y^j x^i)^p - y^j x^i \pmod{f(x)}$. The polynomial $h_{ij}(x)$ is of the form $\sum_{k=0}^{n-1} \sum_{l=0}^{d-1} h_{ij}^{kl} y^l x^k$. The criterion for being a member of the absolute Berlekamp subalgebra of $[u]$ with $u[x] = \sum_{i=0}^{n-1} \sum_{j=0}^{d-1} u_{ij} y^j x^i$ is

$$\sum_{i=0}^{n-1} \sum_{j=0}^{d-1} u_{ij} h_{ij}(x) = 0 \ ,$$

which, considering the coefficients of the monomials $y^l x^k$, is equivalent to the following system of equations:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{d-1} h_{ij}^{kl} u_{ij} = 0 \ \ (k = 0, \ldots, n-1, \, l = 0, \ldots, d-1) \ .$$

This is indeed a homogeneous system of linear equations in the variables u_{ij}. Systems of linear equations over fields can be solved in polynomial time (see Section 31.4), the operations in the ring $\mathbb{F}_q[x]/(f(x))$ can be performed in polynomial time, and the fast exponentiation also runs in polynomial time. Thus the following theorem is valid.

Theorem 5.39 *Let $f \in \mathbb{F}_q[x]$. Then it is possible to compute the Berlekamp subalgebras $B_f \leq \mathbb{F}_q[x]/(f(x))$ and $A_f \leq \mathbb{F}_q[x]/(f(x))$, in the sense that an $\mathbb{F}_q$-basis of B_f and $\mathbb{F}_p$-basis of A_f can be obtained, using polynomial time deterministic algorithms.*

By (5.6) and (5.7),

$$B_f = \{[u(x)]_f \in \mathbb{F}_q[x]/(f) : \ [u^q(x)]_{f_i^{e_i}} = [u(x)]_{f_i^{e_i}} \ \ (i = 1, \ldots, s)\} \tag{5.8}$$

and

$$A_f = \{[u(x)]_f \in \mathbb{F}_q[x]/(f) : \ [u^p(x)]_{f_i^{e_i}} = [u(x)]_{f_i^{e_i}} \ \ (i = 1, \ldots, s)\} \ . \tag{5.9}$$

The following theorem shows that the elements of the Berlekamp subalgebra can be characterised by their Chinese remainders.

Theorem 5.40

$$B_f = \{[u(x)]_f \in \mathbb{F}_q[x]/(f) : \ \exists c_i \in \mathbb{F}_q \text{ such that } [u(x)]_{f_i^{e_i}} = [c_i]_{f_i^{e_i}} \ (i = 1, \ldots, s)\}$$

and

$$A_f = \{[u(x)]_f \in \mathbb{F}_q[x]/(f) : \ \exists c_i \in \mathbb{F}_p \text{ such that } [u(x)]_{f_i^{e_i}} = [c_i]_{f_i^{e_i}} \ (i = 1, \ldots, s)\} \ .$$

Proof Using the Chinese remainder theorem, and equations (5.8), (5.9), we are only required to prove that

$$u^q(x) \equiv u(x) \pmod{g^e(x)} \Longleftrightarrow \exists c \in \mathbb{F}_q \text{ such that } u(x) \equiv c \pmod{g^e(x)} ,$$

and

$$u^p(x) \equiv u(x) \pmod{g^e(x)} \Longleftrightarrow \exists c \in \mathbb{F}_p \text{ such that } u(x) \equiv c \pmod{g^e(x)}$$

where $g(x) \in \mathbb{F}_q[x]$ is an irreducible polynomial, $u(x) \in \mathbb{F}_q[x]$ is an arbitrary polynomial and e is a positive integer. In both of the cases, the direction $\Leftarrow$ is a simple consequence of Theorem 5.19. As $\mathbb{F}_p = \{a \in \mathbb{F}_q \mid a^p = a\}$, the implication $\Rightarrow$ concerning the absolute Berlekamp subalgebra follows from that concerning the Berlekamp subalgebra, and so it suffices to consider the latter.

The residue class ring $\mathbb{F}_q[x]/(g(x))$ is a field, and so the polynomial $x^q - x$ has at most q roots in $\mathbb{F}_q[x]/(g(x))$. However, we already obtain q distinct roots from Theorem 5.19, namely the elements of $\mathbb{F}_q$ (the constant polynomials modulo $g(x)$). Thus

$$u^q(x) \equiv u(x) \pmod{g(x)} \Longleftrightarrow \exists c \in \mathbb{F}_q \text{ such that } u(x) \equiv c \pmod{g(x)} .$$

Hence, if $u^q(x) \equiv u(x) \pmod{g^e(x)}$, then $u(x)$ is of the form $u(x) = c + h(x)g(x)$ where $h(x) \in \mathbb{F}_q[x]$. Let N be an arbitrary positive integer. Then

$$u(x) \equiv u^q(x) \equiv u^{q^N}(x) \equiv (c + h(x)g(x))^{q^N} \equiv c + h(x)^{q^N} g(x)^{q^N} \equiv c \pmod{g^{q^N}(x)} .$$

If we choose N large enough so that $q^N \geq e$ holds, then, by the congruence above, $u(x) \equiv c \pmod{g^e(x)}$ also holds. ∎

An element $[u(x)]_f$ of B_f or A_f is said to be ***non-trivial*** if there is no element $c \in \mathbb{F}_q$ such that $u(x) \equiv c \pmod{f(x)}$. By the previous theorem and the Chinese remainder theorem, this holds if and only if there are i, j such that $c_i \neq c_j$. Clearly a necessary condition is that $s > 1$, that is, $f(x)$ must have at least two irreducible factors.

Lemma 5.41 *Let $[u(x)]_f$ be a non-trivial element of the Berlekamp subalgebra B_f. Then there is an element $c \in \mathbb{F}_q$ such that the polynomial* $\gcd(u(x) - c, f(x))$ *is a proper divisor of $f(x)$. If $[u(x)]_f \in A_f$, then there exists such an element c in the prime field $\mathbb{F}_p$.*

Proof Let i and j be integers such that $c_i \neq c_j \in \mathbb{F}_q$, $u(x) \equiv c_i \pmod{f_i^{e_i}(x)}$, and $u(x) \equiv c_j \pmod{f_j^{e_j}(x)}$. Then, choosing $c = c_i$, the polynomial $u(x) - c$ is divisible by $f_i^{e_i}(x)$, but not divisible by $f_j^{e_j}(x)$. If, in addition, $u(x) \in A_f$, then also $c = c_i \in \mathbb{F}_p$. ∎

Assume that we have a basis of A_f at hand. At most one of the basis elements can be trivial, as a trivial element is a scalar multiple of 1. If $f(x)$ is not a power of an irreducible polynomial, then there will surely be a non-trivial basis element $[u(x)]_f$, and so, using the idea in the previous lemma, $f(x)$ can be factored two factors.

Theorem 5.42 *A polynomial $f(x) \in \mathbb{F}_q[x]$ can be factored with a deterministic algorithm whose running time is polynomial in p,* $\deg f$, *and* $\lg q$.

Proof It suffices to show that f can be factored to *two* factors within the given time bound. The method can then be repeated.

BERLEKAMP-DETERMINISTIC(f)

```
1  S ← a basis of A_f
2  if |S| > 1
3     then u ← a non-trivial element of S
4     for c ∈ F_p
5         do g ← gcd(u − c, f)
6            if 0 < deg g < deg f
7               then return (g, f/g)
8     else return "a power of an irreducible"
```

In the first stage, in line 1, we determine a basis of the absolute Berlekamp subalgebra. The cost of this is polynomial in $\deg f$ and $\lg q$. In the second stage (lines 2–8), after taking a non-trivial basis element $[u(x)]_f$, we compute the greatest common divisors $\gcd(u(x) - c, f(x))$ for all $c \in \mathbb{F}_p$. The cost of this is polynomial in p and $\deg f$.

If there is no non-trivial basis-element, then A_f is 1-dimensional and f is the e_1-th power of the irreducible polynomial f_1 where f_1 and e_1 can, for instance, be determined using the ideas presented in Section 5.3.1. ■

The time bound in the previous theorem is *not polynomial* in the input size, as it contains p instead of $\lg p$. However, if p is small compared to the other parameters (for instance in coding theory we often have $p = 2$), then the running time of the algorithm will be polynomial in the input size.

Corollary 5.43 *Suppose that p can be bounded by a polynomial function of $\deg f$ and $\lg q$. Then the irreducible factorisation of f can be obtained in polynomial time.*

The previous two results are due to E. R. Berlekamp. The most important open problem in the area discussed here is the existence of a deterministic polynomial time method for factoring polynomials. The question is mostly of theoretical interest, since the randomised polynomial time methods, such as the a Cantor -Zassenhaus algorithm, are very efficient in practice.

Berlekamp's randomised algorithm We can obtain a good randomised algorithm using Berlekamp subalgebras. Suppose that q is odd, and, as before, $f \in \mathbb{F}_q[x]$ is the polynomial to be factored.

Let $[u(x)]_f$ be a random element in the Berlekamp subalgebra B_f. An argument, similar to the one in the analysis of the Cantor-Zassenhaus algorithm shows that, provided $f(x)$ has at least two irreducible factors, the greatest common divisor $\gcd(u(x)^{(q-1)/2} - 1, f(x))$ is a proper divisor of $f(x)$ with probability at least $4/9$. Now we present a variation of this idea that uses less random bits: instead of choosing a random element from B_f, we only choose a random element from $\mathbb{F}_q$.

Lemma 5.44 *Suppose that q is odd and let a_1 and a_2 be two distinct elements of $\mathbb{F}_q$. Then there are at least $(q-1)/2$ elements $b \in \mathbb{F}_q$ such that exactly one of the*

elements $(a_1+b)^{(q-1)/2}$ *and* $(a_2+b)^{(q-1)/2}$ *is* 1.

Proof Using the argument at the beginning of the proof of Lemma 5.37, one can easily see that there are $(q-1)/2$ elements in the set $\mathbb{F}_q \setminus \{1\}$ whose $(q-1)/2$-th power is -1. It is also quite easy to check, for a given element $c \in \mathbb{F}_q \setminus \{1\}$, that there is a unique $b \neq -a_2$ such that $c = (a_1+b)/(a_2+b)$. Indeed, the required b is the solution of a linear equation.

By the above, there are $(q-1)/2$ elements $b \in \mathbb{F}_q \setminus \{-a_2\}$ such that

$$\left(\frac{a_1+b}{a_2+b}\right)^{(q-1)/2} = -1 \,.$$

For such a b, one of the elements $(a_1+b)^{(q-1)/2}$ and $(a_2+b)^{(q-1)/2}$ is equal to 1 and the other is equal to -1. ■

Theorem 5.45 *Suppose that q is odd and the polynomial $f(x) \in \mathbb{F}_q[x]$ has at least two irreducible factors in $\mathbb{F}_q[x]$. Let $u(x)$ be a non-trivial element in the Berlekamp subalgebra B_f. If we choose a uniformly distributed random element $b \in \mathbb{F}_q$, then, with probability at least $(q-1)/(2q) \geq 1/3$, the greatest common divisor* $\gcd((u(x)+b)^{(q-1)/2}-1, f(x))$ *is a proper divisor of the polynomial $f(x)$.*

Proof Let $f(x) = \prod_{i=1}^{s} f_i^{e_i}(x)$, where the factors $f_i(x)$ are pairwise distinct irreducible polynomials. The element $[u(x)]_f$ is a non-trivial element of the Berlekamp subalgebra, and so there are indices $0 < i, j \leq s$ and elements $c_i \neq c_j \in \mathbb{F}_q$ such that $u(x) \equiv c_i \pmod{f_i^{e_i}(x)}$ and $u(x) \equiv c_j \pmod{f_j^{e_j}(x)}$. Using Lemma 5.44 with $a_1 = c_i$ and $a_2 = c_j$, we find, for a random element $b \in \mathbb{F}_q$, that the probability that exactly one of the elements $(c_i+b)^{(q-1)/2}-1$ and $(c_j+b)^{(q-1)/2}-1$ is zero is at least $(q-1)/(2q)$. If, for instance, $(c_i+b)^{(q-1)/2}-1 = 0$, but $(c_j+b)^{(q-1)/2}-1 \neq 0$, then $(u(x)+b)^{(q-1)/2}-1 \equiv 0 \pmod{f_i^{e_i}(x)}$ but $(u(x)+b)^{(q-1)/2}-1 \neq 0 \pmod{f_j^{e_j}(x)}$, that is, the polynomial $(u(x)+b)^{(q-1)/2}-1$ is divisible by $f_i^{e_i}(x)$, but not divisible by $f_j^{e_j}(x)$. Thus the greatest common divisor $\gcd(f(x), (u(x)+b)^{(q-1)/2}-1)$ is a proper divisor of f.

The quantity $(q-1)/(2q) = 1/2 - 1/(2q)$ is a strictly increasing function in q, and so it takes its smallest value for the smallest odd prime-power, namely 3. The minimum is $1/3$. ■

The previous theorem gives the following algorithm for factoring a polynomial to two factors.

BERLEKAMP-RANDOMISED(f)

```
1  S ← a basis of B_f
2  if |S| > 1
3     then u ← a non-trivial elements of S
4          c ← a random element (uniformly distributed) of F_q
5          g ← gcd((u − c)^{(q−1)/2} − 1, f)
6          if 0 < deg g < deg f
7             then return (g, f/g)
8             else return "fail"
9     else return "a power of an irreducible"
```

Exercises

5.3-1 Let $f(x) \in \mathbb{F}_p[x]$ be an irreducible polynomial, and let α be an element of the field $\mathbb{F}_p[x]/(f(x))$. Give a polynomial time algorithm for computing α^{-1}. *Hint.* Use the result of Exercise 5.1-6

5.3-2 Let $f(x) = x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 \in \mathbb{F}_2[x]$. Using the DISTINCT-DEGREE-FACTORISATION algorithm, determine the factorisation (5.4) of f.

5.3-3 Follow the steps of the Cantor-Zassenhaus algorithm to factor the polynomial $x^2 + 2x + 9 \in \mathbb{F}_{11}[x]$.

5.3-4 Let $f(x) = x^2 - 3x + 2 \in \mathbb{F}_5[x]$. Show that $\mathbb{F}_5[x]/(f(x))$ coincides with the absolute Berlekamp subalgebra of f, that is, $A_f = \mathbb{F}_5[x]/(f(x))$.

5.3-5 Let $f(x) = x^3 - x^2 + x - 1 \in \mathbb{F}_7[x]$. Using Berlekamp's algorithm, determine the irreducible factors of f: first find a non-trivial element in the Berlekamp subalgebra A_f, then use it to factor f.

5.4. Lattice reduction

Our aim in the rest of this chapter is to present the Lenstra-Lenstra-Lovász algorithm for factoring polynomials with rational coefficients. First we study a geometric problem, which is interesting also in its own right, namely finding short lattice vectors. Finding a shortest non-zero lattice vector is hard: by a result of Ajtai, if this problem could be solved in polynomial time with a randomised algorithm, then so could all the problems in the complexity class NP. For a lattice with dimension n, the lattice reduction method presented in this chapter outputs, in polynomial time, a lattice vector whose length is not greater than $2^{(n-1)/4}$ times the length of a shortest non-zero lattice vector.

5.4.1. Lattices

First, we recall a couple of concepts related to real vector spaces. Let $\mathbb{R}^n$ denote the collection of real vectors of length n. It is routine to check that $\mathbb{R}^n$ is a vector space over the field $\mathbb{R}$. The ***scalar product*** of two vectors $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_n)$ in $\mathbb{R}^n$ is defined as the number $(u, v) = u_1v_1 + u_2v_2 + \cdots + u_nv_n$. The quantity $|u| = \sqrt{(u, u)}$ is called the ***length*** of the vector u. The vectors u and v are said to be ***orthogonal*** if $(u, v) = 0$. A basis $b_1, \ldots, b_n$ of the space $\mathbb{R}^n$ is said to be orthonormal, if, for all i, $(b_i, b_i) = 1$ and, for all i and j such that $i \neq j$, we have $(b_i, b_j) = 0$.

The rank and the determinant of a real matrix, and definite matrices are discussed in Section 31.1.

Definition 5.46 *A set $L \subseteq \mathbb{R}^n$ is said to be a **lattice**, if L is a subgroup with respect to addition, and L is discrete, in the sense that each bounded region of $\mathbb{R}^n$ contains only finitely many points of L. The **rank** of the lattice L is the dimension of the subspace generated by L. Clearly, the rank of L coincides with the cardinality*

of a maximal linearly independent subset of L. If L has rank n, then L is said to be a ***full*** *lattice. The elements of L are called* ***lattice vectors*** *or* ***lattice points****.*

Definition 5.47 *Let $b_1, \ldots, b_r$ be linearly independent elements of a lattice $L \subseteq \mathbb{R}^n$. If all the elements of L can be written as linear combinations of the elements $b_1, \ldots, b_r$ with integer coefficients, then the collection $b_1, \ldots, b_r$ is said to be a* ***basis*** *of L.*

In this case, as the vectors $b_1, \ldots, b_r$ are linearly independent, all vectors of $\mathbb{R}^n$ can uniquely be written as real linear combinations of $b_1, \ldots, b_r$.

By the following theorem, the lattices are precisely those additive subgroups of $\mathbb{R}^n$ that have bases.

Theorem 5.48 *Let $b_1, \ldots, b_r$ be linearly independent vectors in $\mathbb{R}^n$ and let L be the set of integer linear combinations of $b_1, \ldots, b_r$. Then L is a lattice and the vectors $b_1, \ldots, b_r$ form a basis of L. Conversely, if L is a lattice in $\mathbb{R}^n$, then it has a basis.*

Proof Obviously, L is a subgroup, that is, it is closed under addition and subtraction. In order to show that it is discrete, let us assume that $n = r$. This assumption means no loss of generality, as the subspace spanned by $b_1, \ldots, b_r$ is isomorphic to $\mathbb{R}^r$. In this case, $\phi : (\alpha_1, \ldots, \alpha_n) \mapsto \alpha_1 b_1 + \ldots + \alpha_n b_n$ is an invertible linear map of $\mathbb{R}^n$ onto itself. Consequently, both ϕ and ϕ^{-1} are continuous. Hence the image of a discrete set under ϕ is also discrete. As $L = \phi(\mathbf{Z}^n)$, it suffices to show that $\mathbf{Z}^n$ is discrete in $\mathbb{R}^n$. This, however, is obvious: if K is a bounded region in $\mathbb{R}^n$, then there is a positive integer ρ, such that the absolute value of each of the coordinates of the elements of K is at most ρ. Thus $\mathbf{Z}^n$ has at most $(2\lfloor\rho\rfloor + 1)^n$ elements in K.

The second assertion is proved by induction on n. If $L = \{0\}$, then we have nothing to prove. Otherwise, by discreteness, there is a shortest non-zero vector, b_1 say, in L. We claim that the vectors of L that lie on the line $\{\lambda b_1 \mid \lambda \in \mathbb{R}\}$ are exactly the integer multiples of b_1. Indeed, suppose that λ is a real number and consider the vector $\lambda b_1 \in L$. As usual, $\{\lambda\}$ denotes the fractional part of λ. Then $0 \neq |\{\lambda\}b_1| < |b_1|$, yet $\{\lambda\}b_1 = \lambda b_1 - [\lambda]b_1$, that is $\{\lambda\}b_1$ is the difference of two vectors of L, and so is itself in L. This, however, contradicts to the fact that b_1 was a shortest non-zero vector in L. Thus our claim holds.

The claim verified in the previous paragraph shows that the theorem is valid when $n = 1$. Let us, hence, assume that $n > 1$. We may write an element of $\mathbb{R}^n$ as the sum of two vectors, one of them is parallel to b_1 and the other one is orthogonal to b_1:

$$v = v^* + \frac{(v, b_1)}{(b_1, b_1)} b_1 \ .$$

Simple computation shows that $(v^*, b_1) = 0$, and the map $v \mapsto v^*$ is linear. Let $L^* = \{v^* | v \in L\}$. We show that L^* is a lattice in the subspace, or hyperplane, $H \cong \mathbb{R}^{n-1}$ formed by the vectors orthogonal to b_1. The map $v \mapsto v^*$ is linear, and so L^* is closed under addition and subtraction. In order to show that it is discrete, let K be a bounded region in H. We are required to show that only finitely many points of L^* are in K. Let $v \in L$ be a vector such that $v^* \in K$. Let λ be the integer that is closest to the number $(v, b_1)/(b_1, b_1)$ and let $v' = v - \lambda b_1$. Clearly, $v' \in L$ and

$v'^* = v^*$. Further, we also have that $|(v', b_1)/(b_1, b_1)| = |(v - \lambda b_1, b_1)/(b_1, b_1)| \leq 1/2$, and so the vector v' lies in the bounded region $K \times \{\mu b_1 : \ -1/2 \leq \mu \leq 1/2\}$. However, there are only finitely many vectors $v' \in L$ in this latter region, and so K also has only finitely many lattice vectors $v^* = v'^* \in L^*$.

We have, thus, shown that L^* is a lattice in H, and, by the induction hypothesis, it has a basis. Let $b_2, \ldots, b_r \in L$ be lattice vectors such that the vectors $b_2^*, \ldots, b_r^*$ form a basis of the lattice L^*. Then, for an arbitrary lattice vector $v \in L$, the vector v^* can be written in the form $\sum_{i=2}^r \lambda_i b_i^*$ where the coefficients λ_i are integers. Then $v' = v - \sum_{i=2}^r \lambda_i b_i \in L$ and, as the map $v \mapsto v^*$ is linear, we have $v'^* = 0$. This, however, implies that v' is a lattice vector on the line λb_1, and so $v' = \lambda_1 b_1$ with some integer λ_1. Therefore $v = \sum_{i=1}^r \lambda_i b_i$, that is, v is an integer linear combination of the vectors $b_1, \ldots, b_r$. Thus the vectors $b_1, \ldots, b_r$ form a basis of L. ∎

A lattice L is always full in the linear subspace spanned by L. Thus, without loss of generality, we will consider only full lattices, and, in the sequel, by a lattice we will always mean a *full lattice* .

Example 5.4 Two familiar lattices in $\mathbb{R}^2$:
1. The ***square lattice*** is the lattice in $\mathbb{R}^2$ with basis $b_1 = (1, 0),\ b_2 = (0, 1)$.
2. The ***triangular lattice*** is the lattice with basis $b_1 = (1, 0),\ b_2 = (1/2, (\sqrt{3})/2)$.

The following simple fact will often be used.

Lemma 5.49 *Let L be a lattice in $\mathbb{R}^n$, and let $b_1, \ldots, b_n$ be a basis of L. If we reorder the basis vectors $b_1, \ldots, b_n$, or if we add to a basis vector an integer linear combination of the other basis vectors, then the collection so obtained will also form a basis of L.*

Proof Straightforward. ∎

Let $b_1, \ldots, b_n$ be a basis in L. The Gram matrix of $b_1, \ldots, b_n$ is the matrix $B = (B_{ij})$ with entries $B_{ij} = (b_i, b_j)$. The matrix B is positive definite, since it is of the form $A^T A$ where A is a full-rank matrix (see Theorem 31.6). Consequently, $\det B$ is a positive real number.

Lemma 5.50 *Let $b_1, \ldots, b_n$ and $w_1, \ldots, w_n$ be bases of a lattice L and let B and W be the matrices $B_{ij} = (b_i, b_j)$ and $W_{ij} = (w_i, w_j)$. Then the determinants of B and W coincide.*

Proof For all $i = 1, \ldots, n$, the vector w_i is of the form $w_i = \sum_{j=1}^n \alpha_{ij} b_j$ where the α_{ij} are integers. Let A be the matrix with entries $A_{ij} = \alpha_{ij}$. Then, as

$$(w_i, w_j) \quad = \quad (\sum_{k=1}^n \alpha_{ik} b_k, \sum_{l=1}^n \alpha_{jl} b_l) = \sum_{k=1}^n \alpha_{ik} \sum_{l=1}^n (b_k, b_l) \alpha_{jl} \ ,$$

we have $W = ABA^T$, and so $\det W = \det B (\det A)^2$. The number $\det W / \det B = (\det A)^2$ is a non-negative integer, since the entries of A are integers. Swapping the two bases, the same argument shows that $\det B / \det W$ is also a non-negative integer. This can only happen if $\det B = \det W$. ∎

Definition 5.51 (The determinant of a lattice). *The determinant of a lattice L is* $\det L = \sqrt{\det B}$ *where B is the Gram matrix of a basis of L.*

By the previous lemma, $\det L$ is independent of the choice of the basis. The quantity $\det L$ has a geometric meaning, as $\det L$ is the volume of the solid body, the so-called parallelepiped, formed by the vectors $\{\sum_{i=1}^n \alpha_i b_i : \ 0 \le \alpha_1, \dots, \alpha_n \le 1\}$.

Remark 5.52 *Assume that the coordinates of the vectors b_i in an orthonormal basis of $\mathbb{R}^n$ are $\alpha_{i1}, \dots, \alpha_{in}$ $(i = 1, \dots, n)$. Then the Gram matrix B of the vectors $b_1, \dots, b_n$ is $B = AA^T$ where A is the matrix $A_{ij} = \alpha_{ij}$. Consequently, if $b_1, \dots, b_n$ is a basis of a lattice L, then* $\det L = |\det A|$.

Proof The assertion follows from the equations $(b_i, b_j) = \sum_{k=1}^n \alpha_{ik}\alpha_{jk}$. ∎

5.4.2. Short lattice vectors

We will need a fundamental result in convex geometry. In order to prepare for this, we introduce some simple notation. Let $H \subseteq \mathbb{R}^n$. The set H is said to be ***centrally symmetric***, if $v \in H$ implies $-v \in H$. The set H is ***convex,*** if $u, v \in H$ implies $\lambda u + (1-\lambda)v \in H$ for all $0 \le \lambda \le 1$.

Theorem 5.53 (Minkowski's Convex Body Theorem). *Let L be a lattice in $\mathbb{R}^n$ and let $K \subseteq \mathbb{R}^n$ be a centrally symmetric, bounded, closed, convex set. Suppose that the volume of K is at least $2^n \det L$. Then $K \cap L \neq \{0\}$.*

Proof By the conditions, the volume of the set $(1/2)K := \{(1/2)v : \ v \in K\}$ is at least $\det L$. Let $b_1, \dots, b_n$ be a basis of the lattice L and let $P = \{\sum_{i=1}^n \alpha_i b_i : \ 0 \le \alpha_1, \dots, \alpha_n < 1\}$ be the corresponding half-open parallelepiped. Then each of the vectors in $\mathbb{R}^n$ can be written uniquely in the form $x + z$ where $x \in L$ and $z \in P$. For an arbitrary lattice vector $x \in L$, we let

$$K_x = (1/2)K \cap (x + P) = (1/2)K \cap \{x + z : \ z \in P\} \ .$$

As the sets $(1/2)K$ and P are bounded, so is the set

$$(1/2)K - P = \{u - v : \ u \in (1/2) \cdot K, \ v \in P\} .$$

As L is discrete, L only has finitely many points in $(1/2)K - P$; that is, $K_x = \emptyset$, except for finitely many $x \in L$. Hence $S = \{x \in L : \ K_x \neq \emptyset\}$ is a finite set, and, moreover, the set $(1/2)K$ is the disjoint union of the sets K_x $(x \in S)$. Therefore, the total volume of these sets is at least $\det L$. For a given $x \in S$, we set $P_x = K_x - x = \{z \in P : \ x + z \in (1/2)K\}$. Consider the closure $\overline{P}$ and $\overline{P_x}$ of the sets P and P_x, respectively:

$$\overline{P} = \left\{ \sum_{i=1}^n \alpha_i b_i : \ 0 \le \alpha_1, \dots, \alpha_n \le 1 \right\}$$

and $\overline{P_x} = \left\{ z \in \overline{P} : \ x + z \in (1/2)K \right\}$. The total volume of the closed sets $\overline{P_x} \subseteq \overline{P}$ is at least as large as the volume of the set $\overline{P}$, and so these sets cannot be disjoint:

there are $x \neq y \in S$ and $z \in \overline{P}$ such that $z \in \overline{P_x} \cap \overline{P_y}$, that is, $x + z \in (1/2)K$ and $y + z \in (1/2)K$. As $(1/2) \cdot K$ is centrally symmetric, we find that $-y - z \in (1/2) \cdot K$. As $(1/2)K$ is convex, we also have $(x - y)/2 = ((x + z) + (-y - z))/2 \in (1/2)K$. Hence $x - y \in K$. On the other hand, the difference $x - y$ of two lattice points lies in $L \setminus \{0\}$. ∎

Minkowski's theorem is sharp. For, let $\epsilon > 0$ be an arbitrary positive number, and let $L = \mathbf{Z}^n$ be the lattice of points with integer coordinates in $\mathbb{R}^n$. Let K be the set of vectors $(v_1, \ldots, v_n) \in \mathbb{R}^n$ for which $-1 + \epsilon \leq v_i \leq 1 - \epsilon$ $(i = 1, \ldots, n)$. Then K is bounded, closed, convex, centrally symmetric with respect to the origin, its volume is $(1 - \epsilon)^n 2^n \det L$, yet $L \cap K = \{0\}$.

Corollary 5.54 *Let L be a lattice in $\mathbb{R}^n$. Then L has a lattice vector $v \neq 0$ whose length is at most $\sqrt{n} \sqrt[n]{\det L}$.*

Proof Let K be the following centrally symmetric cube with side length $s = 2 \sqrt[n]{\det L}$:

$$K = \{(v_1, \ldots, v_n) \in \mathbb{R}^n : \ -s/2 \leq v_i \leq s/2, \ i = 1, \ldots, n\} .$$

The volume of the cube K is exactly $2^n \det L$, and so it contains a non-zero lattice vector. However, the vectors in K have length at most $\sqrt{n} \sqrt[n]{\det L}$. ∎

We remark that, for $n > 1$, we can find an even shorter lattice vector, if we replace the cube in the proof of the previous assertion by a suitable ball.

5.4.3. Gauss' algorithm for two-dimensional lattices

Our goal is to design an algorithm that finds a non-zero short vector in a given lattice. In this section we consider this problem for two-dimensional lattices, which is the simplest non-trivial case. Then there is an elegant, instructive, and efficient algorithm that finds short lattice vectors. This algorithm also serves as a basis for the higher-dimensional cases. Let L be a lattice with basis b_1, b_2 in $\mathbb{R}^2$.

GAUSS(b_1, b_2)

```
1  (a, b) ← (b_1, b_2)
2  forever
3        do b ← the shortest lattice vector on the line b − λa
4        if |b| < |a|
5           then b ↔ a
6           else return (a, b)
```

In order to analyse the procedure, the following facts will be useful.

Lemma 5.55 *Suppose that a and b are two linearly independent vectors in the plane $\mathbb{R}^2$, and let L be the lattice generated by them. The vector b is a shortest non-zero vector of L on the line $b - \lambda a$ if and only if*

$$|(b, a)/(a, a)| \leq 1/2 . \tag{5.10}$$

Proof We write b as the sum of a vector parallel to a and a vector orthogonal to a:

$$b = (b,a)/(a,a)a + b^* \ . \tag{5.11}$$

Then, as the vectors a and b^* are orthogonal,

$$|b - \lambda a|^2 = \left| \left(\frac{(b,a)}{(a,a)} - \lambda \right) a + b^* \right|^2 = \left(\frac{(b,a)}{(a,a)} - \lambda \right)^2 |a|^2 + |b^*|^2 \ .$$

This quantity takes its smallest value for the integer λ that is the closest to the number $(b,a)/(a,a)$. Hence $\lambda = 0$ gives the minimal value if and only if (5.10) holds. ■

Lemma 5.56 *Suppose that the linearly independent vectors a and b form a basis for a lattice $L \subseteq \mathbb{R}^2$ and that inequality (5.10) holds. Assume, further, that*

$$|b|^2 \geq (3/4)|a|^2 \ . \tag{5.12}$$

Write b, as in (5.11), as the sum of the vector $((b,a)/(a,a))a$, which is parallel to a, and the vector $b^* = b - ((b,a)/(a,a))a$, which is orthogonal to a. Then

$$|b^*|^2 \geq (1/2)|a|^2 \ . \tag{5.13}$$

Further, either b or a is a shortest non-zero vector in L.

Proof By the assumptions,

$$|a|^2 \leq \frac{4}{3}|b|^2 = \frac{4}{3}|b^*|^2 + \frac{4}{3}\left((b,a)/(a,a)\right)^2 |a|^2 \leq \frac{4}{3}|b^*|^2 + (1/3)|a|^2 \ .$$

Rearranging the last displayed line, we obtain $|b^*|^2 \geq (1/2)|a|^2$.

The length of a vector $0 \neq v = \alpha a + \beta b \in L$ can be computed as

$$|\alpha a + \beta b|^2 = |\beta b^*|^2 + \left(\alpha + \beta(b,a)/(a,a)\right)^2 |a|^2 \geq \beta^2 |b^*|^2 \geq (1/2)\beta^2 |a|^2,$$

which implies $|v| > |a|$ whenever $|\beta| \geq 2$. If $\beta = 0$ and $\alpha \neq 0$, then $|v| = |\alpha| \cdot |a| \geq |a|$. Similarly, $\alpha = 0$ and $\beta \neq 0$ gives $|v| = |\beta| \cdot |b| \geq |b|$. It remains to consider the case when $\alpha \neq 0$ and $\beta = \pm 1$. As $|-v| = |v|$, we may assume that $\beta = 1$. In this case, however, v is of the form $v = b - \lambda a$ ($\lambda = -\alpha$), and, by Lemma 5.55, the vector b is a shortest lattice vector on this line. ■

Theorem 5.57 *Let v be a shortest non-zero lattice vector in L. Then Gauss' algorithm terminates after $O(1 + \lg(|b_1|/|v|))$ iterations, and the resulting vector a is a shortest non-zero vector in L.*

Proof First we verify that, during the course of the algorithm, the vectors a and b will always form a basis for the lattice L. If, in line 3, we replace b by a vector of the form $b' = b - \lambda a$, then, as $b = b' + \lambda a$, the pair a, b' remains a basis of L. The swap in line 5 only concerns the order of the basis vectors. Thus a and b is always a basis of L, as we claimed.

By Lemma 5.55, inequality (5.10) holds after the first step (line 3) in the loop,

and so we may apply Lemma 5.56 to the scenario before lines 4–5. This shows that if none of a and b is shortest, then $|b|^2 \leq (3/4)|a|^2$. Thus, except perhaps for the last execution of the loop, after each swap in line 5, the length of a is decreased by a factor of at least $\sqrt{3/4}$. Thus we obtain the bound for the number of executions of the loop. Lemma 5.56 implies also that the vector a at the end is a shortest non-zero vector in L. ∎

Gauss' algorithm gives an efficient polynomial time method for computing a shortest vector in the lattice $L \subseteq \mathbb{R}^2$. The analysis of the algorithm gives the following interesting theoretical consequence.

Corollary 5.58 *Let L be a lattice in $\mathbb{R}^2$, and let a be a shortest non-zero lattice vector in L. Then $|a|^2 \leq (2/\sqrt{3}) \det L$.*

Proof Let b be a vector in L such that b is linearly independent of a and (5.10) holds. Then

$$|a|^2 \leq |b|^2 = |b^*|^2 + \left(\frac{(b,a)}{(a,a)}\right)^2 |a|^2 \leq |b^*|^2 + \frac{1}{4}|a|^2 \ ,$$

which yields $(3/4)|a|^2 \leq |b^*|^2$. The area of the fundamental parallelogram can be computed using the well-known formula

$$\text{area} = \text{base} \cdot \text{height},$$

and so $\det L = |a||b^*|$. The number $|b^*|$ can now be bounded by the previous inequality. ∎

5.4.4. A Gram-Schmidt orthogonalisation and weak reduction

Let $b_1, \ldots, b_n$ be a linearly independent collection of vectors in $\mathbb{R}^n$. For an index i with $i \in \{1, \ldots, n\}$, we let b_i^* denote the component of b_i that is orthogonal to the subspace spanned by $b_1, \ldots, b_{i-1}$. That is,

$$b_i = b_i^* + \sum_{j=1}^{i-1} \lambda_{ij} b_j \ ,$$

where

$$(b_i^*, b_j) = 0 \quad \text{for} \quad j = 1, \ldots, i-1 \ .$$

Clearly $b_1^* = b_1$. The vectors $b_1^*, \ldots, b_{i-1}^*$ span the same subspace as the vectors $b_1, \ldots, b_{i-1}$, and so, with suitable coefficients μ_{ij}, we may write

$$b_i = b_i^* + \sum_{j=1}^{i-1} \mu_{ij} b_j^* \ , \tag{5.14}$$

and

$$(b_i^*, b_j^*) = 0, \quad \text{if } j \neq i \ .$$

By the latter equations, the vectors $b_1^*, \ldots, b_{i-1}^*, b_i^*$ form an orthogonal system, and so

$$\mu_{ij} = \frac{(b_i, b_j^*)}{(b_j^*, b_j^*)} \qquad (j = 1, \ldots, i-1) \; . \tag{5.15}$$

The set of the vectors $b_1^*, \ldots, b_n^*$ is said to be the ***Gram-Schmidt orthogonalisation*** of the vectors $b_1, \ldots, b_n$.

Lemma 5.59 *Let $L \subseteq \mathbb{R}^n$ be a lattice with basis $b_1, \ldots, b_n$. Then*

$$\det L = \prod_{i=1}^{n} |b_i^*| \; .$$

Proof Set $\mu_{ii} = 1$ and $\mu_{ij} = 0$, if $j > i$. Then $b_i^* = \sum_{k=1}^{n} \mu_{ik} b_k$, and so

$$(b_i^*, b_j^*) = \sum_{k=1}^{n} \mu_{ik} \sum_{l=1}^{n} (b_k, b_l) \mu_{jl},$$

that is, $B^* = MBM^T$ where B and B^* are the Gram matrices of the collections $b_1, \ldots, b_n$ and $b_1^*, \ldots, b_n^*$, respectively, and M is the matrix with entries μ_{ij}. The matrix M is a lower triangular matrix with ones in the main diagonal, and so $\det M = \det M^T = 1$. As B^* is a diagonal matrix, we obtain $\prod_{i=1}^{n} |b_i^*|^2 = \det B^* = (\det M)(\det B)(\det M^T) = \det B$. ∎

Corollary 5.60 (Hadamard inequality). $\prod_{i=1}^{n} |b_i| \geq \det L$.

Proof The vector b_i can be written as the sum of the vector b_i^* and a vector orthogonal to b_i^*, and hence $|b_i^*| \leq |b_i|$. ∎

The vector b_i^* is the component of b_i orthogonal to the subspace spanned by the vectors $b_1, \ldots, b_{i-1}$. Thus b_i^* does not change if we subtract a linear combination of the vectors $b_1, \ldots, b_{i-1}$ from b_i. If, in this linear combination, the coefficients are integers, then the new sequence $b_1, \ldots, b_n$ will be a basis of the same lattice as the original. Similarly to the first step of the loop in Gauss' algorithm, we can make the numbers μ_{ij} in (5.15) small. The input of the following procedure is a basis $b_1, \ldots, b_n$ of a lattice L.

WEAK-REDUCTION$(b_1, \ldots, b_n)$

1 **for** $j \leftarrow n-1$ **downto** 1
2 **do for** $i \leftarrow j+1$ **to** n
3 $b_i \leftarrow b_i - \lambda b_j$, where λ is the integer nearest the number $(b_i, b_j^*)/(b_j^*, b_j^*)$
4 **return** $(b_1, \ldots, b_n)$

Definition 5.61 (Weakly reduced basis). *A basis $b_1, \ldots, b_n$ of a lattice is said to be **weakly reduced** if the coefficients μ_{ij} in (5.15) satisfy*

$$|\mu_{ij}| \leq \frac{1}{2} \quad \textit{for} \quad 1 \leq j < i \leq n \; .$$

Lemma 5.62 *The basis given by the procedure* WEAK-REDUCTION *is weakly reduced.*

Proof By the remark preceding the algorithm, we obtain that the vectors $b_1^*, \ldots, b_n^*$ never change. Indeed, we only subtract linear combinations of vectors with index less than i from b_i. Hence the inner instruction does not change the value of (b_k, b_l^*) with $k \neq i$. The values of the (b_i, b_l^*) do not change for $l > j$ either. On the other hand, the instruction achieves, with the new b_i, that the inequality $|\mu_{ij}| \leq 1/2$ holds:

$$|(b_i - \lambda b_j^*, b_j^*)| = |(b_i, b_j^*) - \lambda(b_j^*, b_j^*)| = |(b_i, b_j^*) - \lambda(b_j^*, b_j^*)| \leq \frac{1}{2}(b_j^*, b_j^*) \ .$$

By the observations above, this inequality remains valid during the execution of the procedure. ∎

5.4.5. Lovász-reduction

First we define, in an arbitrary dimension, a property of the bases that usually turns out to be useful. The definition will be of a technical nature. Later we will see that these bases are interesting, in the sense that they consist of short vectors. This property will make them widely applicable.

Definition 5.63 *A basis $b_1, \ldots, b_n$ of a lattice L is said to be* ***(Lovász-)reduced*** *if*

- *it is weakly reduced,*

and, using the notation introduced for the Gram-Schmidt orthogonalisation,

- $|b_i^*|^2 \leq (/3)|b_{i+1}^* + \mu_{i+1,i} b_i^*|^2$ *for all* $1 \leq i < n$.

Let us observe the analogy of the conditions above to the inequalities that we have seen when investigating Gauss' algorithm. For $i = 1$, $a = b_1$ and $b = b_2$, being weakly reduced ensures that b is a shortest vector on the line $b - \lambda a$. The second condition is equivalent to the inequality $|b|^2 \geq (3/4)|a|^2$, but here it is expressed in terms of the Gram-Schmidt basis. For a general index i, the same is true, if a plays the rôle of the vector b_i, and b plays the rôle of the component of the vector b_{i+1} that is orthogonal to the subspace spanned by $b_1, \ldots, b_{i-1}$.

LOVÁSZ-REDUCTION$(b_1, \ldots, b_n)$

```
1  forever
2        do (b_1, ..., b_n) ←WEAK-REDUCTION(b_1, ..., b_n)
3        find an index i for which the second condition of being reduced is violated
4        if there is such an i
5           then b_i ↔ b_{i+1}
6           else  return (b_1, ..., b_n)
```

Theorem 5.64 *Suppose that in the lattice $L \subseteq \mathbb{R}^n$ each of the pairs of the lattice*

vectors has an integer scalar product. Then the swap in the 5th line of the LOVÁSZ-REDUCTION *occurs at most* $\lg_{4/3}(B_1 \cdots B_{n-1})$ *times where* B_i *is the upper left* $(i \times i)$*-subdeterminant of the Gram matrix of the initial basis* $b_1, \ldots, b_n$*.*

Proof The determinant B_i is the determinant of the Gram matrix of $b_1, \ldots, b_i$, and, by the observations we made at the discussion of the Gram-Schmidt orthogonalisation, $B_i = \prod_{j=1}^{i} |b_j^*|^2$. This, of course, implies that $B_i = B_{i-1}|b_i^*|^2$ for $i > 1$. By the above, the procedure WEAK-REDUCTION cannot change the vectors b_i^*, and so it does not change the product $\prod_{j=1}^{n-1} B_j$ either. Assume, in line 5 of the procedure, that a swap $b_i \leftrightarrow b_{i+1}$ takes place. Observe that, unless $j = i$, the sets $\{b_1, \ldots, b_j\}$ do not change, and neither do the determinants B_j. The rôle of the vector b_i^* is taken over by the vector $b_{i+1}^* + \mu_{i,i+1} b_i$, whose length, because of the conditions of the swap, is at most $\sqrt{3/4}$ times the length of b_i^*. That is, the new B_i is at most $3/4$ times the old. By the observation above, the new value of $B = \prod_{j=1}^{n-1} B_j$ will also be at most $3/4$ times the old one. Then the assertion follows from the fact that the quantity B remains a positive integer. ∎

Corollary 5.65 *Under the conditions of the previous theorem, the cost of the procedure* LOVÁSZ-REDUCTION *is at most* $O(n^5 \lg nC)$ *arithmetic operations with rational numbers where* C *is the maximum of* 2 *and the quantities* $|(b_i, b_j)|$ *with* $i, j = 1, \ldots, n$*.*

Proof It follows from the Hadamard inequality that

$$B_i \leq \prod_{j=1}^{i} \sqrt{(b_1, b_j)^2 + \ldots + (b_i, b_j)^2} \leq (\sqrt{i}C)^i \leq (\sqrt{n}C)^n \ .$$

Hence $B_1 \cdots B_{n-1} \leq (\sqrt{n}C)^{n(n-1)}$ and $\lg_{4/3}(B_1 \ldots B_{n-1}) = O(n^2 \lg nC)$. By the previous theorem, this is the number of iterations in the algorithm. The cost of the Gram-Schmidt orthogonalisation is $O(n^3)$ operations, and the cost of weak reduction is $O(n^2)$ scalar product computations, each of which can be performed using $O(n)$ operations (provided the vectors are represented by their coordinates in an orthogonal basis). ∎

One can show that the length of the integers that occur during the run of the algorithm (including the numerators and the denominators of the fractions in the Gram-Schmidt orthogonalisation) will be below a polynomial bound.

5.4.6. Properties of reduced bases

Theorem 5.67 of this section gives a summary of the properties of reduced bases that turn out to be useful in their applications. We will find that a reduced basis consists of relatively short vectors. More precisely, $|b_1|$ will approximate, within a constant factor depending only on the dimension, the length of a shortest non-zero lattice vector.

Lemma 5.66 *Let us assume that the vectors* $b_1, \ldots, b_n$ *form a reduced basis of a lattice* L*. Then, for* $1 \leq j \leq i \leq n$*,*

$$(b_i^*, b_i^*) \geq 2^{j-i}(b_j^*, b_j^*) \ . \tag{5.16}$$

In particular,

$$(b_i^*, b_i^*) \geq 2^{1-i}(b_1^*, b_1^*) \ . \tag{5.17}$$

Proof Substituting $a = b_i^*$, $b = b_{i+1}^* + ((b_{i+1}, b_i^*))/((b_i^*, b_i^*)b_i^*)$, Lemma 5.56 gives, for all $1 \leq i < n$, that

$$(b_{i+1}^*, b_{i+1}^*) \geq (1/2)(b_i^*, b_i^*) \ .$$

Thus, inequality (5.16) follows by induction. ∎

Now we can formulate the fundamental theorem of reduced bases.

Theorem 5.67 *Assume that the vectors $b_1, \ldots, b_n$ form a reduced basis of a lattice L. Then*

(i) $|b_1| \leq 2^{(n-1)/4}(\det L)^{(1/n)}$.

(ii) $|b_1| \leq 2^{(n-1)/2}|b|$ *for all lattice vectors* $0 \neq b \in L$. *In particular, the length of* b_1 *is not greater than* $2^{(n-1)/2}$ *times the length of a shortest non-zero lattice vector.*

(iii) $|b_1| \cdots |b_n| \leq 2^{(n(n-1))/4} \det L$.

Proof (i) Using inequality (5.17),

$$(\det L)^2 = \prod_{i=1}^{n}(b_i^*, b_i^*) \geq \prod_{i=1}^{n}(2^{1-i}(b_1, b_1)) = 2^{\frac{-n(n-1)}{2}}(b_1, b_1)^n \ ,$$

and so assertion (i) holds.

(ii) Let $b = \sum_{i=1}^{n} z_i b_i \in L$ with $z_i \in \mathbf{Z}$ be a lattice vector. Assume that z_j is the last non-zero coefficient and write $b_j = b_j^* + v$ where v is a linear combination of the vectors $b_1, \ldots, b_{j-1}$. Hence $b = z_j b_j^* + w$ where w lies in the subspace spanned by $b_1, \ldots, b_{j-1}$. As b_j^* is orthogonal to this subspace,

$$(b, b) = z_j^2(b_j^*, b_j^*) + (w, w) \geq (b_j^*, b_j^*) \geq 2^{1-j}(b_1, b_1) \geq 2^{1-n}(b_1, b_1) \ ,$$

and so assertion (ii) is valid.

(iii) First we show that $(b_i, b_i) \leq 2^{i-1}(b_i^*, b_i^*)$. This inequality is obvious if $i = 1$, and so we assume that $i > 1$. Using the decomposition (5.14) of the vector b_i and the fact that the basis is weakly reduced, we obtain that

$$\begin{aligned}(b_i, b_i) &= \sum_{j=1}^{i}\left(\frac{(b_i, b_j^*)}{(b_j^*, b_j^*)}\right)^2 (b_j^*, b_j^*) \leq (b_i^*, b_i^*) + \frac{1}{4}\sum_{j=1}^{i-1}(b_j^*, b_j^*) \leq (b_i^*, b_i^*) + \frac{1}{4}\sum_{j=1}^{i-1} 2^{i-j}(b_i^*, b_i^*) \\ &\leq (2^{i-2} + 1)(b_i^*, b_i^*) \leq 2^{i-1}(b_i^*, b_i^*) \ .\end{aligned}$$

Multiplying these inequalities for $i = 1, \ldots, n$,

$$\prod_{i=1}^{n}(b_i, b_i) \leq \prod_{i=1}^{n} 2^{i-1}(b_i^*, b_i^*) = 2^{\frac{n(n-1)}{2}} \prod_{i=1}^{n}(b_i^*, b_i^*) = 2^{\frac{n(n-1)}{2}}(\det L)^2 \ ,$$

which is precisely the inequality in (iii). ■

It is interesting to compare assertion (i) in the previous theorem and Corollary 5.54 after Minkowski's theorem. Here we obtain a weaker bound for the length of b_1, but this vector can be obtained by an efficient algorithm. Essentially, the existence of the basis that satisfies assertion (iii) was first shown by Hermite using the tools in the proofs of Theorems 5.48 and 5.67. Using a Lovász-reduced basis, the cost of finding a shortest vector in a lattice with dimension n is at most polynomial in the input size and in 3^{n^2}; see Exercise 5.4-4.

Exercises

5.4-1 *The triangular lattice is optimal.* Show that the bound in Corollary 5.58 is sharp. More precisely, let $L \subseteq \mathbb{R}^2$ be a full lattice and let $0 \neq a \in L$ be a shortest vector in L. Verify that the inequality $|a|^2 = (2/\sqrt{3}) \det L$ holds if and only if L is similar to the triangular lattice.

5.4-2 *The denominators of the Gram-Schmidt numbers.* Let us assume that the Gram matrix of a basis $b_1, \ldots, b_n$ has only integer entries. Show that the numbers μ_{ij} in (5.15) can be written in the form $\mu_{ij} = \zeta_{ij} / \prod_{k=1}^{j-1} B_k$ where the ζ_{ij} are integers and B_k is the determinant of the Gram matrix of the vectors $b_1, \ldots, b_k$.

5.4-3 *The length of the vectors in a reduced basis.* Let $b_1, \ldots, b_n$ be a reduced basis of a lattice L and let us assume that the numbers (b_i, b_i) are integers. Give an upper bound depending only on n and $\det L$ for the length of the vectors b_i. More precisely, prove that

$$|b_i| \leq 2^{\frac{n(n-1)}{4}} \det L \ .$$

5.4-4 *The coordinates of a shortest lattice vector.* Let $b_1, \ldots, b_n$ be a reduced basis of a lattice L. Show that each of the shortest vectors in L is of the form $\sum z_i b_i$ where $z_i \in \mathbf{Z}$ and $|z_i| \leq 3^n$. Consequently, for a bounded n, one can find a shortest non-zero lattice vector in polynomial time.

Hint. Assume, for some lattice vector $v = \sum z_i b_i$, that $|v| \leq |b_1|$. Let us write v in the basis $b_1^*, \ldots, b_n^*$:

$$v = \sum_{j=1}^{n} (z_j + \sum_{i=j+1}^{n} \mu_{ij} z_i) b_j^* \ .$$

It follows from the assumption that each of the components of v (in the orthogonal basis) is at most as long as $b_1 = b_1^*$:

$$\left| z_j + \sum_{i=j+1}^{n} \mu_{ij} z_i \right| \leq \frac{|b_1^*|}{|b_j^*|} \ .$$

Use then the inequalities $|\mu_{ij}| \leq 1/2$ and (5.17).

5.5. Factoring polynomials in $\mathbb{Q}[x]$

In this section we study the problem of factoring polynomials with rational coefficients. The input of the ***factorisation problem*** is a polynomial $f(x) \in \mathbb{Q}[x]$. Our

goal is to compute a factorisation

$$f = f_1^{e_1} f_2^{e_2} \cdots f_s^{e_s}, \tag{5.18}$$

where the polynomials $f_1, \ldots, f_s$ are pairwise relatively prime, and irreducible over $\mathbb{Q}$, and the numbers e_i are positive integers. By Theorem 5.4, f determines, essentially uniquely, the polynomials f_i and the exponents e_i.

5.5.1. Preparations

First we reduce the problem (5.18) to another problem that can be handled more easily.

Lemma 5.68 *We may assume that the polynomial $f(x)$ has integer coefficients and it has leading coefficient* 1.

Proof Multiplying by the common denominator of the coefficients, we may assume that $f(x) = a_0 + a_1 x + \cdots + a_n x^n \in \mathbf{Z}[x]$. Performing the substitution $y = a_n x$, we obtain the polynomial

$$g(y) = {a_n}^{n-1} f\left(\frac{y}{a_n}\right) = y^n + \sum_{i=0}^{n-1} a_n^{n-i-1} a_i y^i \ ,$$

which has integer coefficients and its leading coefficient is 1. Using a factorisation of $g(y)$, a factorisation of $f(x)$ can be obtained efficiently. ∎

Primitive polynomials, Gauss' lemma

Definition 5.69 *A polynomial $f(x) \in \mathbf{Z}[x]$ is said to be **primitive**, if the greatest common divisor of its coefficients is* 1.

A polynomial $f(x) \in \mathbf{Z}[x] \setminus \{0\}$ can be written in a unique way as the product of an integer and a primitive polynomial in $\mathbf{Z}[x]$. Indeed, if a is the greatest common divisor of the coefficients, then $f(x) = a(1/a)f(x)$. Clearly, $(1/a)f(x)$ is a primitive polynomial with integer coefficients.

Lemma 5.70 (Gauss' Lemma). *If $u(x),\ v(x) \in \mathbf{Z}[x]$ are primitive polynomials, then so is the product $u(x)v(x)$.*

Proof We argue by contradiction and assume that p is a prime number that divides all the coefficients of uv. Set $u(x) = \sum_{i=0}^{n} u_i x^i$, $v(x) = \sum_{j=0}^{m} v_j x^j$ and let i_0 and j_0 be the smallest indices such that $p \nmid u_{i_0}$ and $p \nmid v_{j_0}$. Let $k_0 = i_0 + j_0$ and consider the coefficient of x^{k_0} in the product $u(x)v(x)$. This coefficient is

$$\sum_{i+j=k_0} u_i v_j = u_{i_0} v_{j_0} + \sum_{i=0}^{i_0-1} u_i v_{k_0-i} + \sum_{j=0}^{j_0-1} u_{k_0-j} v_j \ .$$

Both of the sums on the right-hand side of this equation are divisible by p, while $u_{i_0} v_{j_0}$ is not, and hence the coefficient of x^{k_0} in $u(x)v(x)$ cannot be divisible by p after all. This, however, is a contradiction. ∎

Proposition 5.71 *Let us assume that $g(x),\ h(x) \in \mathbb{Q}[x]$ are polynomials with rational coefficients and leading coefficient 1 such that the product $g(x)h(x)$ has integer coefficients. Then the polynomials $g(x)$ and $h(x)$ have integer coefficients.*

Proof Let us multiply $g(x)$ and $h(x)$ by the least common multiple c_g and c_h, respectively, of the denominators of their coefficients. Then the polynomials $c_g g(x)$ and $c_h h(x)$ are primitive polynomials with integer coefficients. Hence, by Gauss' Lemma, so is the product $c_g c_h g(x)h(x) = (c_g g(x))(c_h h(x))$. As the coefficients of $g(x)h(x)$ are integers, each of its coefficients is divisible by the integer $c_g c_h$. Hence $c_g c_h = 1$, and so $c_g = c_h = 1$. Therefore $g(x)$ and $h(x)$ are indeed polynomials with integer coefficients. ∎

One can show similarly, for a polynomial $f(x) \in \mathbf{Z}[x]$, that factoring $f(x)$ in $\mathbf{Z}[x]$ is equivalent to factoring the primitive part of $f(x)$ in $\mathbb{Q}[x]$ and factoring an integer, namely the greatest common divisor of the coefficients

Mignotte's bound As we work over an infinite field, we have to pay attention to the size of the results in our computations.

Definition 5.72 *The **norm** of a polynomial $f(x) = \sum_{i=0}^{n} a_i x^i \in \mathbb{C}[x]$ with complex coefficients is the real number $\|f(x)\| = \sqrt{\sum_{i=0}^{n} |a_i|^2}$.*

The inequality $\max_{i=0}^{n} |a_i| \leq \|f(x)\|$ implies that a polynomial $f(x)$ with integer coefficients can be described using $O(n \lg \|f(x)\|)$ bits.

Lemma 5.73 *Let $f(x) \in \mathbb{C}[x]$ be a polynomial with complex coefficients. Then, for all $c \in \mathbb{C}$, we have*

$$\|(x-c)f(x)\| = \|(\bar{c}x - 1)f(x)\| \ ,$$

where $\bar{c}$ is the usual conjugate of the complex number c.

Proof Let us assume that $f(x) = \sum_{i=0}^{n} a_i x^i$ and set $a_{n+1} = a_{-1} = 0$. Then

$$(x-c)f(x) = \sum_{i=0}^{n+1} (a_{i-1} - ca_i)x^i \ ,$$

and hence

$$\begin{aligned} \|(x-c)f(x)\|^2 &= \sum_{i=0}^{n+1} |a_{i-1} - ca_i|^2 = \sum_{i=0}^{n+1} (|a_{i-1}|^2 + |ca_i|^2 - a_{i-1}\overline{ca_i} - \overline{a_{i-1}}ca_i) \\ &= \|f(x)\|^2 + |c|^2 \|f(x)\|^2 - \sum_{i=0}^{n+1} (a_{i-1}\overline{ca_i} + \overline{a_{i-1}}ca_i) \ . \end{aligned}$$

Performing similar computations with the right-hand side of the equation in the lemma, we obtain that

$$(\bar{c}x - 1)f(x) = \sum_{i=0}^{n+1} (\bar{c}a_{i-1} - a_i)x^i \ ,$$

and so

$$\begin{aligned}\|(\overline{c}x-1)f(x)\|^2 &= \sum_{i=0}^{n+1}|\overline{c}a_{i-1}-a_i|^2=\sum_{i=0}^{n+1}(|\overline{c}a_{i-1}|^2+|a_i|^2-\overline{c}a_{i-1}\overline{a_i}-c\overline{a_{i-1}}a_i)\\ &= \|f(x)\|^2+|c|^2\|f(x)\|^2-\sum_{i=0}^{n+1}(a_{i-1}\overline{ca_i}+\overline{a_{i-1}}ca_i)\ .\end{aligned}$$

The proof of the lemma is now complete. ∎

Theorem 5.74 (Mignotte). *Let us assume that the polynomials $f(x), g(x) \in \mathbb{C}[x]$ have complex coefficients and leading coefficient 1 and that $g(x)|f(x)$. If $\deg(g(x)) = m$, then $\|g(x)\| \leq 2^m \|f(x)\|$.*

Proof By the fundamental theorem of algebra, $f(x) = \prod_{i=1}^{n}(x-\alpha_i)$ where $\alpha_1, \ldots, \alpha_n$ are the complex roots of the polynomial $f(x)$ (with multiplicity). Then there is a subset $I \subseteq \{1, \ldots, n\}$ such that $g(x) = \prod_{i\in I}(x-\alpha_i)$. First we claim, for an arbitrary set $J \subseteq \{1, \ldots, n\}$, that

$$\prod_{i\in J}|\alpha_i| \leq \|f(x)\|\ . \tag{5.19}$$

If J contains an integer i with $\alpha_i = 0$, then this inequality will trivially hold. Let us hence assume that $\alpha_i \neq 0$ for every $i \in J$. Set $\overline{J} = \{1, \ldots, n\} \setminus J$ and $h(x) = \prod_{i\in\overline{J}}(x-\alpha_i)$. Applying Lemma 5.73 several times, we obtain that

$$\|f(x)\| = \left\|\prod_{i\in J}(x-\alpha_i)h(x)\right\| = \left\|\prod_{i\in J}(\overline{\alpha_i}x-1)h(x)\right\| = |\prod_{i\in J}\overline{\alpha_i}|\cdot\|u(x)\|\ ,$$

where $u(x) = \prod_{i\in J}(x-1/\overline{\alpha_i})h(x)$. As the leading coefficient of $u(x)$ is 1, $\|u(x)\| \geq 1$, and so

$$|\prod_{i\in J}\alpha_i| = |\prod_{i\in J}\overline{\alpha_i}| = \|f(x)\|\,/\,\|u(x)\| \leq \|f(x)\|\ .$$

Let us express the coefficients of $g(x)$ using its roots:

$$\begin{aligned}g(x) &= \prod_{i\in I}(x-\alpha_i)=\sum_{J\subseteq I}\left((-1)^{|J|}\prod_{j\in J}\alpha_j x^{m-|J|}\right)\\ &= \sum_{i=0}^{m}(-1)^{m-i}\left(\sum_{J\subseteq I,|J|=m-i}\prod_{j\in J}\alpha_j\right)x^i\ .\end{aligned}$$

For an arbitrary polynomial $t(x) = t_0 + \cdots + t_k x^k$, the inequality $\|t(x)\| \leq |t_0| + \cdots + |t_k|$ is valid. Therefore, using inequality (5.19), we find that

$$\begin{aligned}\|g(x)\| &\leq \sum_{i=0}^{m}\left|\sum_{J\subseteq I,|J|=m-i}\prod_{j\in J}\alpha_j\right|\\ &\leq \sum_{J\subseteq I}\left|\prod_{j\in J}\alpha_j\right| \leq 2^m\|f(x)\|\ .\end{aligned}$$

The proof is now complete. ■

Corollary 5.75 *The bit size of the irreducible factors in $\mathbb{Q}[x]$ of an $f(x) \in \mathbf{Z}[x]$ with leading coefficient 1 is polynomial in the bit size of $f(x)$.*

Resultant and good reduction Let $\mathbb{F}$ be an arbitrary field, and let $f(x),\ g(x) \in \mathbb{F}[x]$ be polynomials with degree n and m, respectively: $f = a_0 + a_1x + \ldots + a_nx^n$, $g = b_0 + b_1x + \ldots + b_mx^m$ where $a_n \neq 0 \neq b_m$. We recall the concept of the ***resultant*** from Chapter 3. The resultant of f and g is the determinant of the $((m+n)\times(m+n))$-matrix

$$M = \begin{pmatrix}
a_0 & a_1 & a_2 & a_3 & \cdots & a_n & & & \\
 & a_0 & a_1 & a_2 & \cdots & a_{n-1} & a_n & & \\
 & & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\
 & & & a_0 & a_1 & \cdots & a_{n-2} & a_{n-1} & a_n \\
b_0 & b_1 & \cdots & b_{m-1} & b_m & & & & \\
 & b_0 & b_1 & \cdots & b_{m-1} & b_m & & & \\
 & & b_0 & b_1 & \cdots & b_{m-1} & b_m & & \\
 & & & \ddots & \ddots & \ddots & \ddots & \ddots & \\
 & & & & b_0 & b_1 & \cdots & b_{m-1} & b_m.
\end{pmatrix}. \tag{5.20}$$

The matrix above is usually referred to as the Sylvester matrix. The blank spaces in the Sylvester matrix represent zero entries.

The resultant provides information about the common factors of f and g. One can use it to express, particularly elegantly, the fact that two polynomials are relatively prime:

$$\gcd(f(x), g(x)) = 1 \Leftrightarrow Res(f, g) \neq 0\ . \tag{5.21}$$

Corollary 5.76 *Let $f(x) = a_0 + a_1x + \cdots + a_nx^n \in \mathbf{Z}[x]$ be a square-free (in $\mathbb{Q}[x]$), non-constant polynomial. Then $Res(f(x), f'(x))$ is an integer. Further, assume that p is a prime not dividing na_n. Then the polynomial $f(x) \pmod p$ is square-free in $\mathbb{F}_p[x]$ if and only if p does not divide $Res(f(x), f'(x))$.*

Proof The entries of the Sylvester matrix corresponding to $f(x)$ and $f'(x)$ are integers, and so is its determinant. The polynomial f has no multiple roots over $\mathbb{Q}$, and so, by Exercise 5.5-1, $\gcd(f(x), f'(x)) = 1$, which gives, using (5.21), that $Res(f(x), f'(x)) \neq 0$. Let $F(x)$ denote the polynomial f reduced modulo p. Then it follows from our assumptions that $Res(F(x), F'(x))$ is precisely the residue of $Res(f(x), f'(x))$ modulo p. By Exercise 5.5-1, the polynomial $F(x)$ is square-free precisely when $\gcd(F(x), F'(x)) = 1$, which is equivalent to $Res(F(x), F'(x)) \neq 0$. This amounts to saying that p does not divide the integer $Res(f(x), f'(x))$. ■

Corollary 5.77 *If $f(x) \in \mathbf{Z}[x]$ is a square-free polynomial with degree n, then there is a prime $p = O((n \lg n + 2n \lg \|f\|)^2)$ (that is, the absolute value of p is polynomial in the bit size of f) such that the polynomial $f(x) \pmod p$ is square-free in $\mathbb{F}_p[x]$.*

Proof By the Prime Number Theorem (Theorem 33.37), for large enough K, the product of the primes in the interval $[1, K]$ is at least $2^{(0.9K/\ln K)}$.

Set $K = ((n+1)\lg n + 2n \lg \|f\|)^2$. If K is large enough, then

$$p_1 \cdots p_l \geq 2^{(0.9K/\ln K)} > 2^{\sqrt{K}} \geq n^{n+1} \|f\|^{2n} \geq n^{n+1} \|f\|^{2n-1} |a_n| \tag{5.22}$$

where $p_1, \ldots, p_l$ are primes not larger than K, and a_n is the leading coefficient of f.

Let us suppose, for the primes $p_1, \ldots, p_l$, that $f(x) \pmod{p_i}$ is not square-free in $\mathbb{F}_{p_i}[x]$. Then the product $p_1 \cdots p_l$ divides $Res(f(x), f'(x)) \cdot na_n$, and so

$$p_1 \cdots p_l \leq |Res(f, f')| \cdot |na_n| \leq \|f\|^{n-1} \cdot \|f'\|^n \cdot |na_n| \leq n^{n+1} \|f\|^{2n-1} |a_n| \,.$$

(In the last two inequalities, we used the Hadamard inequality, and the fact that $\|f'(x)\| \leq n \|f(x)\|$.) This contradicts to inequality (5.22), which must be valid because of the choice of K. ∎

We note that using the Prime Number Theorem more carefully, one can obtain a stronger bound for p.

Hensel lifting We present a general procedure that can be used to obtain, given a factorisation modulo a prime p, a factorisation modulo p^N of a polynomial with integer coefficients.

Theorem 5.78 (Hensel's lemma). *Suppose that $f(x),\ g(x),\ h(x) \in \mathbf{Z}[x]$ are polynomials with leading coefficient 1 such that $f(x) \equiv g(x)h(x) \pmod{p}$, and, in addition, $g(x) \pmod{p}$ and $h(x) \pmod{p}$ are relatively prime in $\mathbb{F}_p[x]$. Then, for an arbitrary positive integer t, there are polynomials $g_t(x),\ h_t(x) \in \mathbf{Z}[x]$ such that*

- *both of the leading coefficients of $g_t(x)$ and $h_t(x)$ are equal to 1,*
- *$g_t(x) \equiv g(x) \pmod{p}$ and $h_t(x) \equiv h(x) \pmod{p}$,*
- *$f(x) \equiv g_t(x)h_t(x) \pmod{p^t}$.*

Moreover, the polynomials $g_t(x)$ and $h_t(x)$ satisfying the conditions above are unique modulo p^t.

Proof From the conditions concerning the leading coefficients, we obtain that $\deg f(x) = \deg g(x) + \deg h(x)$, and, further, that $\deg g_t(x) = \deg g(x)$ and $\deg h_t(x) = \deg h(x)$, provided the suitable polynomials $g_t(x)$ and $h_t(x)$ indeed exist. The existence is proved by induction on t. In the initial step, $t = 1$ and the choice $g_1(x) = g(x)$ and $h_1(x) = h(x)$ is as required.

The induction step $t \to t+1$: let us assume that there exist polynomials $g_t(x)$ and $h_t(x)$ that are well-defined modulo p^t and satisfy the conditions. If the polynomials $g_{t+1}(x)$ and $h_{t+1}(x)$ exist, then they must satisfy the conditions imposed on $g_t(x)$ and $h_t(x)$. As $g_t(x)$ and $h_t(x)$ are unique modulo p^t, we may write $g_{t+1}(x) = g_t(x) + p^t\delta_g(x)$ and $h_{t+1}(x) = h_t(x) + p^t\delta_h(x)$ where $\delta_g(x)$ and $\delta_h(x)$ are polynomials with integer coefficients. The condition concerning the leading coefficients guarantees that $\deg \delta_g(x) < \deg g(x)$ and that $\deg \delta_h(x) < \deg h(x)$.

By the induction hypothesis, $f(x) = g_t(x)h_t(x) + p^t\lambda(x)$ where $\lambda(x) \in \mathbf{Z}[x]$. The observations about the degrees of the polynomials $g_t(x)$ and $h_t(x)$ imply that the

degree of $\lambda(x)$ is smaller than $\deg f(x)$. Now we may compute that

$$\begin{aligned} g_{t+1}(x)h_{t+1}(x) - f(x) &= g_t(x)h_t(x) - f(x) + p^t h_t(x)\delta_g(x) + \\ &+ p^t g_t(x)\delta_h(x) + p^{2t}\delta_g(x)\delta_h(x) \\ &\equiv -p^t\lambda(x) + p^t h_t(x)\delta_g(x) + p^t g_t(x)\delta_h(x) \pmod{p^{2t}} . \end{aligned}$$

As $2t > t+1$, the congruence above holds modulo p^{t+1}. Thus $g_{t+1}(x)$ and $h_{t+1}(x)$ satisfy the conditions if and only if

$$p^t h_t(x)\delta_g(x) + p^t g_t(x)\delta_h(x) \equiv p^t\lambda(x) \pmod{p^{t+1}} .$$

This, however, amounts to saying, after cancelling p^t from both sides, that

$$h_t(x)\delta_g(x) + g_t(x)\delta_h(x) \equiv \lambda(x) \pmod{p} .$$

Using the congruences $g_t(x) \equiv g(x) \pmod{p}$ and $h_t(x) \equiv h(x) \pmod{p}$ we obtain that this is equivalent to the congruence

$$h(x)\delta_g(x) + g(x)\delta_h(x) \equiv \lambda(x) \pmod{p} . \tag{5.23}$$

Considering the inequalities $\deg\delta_g(x) < \deg g_t(x)$ and $\deg\delta_h(x) < \deg h_t(x)$ and the fact that in $\mathbb{F}_p[x]$ the polynomials $g(x) \pmod{p}$ and $h(x) \pmod{p}$ are relatively prime, we find that equation (5.23) can be solved uniquely in $\mathbb{F}_p[x]$. For, if $u(x)$ and $v(x)$ form a solution to $u(x)g(x) + v(x)h(x) \equiv 1 \pmod{p}$, then, by Theorem 5.12, the polynomials

$$\delta_g(x) = v(x)\lambda(x) \pmod{g(x)} ,$$

and

$$\delta_h(x) = u(x)\lambda(x) \pmod{h(x)}$$

form a solution of (5.23). The uniqueness of the solution follows from the bounds on the degrees, and from the fact that $g(x) \pmod{p}$ and $h(x) \pmod{p}$ relatively prime. The details of this are left to the reader. ∎

Corollary 5.79 *Assume that* p*, and the polynomials* $f(x)$*,* $g(x)$*,* $h(x) \in \mathbf{Z}[x]$ *satisfy the conditions of Hensel's lemma. Set* $\deg f = n$ *and let* N *be a positive integer. Then the polynomials* $g_N(x)$ *and* $h_N(x)$ *can be obtained using* $O(Nn^2)$ *arithmetic operations modulo* p^N*.*

Proof The proof of Theorem 5.78 suggests the following algorithm.

HENSEL-LIFTING(f, g, h, p, N)

```
1  (u(x), v(x)) ← is a solution, in F_p[x], of u(x)g(x) + v(x)h(x) ≡ 1 (mod p)
2  (G(x), H(x)) ← (g(x), h(x))
3  for t ← 1 to N − 1
4       do λ(x) ← (f(x) − G(x) · H(x))/p^t
5          δ_g(x) ← v(x)λ(x) reduced modulo g(x) (in F_p[x])
6          δ_h(x) ← u(x)λ(x) reduced modulo h(x) (in F_p[x])
7          (G(x), H(x)) ← (G(x) + p^t δ_g(x), H(x) + p^t δ_h(x)) (in (Z/(p^{t+1}))[x])
8  return (G(x), H(x))
```

The polynomials u and v can be obtained using $O(n^2)$ operations in $\mathbb{F}_p$ (see Theorem 5.12 and the remark following it). An iteration $t \to t+1$ consists of a constant number of operations with polynomials, and the cost of one run of the main loop is $O(n^2)$ operations (modulo p and p^{t+1}). The total cost of reaching $t = N$ is $O(Nn^2)$ operations. ∎

5.5.2. The Berlekamp-Zassenhaus algorithm

The factorisation problem (5.18) was efficiently reduced to the case in which the polynomial f has integer coefficients and leading coefficient 1. We may also assume that $f(x)$ has no multiple factors in $\mathbb{Q}[x]$. Indeed, in our case $f'(x) \neq 0$, and so the possible multiple factors of f can be separated using the idea that we already used over finite fields as follows. By Lemma 5.13, the polynomial $g(x) = f(x)/(f(x), f'(x))$ is already square-free, and, using Lemma 5.14, it suffices to find its factors with multiplicity one. From Proposition 5.71, we can see that $g(x)$ has integer coefficients and leading coefficient 1. Computing the greatest common divisor and dividing polynomials can be performed efficiently, and so the reduction can be carried out in polynomial time. (In the computation of the greatest common divisor, the intermediate expression swell can be avoided using the techniques used in number theory.)

In the sequel we assume that the polynomial

$$f(x) = x^n + \sum_{i=0}^{n-1} a_i x^i \in \mathbf{Z}[x]$$

we want to factor is square-free, its coefficients are integers, and its leading coefficient is 1.

The fundamental idea of the Berlekamp-Zassenhaus algorithm is that we compute the irreducible factors of $f(x)$ modulo p^N where p is a suitably chosen prime and N is large enough. If, for instance, $p^N > 2 \cdot 2^{n-1} \|f\|$, and we have already computed the coefficients of a factor modulo p^N, then, by Mignotte's theorem, we can obtain the coefficients of a factor in $\mathbb{Q}[x]$.

From now on, we will also assume that p is a prime such that the polynomial $f(x) \pmod p$ is square-free in $\mathbb{F}_p[x]$. Using linear search such a prime p can be found in polynomial time (Corollary 5.77). One can even assume that p is polynomial in the bit size of $f(x)$.

The irreducible factors in $\mathbb{F}_p[x]$ of the polynomial $f(x) \pmod p$ can be found using Berlekamp's deterministic method (Theorem 5.42). Let $g_1(x), \ldots, g_r(x) \in \mathbf{Z}[x]$ be polynomials, all with leading coefficient 1, such that the $g_i(x) \pmod p$ are the irreducible factors of the polynomial $f(x) \pmod p$ in $\mathbb{F}_p[x]$.

Using the technique of Hensel's lemma (Theorem 5.78) and Corollary 5.79, the system $g_1(x), \ldots, g_r(x)$ can be lifted modulo p^N. To simplify the notation, we assume now that $g_1(x), \ldots, g_r(x) \in \mathbf{Z}[x]$ are polynomials with leading coefficients 1 such that

$$f(x) \equiv g_1(x) \cdots g_r(x) \pmod{p^N}$$

and the $g_i(x) \pmod p$ are the irreducible factors of the polynomial $f(x) \pmod p$ in $\mathbb{F}_p[x]$.

Let $h(x) \in \mathbf{Z}[x]$ be an irreducible factor with leading coefficient 1 of the polynomial $f(x)$ in $\mathbb{Q}[x]$. Then there is a uniquely determined set $I \subseteq \{1, \ldots, r\}$ for which

$$h(x) \equiv \prod_{i \in I} g_i(x) \pmod{p^N} .$$

Let N be the smallest integer such that $p^N \geq 2 \cdot 2^{n-1} \|f(x)\|$. Mignotte's bound shows that the polynomial $\prod_{i \in I} g_i(x) \pmod{p^N}$ on the right-hand side, if its coefficients are represented by the residues with the smallest absolute values, coincides with h.

We found that determining the irreducible factors of $f(x)$ is equivalent to finding minimal subsets $I \subseteq \{1, \ldots, r\}$ for which there is a polynomial $h(x) \in \mathbf{Z}[x]$ with leading coefficient 1 such that $h(x) \equiv \prod_{i \in I} g_i(x) \pmod{p^N}$, the absolute values of the coefficients of $h(x)$ are at most $2^{n-1} \|f(x)\|$, and, moreover, $h(x)$ divides $f(x)$. This can be checked by examining at most 2^{r-1} sets I. The cost of examining a single I is polynomial in the size of f.

To summarise, we obtained the following method to factor, in $\mathbb{Q}[x]$, a square-free polynomial $f(x)$ with integer coefficients and leading coefficient 1.

BERLEKAMP-ZASSENHAUS(f)

1 $p \leftarrow$ a prime p such that $f(x) \pmod{p}$ is square-free in $\mathbb{F}_p[x]$
  and $p = O((n \lg n + 2n \lg \|f\|)^2)$
2 $\{g_1, \ldots, g_r\} \leftarrow$ the irreducible factors of $f(x) \pmod{p}$ in $\mathbb{F}_p[x]$
  (using Berlekamp's deterministic method)
3 $N \leftarrow \lfloor \log_p(2^{\deg f} \cdot \|f\|) \rfloor + 1$
4 $\{g_1, \ldots, g_r\} \leftarrow$ the Hensel lifting of the system $\{g_1, \ldots, g_r\}$ modulo p^N
5 $\mathcal{I} \leftarrow$ the collection of minimal subsets $I \neq \emptyset$ of $\{1, \ldots r\}$ such that
  $g_I \leftarrow \prod_{i \in I} g_i$ reduced modulo p^N divides f
6 **return** $\{\prod_{i \in I} g_i : I \in \mathcal{I}\}$

Theorem 5.80 *Let $f(x) = x^n + \sum_{i=0}^{n-1} a_i x^i \in \mathbf{Z}[x]$ be a square-free polynomial with integer coefficients and leading coefficient 1, and let p be a prime number such that the polynomial $f(x) \pmod{p}$ is square-free in $\mathbb{F}_p[x]$ and $p = O((n \lg n + 2n \lg \|f\|)^2)$. Then the irreducible factors of the polynomial f in $\mathbb{Q}[x]$ can be obtained by the Berlekamp-Zassenhaus algorithm. The cost of this algorithm is polynomial in n, $\lg \|f(x)\|$ and 2^r where r is the number of irreducible factors of the polynomial $f(x) \pmod{p}$ in $\mathbb{F}_p[x]$.*

Example 5.5 (Swinnerton-Dyer polynomials) Let

$$f(x) = \prod (x \pm \sqrt{2} \pm \sqrt{3} \pm \cdots \pm \sqrt{p_l}) \in \mathbf{Z}[x] ,$$

where $2, 3, \ldots, p_l$ are the first l prime numbers, and the product is taken over all possible 2^l combinations of the signs $+$ and $-$. The degree of $f(x)$ is $n = 2^l$, and one can show that it is irreducible in $\mathbb{Q}[x]$. On the other hand, for all primes p, the polynomial $f(x) \pmod{p}$ is the product of factors with degree at most 2. Therefore these polynomials represent hard

cases for the Berlekamp-Zassenhaus algorithm, as we need to examine about $2^{n/2-1}$ sets I to find out that f is irreducible.

5.5.3. The LLL algorithm

Our goal in this section is to present the Lenstra-Lenstra-Lovász algorithm (LLL algorithm) for factoring polynomials $f(x) \in \mathbb{Q}[x]$. This was the first polynomial time method for solving the polynomial factorisation problem over $\mathbb{Q}$. Similarly to the Berlekamp-Zassenhaus method, the LLL algorithm starts with a factorisation of f modulo p and then uses Hensel lifting. In the final stages of the work, it uses lattice reduction to find a proper divisor of f, provided one exists. The powerful idea of the LLL algorithm is that it replaced the search, which may have exponential complexity, in the Berlekamp-Zassenhaus algorithm by an efficient lattice reduction.

Let $f(x) \in \mathbf{Z}[x]$ be a square-free polynomial with leading coefficient 1 such that $\deg f = n > 1$, and let p be a prime such that the polynomial $f(x) \pmod p$ is square free in $\mathbb{F}_p[x]$ and $p = O((\lg n + 2n \lg \|f\|)^2)$.

Lemma 5.81 *Suppose that $f(x) \equiv g_0(x)v(x) \pmod{p^N}$ where $g_0(x)$ and $v(x)$ are polynomials with integer coefficients and leading coefficient 1. Let $g(x) \in \mathbf{Z}[x]$ with $\deg g(x) = m < n$ and assume that $g(x) \equiv g_0(x)u(x) \pmod{p^N}$ for some polynomial $u(x)$ such that $u(x)$ has integer coefficients and $\deg u(x) = \deg g(x) - \deg g_0(x)$. Let us further assume that $\|g(x)\|^n \, \|f(x)\|^m < p^N$. Then $\gcd(f(x), g(x)) \neq 1$ in $\mathbb{Q}[x]$.*

Proof Let $d = \deg v(x)$. By the assumptions,

$$f(x)u(x) \equiv g_0(x)u(x)v(x) \equiv g(x)v(x) \pmod{p^N} .$$

Suppose that $u(x) = \alpha_0 + \alpha_1 x + \ldots + \alpha_{m-1}x^{m-1}$ and $v(x) = \beta_0 + \beta_1 x + \ldots + \beta_{n-1}x^{n-1}$. (We know that $\beta_d = 1$. If $i > d$, then $\beta_i = 0$, and similarly, if $j > \deg u(x)$, then $\alpha_j = 0$.) Rewriting the congruence, we obtain

$$x^d g(x) + \sum_{j \neq d} \beta_j x^j g(x) - \sum_i \alpha_i x^i f(x) \equiv 0 \pmod{p^N} .$$

Considering the coefficient vectors of the polynomials $x^j g(x)$ and $x^i f(x)$, this congruence amounts to saying that adding to the $(m+d)$-th row of the Sylvester matrix (5.20) a suitable linear combination of the other rows results in a row in which all the elements are divisible by p^N. Consequently, $\det M \equiv 0 \pmod{p^N}$. The Hadamard inequality (Corollary 5.60) yields that $|\det M| \leq \|f\|^m \, \|g\|^n < p^N$, but this can only happen if $\det M = 0$. However, $\det M = Res(f(x), g(x))$, and so, by (5.21), $\gcd(f(x), g(x)) \neq 1$. ∎

The application of lattice reduction

Set

$$N = \lceil \log_p(2^{2n^2} \, \|f(x)\|^{2n}) \rceil = O(n^2 + n \lg \|f(x)\|) .$$

Further, we let $g_0(x) \in \mathbf{Z}[x]$ be a polynomial with leading coefficient 1 such that $g_0(x) \pmod{p^N}$ is an irreducible factor of $f(x) \pmod{p^N}$. Set $d = \deg g_0(x) < n$.

Define the set L as follows:

$$L = \{g(x) \in \mathbf{Z}[x] : \ \deg g(x) \leq n-1, \ \exists h(x) \in \mathbf{Z}[x], \text{ with } g \equiv hg_0 \pmod{p^N}\} \ . \tag{5.24}$$

Clearly, L is closed under addition of polynomials. We identify a polynomial with degree less than n with its coefficient vector of length n. Under this identification, L becomes a lattice in $\mathbb{R}^n$. Indeed, it is not too hard to show (Exercise 5.5-2) that the polynomials

$$p^N 1, p^N x, \ldots, p^N x^{d-1}, \ g_0(x), xg_0(x), \ldots, x^{n-d-1}g_0(x) \ ,$$

or, more precisely, their coefficient vectors, form a basis of L.

Theorem 5.82 *Let $g_1(x) \in \mathbf{Z}[x]$ be a polynomial with degree less than n such that the coefficient vector of $g_1(x)$ is the first element in a Lovász-reduced basis of L. Then $f(x)$ is irreducible in $\mathbb{Q}[x]$ if and only if* $\gcd(f(x), g_1(x)) = 1$.

Proof As $g_1(x) \neq 0$, it is clear that $\gcd(f(x), g_1(x)) = 1$ whenever $f(x)$ is irreducible. In order to show the implication in the other direction, let us assume that $f(x)$ is reducible and let $g(x)$ be a proper divisor of $f(x)$ such that $g(x) \pmod p$ is divisible by $g_0(x) \pmod p$ in $\mathbb{F}_p[x]$. Using Hensel's lemma (Theorem 5.78), we conclude that $g(x) \pmod{p^N}$ is divisible by $g_0(x) \pmod{p^N}$, that is, $g(x) \in L$. Mignotte's theorem (Theorem 5.74) shows that

$$\|g(x)\| \leq 2^{n-1} \|f(x)\| \ .$$

Now, if we use the properties of reduced bases (second assertion of Theorem 5.67), then we obtain

$$\|g_1(x)\| \leq 2^{(n-1)/2} \|g(x)\| < 2^n \|g(x)\| \leq 2^{2n} \|f(x)\| \ ,$$

and so

$$\|g_1(x)\|^n \|f(x)\|^{\deg g_1} \leq \|g_1(x)\|^n \|f(x)\|^n < 2^{2n^2} \|f(x)\|^{2n} \leq p^N \ .$$

We can hence apply Lemma 5.81, which gives $\gcd(g_1(x), f(x)) \neq 1$. ∎

Based on the previous theorem, the LLL algorithm can be outlined as follows (we only give a version for factoring to two factors). The input is a square-free polynomial $f(x) \in \mathbf{Z}[x]$ with integer coefficients and leading coefficient 1 such that $\deg f = n > 1$.

LLL-POLYNOMIAL-FACTORISATION(f)

1 $p \leftarrow$ a prime p such that $f(x) \pmod p$ is square-free in $\mathbb{F}_p[x]$
  and $p = O((n \lg n + 2n \lg \|f\|)^2)$
2 $w(x) \leftarrow$ an irreducible factor $f(x) \pmod p$ in $\mathbb{F}_p[x]$
  (using Berlekamp's deterministic method)
3 **if** $\deg w = n$
4 **then return** "irreducible"
5 **else** $N \leftarrow \lceil \log_p((2^{2n^2} \|f(x)\|^{2n})) \rceil = O(n^2 + n \lg(\|f(x)\|)$
6 $(g_0, h_0) \leftarrow$ HENSEL-LIFTING$(f, w, f/w \pmod p, p, N)$
7 $(b_1, \ldots, b_n) \leftarrow$ a basis of the lattice $L \subseteq \mathbb{R}^n$ in (5.24)
8 $(g_1, \ldots, g_n) \leftarrow$ LOVÁSZ-REDUCTION$(b_1, \ldots, b_n)$
9 $f^* \leftarrow \gcd(f, g_1)$
10 **if** $\deg f^* > 0$
11 **then return** $(f^*, f/f^*)$
12 **else return** "irreducible"

Theorem 5.83 *Using the LLL algorithm, the irreducible factors in $\mathbb{Q}[x]$ of a polynomial $f \in \mathbb{Q}[x]$ can be obtained deterministically in polynomial time.*

Proof The general factorisation problem, using the method introduced at the discussion of the Berlekamp-Zassenhaus procedure, can be reduced to the case in which the polynomial $f(x) \in \mathbf{Z}[x]$ is square-free and has leading coefficient 1. By the observations made there, the steps in lines 1–7 can be performed in polynomial time. In line 8, the Lovász reduction can be carried out efficiently (Corollary 5.65). In line 9, we may use a modular version of the Euclidean algorithm to avoid intermediate expression swell (see Chapter **??**).

The correctness of the method is asserted by Theorem 5.82. The LLL algorithm can be applied repeatedly to factor the polynomials in the output, in case they are not already irreducible. ■

One can show that the Hensel lifting costs $O(Nn^2) = O(n^4 + n^3 \lg \|f\|)$ operations with moderately sized integers. The total cost of the version of the LLL algorithm above is $O(n^5 \lg(p^N)) = O(n^7 + n^6 \lg \|f\|)$.

Exercises

5.5-1 Let $\mathbb{F}$ be a field and let $0 \neq f(x) \in \mathbb{F}[x]$. The polynomial $f(x)$ has no irreducible factors with multiplicity greater than one if and only if $\gcd(f(x), f'(x)) = 1$. *Hint.* In one direction, one can use Lemma 5.13, and use Lemma 5.14 in the other.

5.5-2 Show that the polynomials

$$p^N 1, p^N x, \ldots, p^N x^{d-1}, g_0(x), x g_0(x), \ldots, x^{n-d-1} g_0(x)$$

form a basis of the lattice in (5.24). *Hint.* It suffices to show that the polynomials $p^N x^j$ $(d \leq j < n)$ can be expressed with the given polynomials. To show this, divide $p^N x^j$ by $g_0(x)$ and compute the remainder.

Problems

5-1 The trace in finite fields

Let $\mathbb{F}_{q^k} \supseteq \mathbb{F}_q$ be finite fields. The definition of the trace map $tr = tr_{k,q}$ on $\mathbb{F}_{q^k}$ is as follows: if $\alpha \in \mathbb{F}_{q^k}$ then

$$tr(\alpha) = \alpha + \alpha^q + \cdots + \alpha^{q^{k-1}} .$$

a. Show that the map tr is $\mathbb{F}_q$-linear and its image is precisely $\mathbb{F}_q$. *Hint.* Use the fact that tr is defined using a polynomial with degree q^{k-1} to show that tr is not identically zero.

b. Let (α, β) be a uniformly distributed random pair of elements from $\mathbb{F}_{q^k} \times \mathbb{F}_{q^k}$. Then the probability that $tr(\alpha) \neq tr(\beta)$ is $1 - 1/q$.

5-2 The Cantor-Zassenhaus algorithm for fields of characteristic 2

Let $\mathbb{F} = \mathbb{F}_{2^m}$ and let $f(x) \in \mathbb{F}[x]$ be a polynomial of the form

$$f = f_1 f_2 \cdots f_s, \tag{5.25}$$

where the f_i are pairwise relatively prime and irreducible polynomials with degree d in $\mathbb{F}[x]$. Also assume that $s \geq 2$.

a. Let $u(x) \in \mathbb{F}[x]$ be a uniformly distributed random polynomial with degree less than $\deg f$. Then the greatest common divisor

$$\gcd(u(x) + u^2(x) + \cdots + u^{2^{md-1}}(x), f(x))$$

is a proper divisor of $f(x)$ with probability at least $1/2$.

Hint. Apply the previous exercise taking $q = 2$ and $k = md$, and follow the argument in Theorem 5.38.

b. Using part (a), give a randomised polynomial time method for factoring a polynomial of the form (5.25) over $\mathbb{F}$.

5-3 Divisors and zero divisors

Let $\mathbb{F}$ be a field. The ring R is said to be an $\mathbb{F}$-***algebra*** (in case $\mathbb{F}$ is clear from the context, R is simply called an algebra), if R is a vector space over $\mathbb{F}$, and $(ar)s = a(rs) = r(as)$ holds for all $r, s \in S$ and $a \in \mathbb{F}$. It is easy to see that the rings $\mathbb{F}[x]$ and $\mathbb{F}[x]/(f)$ are $\mathbb{F}$-algebras.

Let R be a finite-dimensional $\mathbb{F}$-algebra. For an arbitrary $r \in R$, we may consider the map $L_r : R \to R$ defined as $L_r(s) = rs$ for $s \in R$. The map L_r is $\mathbb{F}$-linear, and so we may speak about its minimal polynomial $m_r(x) \in \mathbb{F}[x]$, its characteristic polynomial $k_r(x) \in \mathbb{F}[x]$, and its trace $Tr(r) = Tr(L_r)$. In fact, if U is an ideal in R, then U is an invariant subspace of L_r, and so we can restrict L_r to U, and we may consider the minimal polynomial, the characteristic polynomial, and the trace of the restriction.

a. Let $f(x),\ g(x) \in \mathbb{F}[x]$ with $\deg f > 0$. Show that the residue class $[g(x)]$ is a zero divisor in the ring $\mathbb{F}[x]/(f)$ if and only if f does not divide g and $\gcd(f(x), g(x)) \neq 1$.

b. Let R be an algebra over $\mathbb{F}$, and let $r \in R$ be an element with minimal polynomial $f(x)$. Show that if f is not irreducible over $\mathbb{F}$, then R contains a zero divisor. To be precise, if $f(x) = g(x)h(x)$ is a non-trivial factorisation ($g, h \in \mathbb{F}[x]$), then $g(r)$ and $h(r)$ form a pair of zero divisors, that is, both of them are non-zero, but their product is zero.

5-4 Factoring polynomials over algebraic number fields

a. Let $\mathbb{F}$ be a field with characteristic zero and let R be a finite-dimensional $\mathbb{F}$-algebra with an identity element. Let us assume that $R = S_1 \oplus S_2$ where S_1 and S_2 are non-zero $\mathbb{F}$-algebras. Let $r_1, \ldots, r_k$ be a basis of R over $\mathbb{F}$. Show that there is a j such that $m_{r_j}(x)$ is not irreducible in $\mathbb{F}[x]$.

Hint. This exercise is for readers who are familiar with the elements of linear algebra. Let us assume that the minimal polynomial of r_j is the irreducible polynomial $m(x) = x^d - a_1 x^{d-1} + \cdots + a_d$. Let $k_i(x)$ be the characteristic polynomial of L_{r_j} on the invariant subspace U_i (for $i \in \{1, 2\}$). Here U_1 and U_2 are the sets of elements of the form $(s_1, 0)$ and $(0, s_2)$, respectively where $s_i \in S_i$. Because of our conditions, we can find suitable exponents d_i such that $k_i(x) = m(x)^{d_i}$. This implies that the trace $T_i(r_j)$ of the map L_{r_j} on the subspace U_i is $T_i(r_j) = d_i a_1$. Set $e_i = \dim_{\mathbb{F}} U_i$. Obviously, $e_i = d_i d$, which gives $T_1(r_j)/e_1 = T_2(r_j)/e_2$. If the assertion of the exercise is false, then the latter equation holds for all j, and so, as the trace is linear, it holds for all $r \in R$. This, however, leads to a contradiction: if $r = (1, 0) \in S_1 \oplus S_2$ (1 denotes the unity in S_1), then clearly $T_1(r) = e_1$ and $T_2(r) = 0$.

b. Let $\mathbb{F}$ be an ***algebraic number field***, that is, a field of the form $\mathbb{Q}(\alpha)$ where $\alpha \in \mathbb{C}$, and there is an irreducible polynomial $g(x) \in \mathbf{Z}[x]$ such that $g(\alpha) = 0$. Let $f(x) \in \mathbb{F}[x]$ be a square-free polynomial and set $R = \mathbb{F}[x]/(f)$. Show that R is a finite-dimensional algebra over $\mathbb{Q}$. More precisely, if $\deg g = m$ and $\deg f = n$, then the elements of the form $\alpha^i [x]^j$ $(0 \leq i < m,\ 0 \leq j < n)$ form a basis over $\mathbb{Q}$.

c. Show that if f is reducible over $\mathbb{F}$, then there are $\mathbb{Q}$-algebras $S_1,\ S_2$ such that $R \cong S_1 \oplus S_2$.

Hint. Use the Chinese remainder theorem .

d. Consider the polynomial g above and suppose that a field $\mathbb{F}$ and a polynomial $f \in \mathbb{F}[x]$ are given. Assume, further, that f is square-free and is not irreducible over $\mathbb{F}$. The polynomial f can be factored to the product of two non-constant polynomials in polynomial time.

Hint. By the previous remarks, the minimal polynomial $m(y)$ over $\mathbb{Q}$ of at least one of the elements $\alpha^i [x]^j$ $(0 \leq i \leq m,\ 0 \leq j \leq n)$ is not irreducible in $\mathbb{Q}[y]$. Using the LLL algorithm, $m(y)$ can be factored efficiently in $\mathbb{Q}[y]$. From a factorisation of $m(y)$, a zero divisor of R can be obtained, and this can be used to find a

proper divisor of f in $\mathbb{F}[x]$.

Chapter notes

The abstract algebraic concepts discussed in this chapter can be found in many textbooks; see, for instance, Hungerford's book [114].

The theory of finite fields and the related algorithms are the theme of the excellent books by Lidl and Niederreiter [151] and Shparlinski [224].

Our main algorithmic topics, namely the factorisation of polynomials and lattice reduction are thoroughly treated in the book by von zur Gathen and Gerhard [77]. We recommend the same book to the readers who are interested in the efficient methods to solve the basic problems concerning polynomials. Theorem 8.23 of that book estimates the cost of multiplying polynomials by the Schönhage-Strassen method, while Corollary 11.6 is concerned with the cost of the asymptotically fast implementation of the Euclidean algorithm. Ajtai's result about shortest lattice vectors was published in [7].

The method by Kaltofen and Shoup is a randomised algorithm for factoring polynomials over finite fields, and currently it has one of the best time bounds among the known algorithms. The expected number of $\mathbb{F}_q$-operations in this algorithm is $O(n^{1.815} \lg q)$ where $n = \deg f$. Further competitive methods were suggested by von zur Gathen and Shoup, and also by Huang and Pan. The number of operations required by the latter is $O(n^{1.80535} \lg q)$, if $\lg q < n^{0.00173}$. Among the deterministic methods, the one by von zur Gathen and Shoup is the current champion. Its cost is $\widetilde{O}(n^2 + n^{3/2}s + n^{3/2}s^{1/2}p^{1/2})$ operations in $\mathbb{F}_q$ where $q = p^s$. An important related problem is constructing the field $\mathbb{F}_{q^n}$. The fastest randomised method is by Shoup. Its cost is $O\sim(n^2 + n \lg q)$. For finding a square-free factorisation, Yun gave an algorithm that requires $\widetilde{O}(n) + O(n \lg(q/p))$ field operations in $\mathbb{F}_q$.

The best methods to solve the problem of lattice reduction and that of factoring polynomials over the rationals use modular and numerical techniques. After slightly modifying the definition of reduced bases, an algorithm using $\widetilde{O}(n^{3.381} \lg^2 C)$ *bit operations* for the former problem was presented by Storjohann. (We use the original definition introduced in the paper by Lenstra, Lenstra and Lovász [149].) We also mention Schönhage's method using $\widetilde{O}(n^6 + n^4 \lg^2 l)$ bit operations for factoring polynomials with integer coefficients (l is the length of the coefficients).

Besides factoring polynomials with rational coefficients, lattice reduction can also be used to solve lots of other problems: to break knapsack cryptosystems and random number generators based on linear congruences, simultaneous Diophantine approximation, to find integer linear dependencies among real numbers (this problem plays an important rôle in experiments that attempt to find mathematical identities). These and other related problems are discussed in the book [77].

A further exciting application area is the numerical solution of Diophantine equations. One can read about these developments in in the books by Smart [230] and Gaál [73]. The difficulty of finding a shortest lattice vector was verified in Ajtai's paper [7].

Finally we remark that the practical implementations of the polynomial methods involving lattice reduction are not competitive with the implementations of the Berlekamp-Zassenhaus algorithm, which, in the worst case, has exponential complexity. Nevertheless, the basis reduction performs very well in practice: in fact it is usually much faster than its theoretically proven speed. For some of the problems in the application areas listed above, we do not have another useful method.

The work of the authors was supported in part by grants T042481 and T042706 of the Hungarian Scientific Research Fund.

6. Computer Algebra

Computer systems performing various mathematical computations are inevitable in modern science and technology. We are able to compute the orbits of planets and stars, command nuclear reactors, describe and model many of the natural forces. These computations can be *numerical and symbolical.*
Although *numerical computations* may involve not only elementary arithmetical operations (addition, subtraction, multiplication, division) but also more sophisticated calculations, like computing numerical values of mathematical functions, finding roots of polynomials or computing numerical eigenvalues of matrices, these operations can only be carried out *on numbers.* Furthermore, in most cases these numbers are not exact. Their degree of precision depends on the floating-point arithmetic of the given computer hardware architecture.

Unlike numerical calculations, *symbolic and algebraic computations* operate on *symbols* that represent mathematical objects. These objects may be numbers such as integers, rational numbers, real and complex numbers, but may also be polynomials, rational and trigonometric functions, equations, algebraic structures such as groups, rings, ideals, algebras or elements of them, or even sets, lists, tables.

Computer systems with the ability to handle symbolic computations are called *computer algebra systems* or *symbolic and algebraic systems* or *formula manipulation systems.* In most cases, these systems are able to handle both numerical and graphical computations. The word „symbolic" emphasises that, during the problem-solving procedure, the objects are represented by symbols, and the adjective „algebraic" refers to the algebraic origin of the operations on these symbolic objects.

To characterise the notion „computer algebra", one can describe it as a collection of *computer programs* developed basically to perform

- exact representations of mathematical objects and
- arithmetic with these objects.

On the other hand, computer algebra can be viewed as a *discipline* which has been developed in order to invent, analyse and implement efficient mathematical algorithms based on exact arithmetic for scientific research and applications.

Since computer algebra systems are able to perform error-free computations with arbitrary precision, first we have to clarify the data structures assigned to the various objects. Subsection 6.1 deals with the problems of representing mathematical objects. Furthermore, we describe the symbolic algorithms which are indispensable

in modern science and practice.

The problems of natural sciences are mainly expressed in terms of mathematical equations. Research in solving symbolic linear systems is based on the well-known elimination methods. To find the solutions of non-linear systems, first we analyse different versions of the *Euclidean algorithm* and the method of *resultants*. In the mid-sixties of the last century, Bruno Buchberger presented a method in his PhD thesis for solving multivariate polynomial equations of arbitrary degree. This method is known as the *Gröbner basis method*. At that time , the mathematical community paid little attention to his work, but since then it became the basis of a powerful set of tools for computing with higher degree polynomial equations. This topic is discussed in Subsections 6.2 and 6.3.

The next area to be introduced is the field of *symbolic integration*. Although the nature of the problem was understood long ago (Liouville's principle), it was only in 1969 that Robert Risch invented an algorithm to solve the following: given an elementary function $f(x)$ of a real variable x, decide whether the indefinite integral $\int f(x)dx$ is also an elementary function, and if so, compute the integral. We describe the method in Subsection 6.4.

At the end of this section, we offer a brief survey of the theoretical and practical relations of symbolic algorithms in Subsection 6.5, devoting an independent part to the present computer algebra systems.

6.1. Data representation

In computer algebra, one encounters mathematical objects of different kinds. In order to be able to manipulate these objects on a computer, one first has to represent and store them in the memory of that computer. This can cause several theoretical and practical difficulties. We examine these questions in this subsection.

Consider the *integers*. We know from our studies that the set of integers is countable, but computers can only store finitely many of them. The range of values for such a *single-precision integer* is limited by the number of distinct encodings that can be made in the computer word, which is typically 32 or 64 bits in length. Hence, one cannot directly use the computer's integers to represent the mathematical integers, but must be prepared to write programs to handle „arbitrarily" large integers represented by several computer integers. The term arbitrarily large does not mean infinitely large since some architectural constraints or the memory size limits in any case. Moreover, one has to construct data structures over which efficient operations can be built. In fact, there are two standard ways of performing such a representation.

- Radix notation (a generalisation of conventional decimal notation), in which n is represented as $\sum_{i=0}^{k-1} d_i B^i$, where the digits d_i $(0 \leq i \leq k-1)$ are single precision integers. These integers can be chosen from the *canonical digit set* $\{0 \leq d_i \leq B-1\}$ or from the *symmetrical digit set* $\{-\lfloor B/2 \rfloor < d_i \leq \lfloor B/2 \rfloor\}$, where base B can be, in principle, any positive integer greater than 1. For efficiency, B is chosen so that $B-1$ is representable in a single computer word. The length k of

the linear list $(d_0, d_1, \ldots, d_{k-1})$ used to represent a *multiprecision integer* may be dynamic (i.e. chosen approximately for the particular integer being represented) or static (i.e. pre-specified fixed length), depending on whether the linear list is implemented using linked list allocation or using array (sequential) notation. The sign of n is stored within the list, possibly as the sign of d_0 or one or more of the other entries.

- Modular notation, in which n is represented by its value modulo a sufficient number of large (but representable in one computer word) primes. From the images one can reconstruct n using the Chinese remainder algorithm.

The modular form is fast for addition, subtraction and multiplication but is much slower for divisibility tasks. Hence, the choice of representation influences the algorithms that will be chosen. Indeed, not only the choice of representation influences the algorithms to be used but also the algorithms influence the choice of representation.

Example 6.1 For the sake of simplicity, in the next example we work only with natural numbers. Suppose that we have a computer architecture with machine word 32 bits in length, i.e. our computer is able to perform integer arithmetic with the integers in range $I_1 = [0, 2^{32} - 1] = [0, 4\,294\,967\,295]$. Using this arithmetic, we carry out a new arithmetic by which we are able to perform integer arithmetic with the integers in range $I_2 = [0, 10^{50}]$.

Using radix representation let $B = 10^4$, and let

$$\begin{aligned} n_1 &= 12345678909876543210123456789 0 , \\ n_2 &= 2110 . \end{aligned}$$

Then,

$$\begin{aligned} n_1 &= [7890, 3456, 1012, 5432, 9876, 7890, 3456, 12] , \\ n_2 &= [2110] , \\ n_1 + n_2 &= [0, 3457, 1012, 5432, 9876, 7890, 3456, 12] , \\ n_1 \cdot n_2 &= [7900, 3824, 6049, 1733, 9506, 9983, 3824, 6049, 2] , \end{aligned}$$

where the sum and the product were computed using radix notation.

Switching to modular representation we have to choose pairwise relatively prime integers from the interval I_1 such that their product is greater than 10^{50}. Let, for example, the primes be

$$\begin{aligned} m_1 = 4294967291,\ m_2 = 4294967279,\ m_3 = 4294967231 , \\ m_4 = 4294967197,\ m_5 = 4294967189,\ m_6 = 4294967161 , \end{aligned}$$

where $\prod_{i=1}^{6} m_i > 10^{50}$. Then, an integer from the interval I_2 can be represented by a 6-tuple from the interval I_1. Therefore,

$$\begin{aligned} n_1 \equiv 2009436698 \pmod{m_1}, \quad n_1 \equiv 961831343 \pmod{m_2} , \\ n_1 \equiv 4253639097 \pmod{m_3}, \quad n_1 \equiv 1549708 \pmod{m_4} , \\ n_1 \equiv 2459482973 \pmod{m_5}, \quad n_1 \equiv 3373507250 \pmod{m_6} , \end{aligned}$$

furthermore, $n_2 \equiv 2110 \pmod{m_i}$, $(1 \leq i \leq 6)$. Hence

$$\begin{aligned} n_1 + n_2 &= [2009438808, 961833453, 4253641207, 1551818, 2459485083, 3373509360] , \\ n_1 \cdot n_2 &= [778716563, 2239578042, 2991949111, 3269883880, 1188708718, 1339711723] , \end{aligned}$$

where addition and multiplication were carried out using modular arithmetic.

More generally, concerning the choice of representation of other mathematical objects, it is worth distinguishing three levels of abstraction:

1. *Object level.* This is the level where the objects are considered as formal mathematical objects. For example $3+3$, $4\cdot 4-10$ and 6 are all representations of the integer 6. On the object level, the polynomials $(x-1)^2(x+1)$ and x^3-x^2-x+1 are considered equal.

2. *Form level.* On this level, one has to distinguish between different representations of an object. For example $(x-1)^2(x+1)$ and x^3-x^2-x+1 are considered different representations of the same polynomial, namely the former is a product, a latter is a sum.

3. *Data structure level.* On this level, one has to consider different ways of representing an object in a computer memory. For example, we distinguish between representations of the polynomial x^3-x^2-x+1 as

   - an array $[1,-1,-1,1]$,
   - a linked list $[1,0]\to[-1,1]\to[-1,2]\to[1,3]$.

In order to represent objects in a computer algebra system, one has to make choices on both the form and the data structure level. Clearly, various representations are possible for many objects. The problem of „how to represent an object" becomes even more difficult when one takes into consideration other criteria, such as memory space, computation time, or readability. Let us see an example. For the polynomial

$$\begin{aligned} f(x) &= (x-1)^2(x+1)^3(2x+3)^4 \\ &= 16x^9-80x^8+88x^7+160x^6-359x^5+x^4+390x^3-162x^2-135x+81 \end{aligned}$$

the product form is more comprehensive, but the second one is more suitable to know the coefficient of, say, x^5. Two other illustrative examples are

- $x^{1000}-1$ and $(x-1)(x^{999}+x^{998}+\cdots+x+1)$,
- $(x+1)^{1000}$ and $x^{1000}+1000x^{999}+\cdots+1000x+1$.

It is very hard to find any good strategy to represent mathematical objects satisfying several criteria. In practice, one object may have several different representations. This, however, gives rise to the problem of detecting equality when different representations of the same object are encountered. In addition, one has to be able to convert a given representation to others and simplify the representations.

Consider the *integers.* In the form level, one can represent the integers using base B representation, while at the data structure level they can be represented by a linked list or as an array.

Rational numbers can be represented by two integers, a numerator and a denominator. Considering memory constraints, one needs to ensure that rational numbers are in lowest terms and also that the denominator is positive (although other choices, such as positive numerator, are also possible). This implies that a greatest common

divisor computation has to be performed. Since the ring of integers is a Euclidean domain, this can be easily computed using the Euclidean algorithm. The uniqueness of the representation follows from the choice of the denominator's sign.

Multivariate polynomials (elements of $R[x_1, x_2, \ldots, x_n]$, where R is an integral domain) can be represented in the form $a_1 x^{e_1} + a_2 x^{e_2} + \cdots + a_n x^{e_n}$, where $a_i \in R \backslash \{0\}$ and for $e_i = (e_{i_1}, \ldots, e_{i_n})$, one can write x^{e_i} for $x_1^{e_{i_1}} x_2^{e_{i_2}} \cdots x_n^{e_{i_n}}$. In the form level, one can consider the following levels of abstraction:

1. Expanded or factored representation, where the products are multiplied out or the expression is in product form. Compare
   - $x^2y - x^2 + y - 1$, and
   - $\left(x^2 + 1\right)(y - 1)$.
2. Recursive or distributive representation (only for multivariate polynomials). In the bivariate case, the polynomial $f(x, y)$ can be viewed as an element of the domain $R[x, y]$, $(R[x])[y]$ or $(R[y])[x]$. Compare
   - $x^2y^2 + x^2 + xy^2 - 1$,
   - $(x^2 + x)y^2 + x^2 - 1$, and
   - $(y^2 + 1)x^2 + y^2x - 1$.

At the data structure level, there can be dense or sparse representation. Either all terms are considered, or only those having non-zero coefficients. Compare $x^4 + 0x^3 + 0x^2 + 0x - 1$ and $x^4 - 1$. In practice, multivariate polynomials are represented mainly in the sparse way.

The traditional approach of representing *power series* of the form $\sum_{i=0}^{\infty} a_i x^i$ is to truncate at some specified point, and then to regard them as univariate polynomials. However, this is not a real representation, since many power series can have the same representation. To overcome this disadvantage, there exists a technique of representing power series by a procedure generating all coefficients (rather than by any finite list of coefficients). The generating function is a computable function f such that $f(i) = a_i$. To perform an operation with power series, it is enough to know how to compute the coefficients of the resulting series from the coefficients of the operands. For example, the coefficients h_i of the product of the power series f and g can be computed as $h_i = \sum_{k=0}^{i} f_k g_{i-k}$. In that way, the coefficients are computed when they are needed. This technique is called ***lazy evaluation***.

Since computer algebra programs compute in a symbolic way with arbitrary accuracy, in addition to examining *time complexity* of the algorithms it is also important to examine their *space complexity.*[1] Consider the simple problem of solving a linear system having n equations an n unknowns with integer coefficients which require ω computer word of storage. Using Gaussian elimination, it is easy to see that each coefficient of the reduced linear system may need $2^{n-1}\omega$ computer words of storage. In other words, Gaussian elimination suffers from exponential growth

[1] We consider the running time as the number of operations executed, according to the RAM-model. Considering the Turing-machine model, and using machine words with constant length, we do not have this problem, since in this case space is always bounded by the time.

in the size of the coefficients. Note that if we applied the same method to linear systems having polynomial coefficients, we would have exponential growth both in the size of the numerical coefficients of the polynomials and in the degrees of the polynomials themselves. In spite of the observed exponential growth, the final result of the Gaussian elimination will always be of reasonable size because by Cramer's rule we know that each component of the solution to such a linear system is a ratio of two determinants, each of which requires approximately $n\omega$ computer words. The phenomenon described above is called ***intermediate expression swell.*** This often appears in computer algebra algorithms.

Example 6.2 Using only integer arithmetic we solve the following system of linear equations:

$$\begin{aligned} 37x + 22y + 22z &= 1\ , \\ 31x - 14y - 25z &= 97\ , \\ -11x + 13y + 15z &= -86\ . \end{aligned}$$

First, we eliminate variable x from the second equation. We multiply the first row by 31, the second by -37 and take their sum. If we apply this strategy for the third equation to eliminate variable x, we get the following system.

$$\begin{aligned} 37x + 22y + 22z &= 1\ , \\ 1200y + 1607z &= -3558\ , \\ 723y + 797z &= -3171\ . \end{aligned}$$

Now, we eliminate variable y multiplying the second equation by 723, the third one by -1200, then taking their sum. The result is

$$\begin{aligned} 37x + 22y + 22z &= 1\ , \\ 1200y + 1607z &= -3558\ , \\ 205461z &= 1232766\ . \end{aligned}$$

Continuing this process of eliminating variables, we get the following system:

$$\begin{aligned} 1874311479932400x &= 5622934439797200\ , \\ 246553200y &= -2712085200\ , \\ 205461z &= 1232766\ . \end{aligned}$$

After some simplification, we get that $x = 3, y = -11, z = 6$. If we apply greatest common divisor computations in each elimination step, the coefficient growth will be less drastic.

In order to avoid the intermediate expression swell phenomenon, one uses modular techniques. Instead of performing the operations in the base structure R (e.g. Euclidean ring), they are performed in some factor structure, and then, the result is transformed back to R (Figure 6.1). In general, modular computations can be performed efficiently, and the reconstruction steps can be made with some interpolation strategy. Note that modular algorithms are very common in computer algebra, but it is not a universal technique.

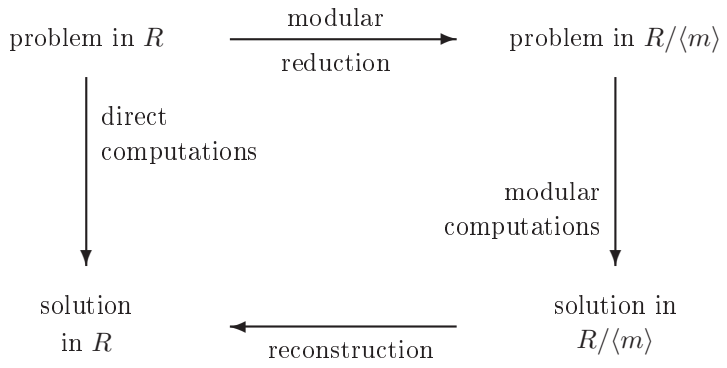


Figure 6.1. The general scheme of modular computations.

6.2. Common roots of polynomials

Let R be an integral domain and let

$$f(x) = f_0 + f_1 x + \cdots + f_{m-1}x^{m-1} + f_m x^m \in R[x],\ f_m \neq 0\ , \tag{6.1}$$
$$g(x) = g_0 + g_1 x + \cdots + g_{n-1}x^{n-1} + g_n x^n \in R[x],\ g_n \neq 0 \tag{6.2}$$

be arbitrary polynomials with $n, m \in \mathbb{N}, n + m > 0$. Let us give a necessary and sufficient condition for f and g sharing a common root in R.

6.2.1. Classical and extended Euclidean algorithm

If T is a field, then $T[x]$ is a Euclidean domain. Recall that we call an integral domain R Euclidean together with the function $\varphi : R \setminus \{0\} \to \mathbb{N}$ if for all $a, b \in R$ $(b \neq 0)$, there exist $q, r \in R$ such that $a = qb + r$, where $r = 0$ or $\varphi(r) < \varphi(b)$; furthermore, for all $a, b \in R \setminus \{0\}$, we have $\varphi(ab) \geq \varphi(a)$. The element $q = a \operatorname{quo} b$ is called the ***quotient*** and $r = a \operatorname{rem} b$ is called the ***remainder***. If we are working in a Euclidean domain, we would like the greatest common divisor to be unique. For this, a unique element has to be chosen from each equivalence class obtained by multiplying by the units of the ring R. (For example, in case of integers we always choose the non-negative one from the classes $\{0\}, \{-1, 1\}, \{-2, 2\}, \ldots$) Thus, every element $a \in R$ has a unique form

$$a = \operatorname{unit}(a) \cdot \operatorname{normal}(a)\ ,$$

where $\operatorname{normal}(a)$ is called the ***normal form*** of a. Let us consider a Euclidean domain $R = T[x]$ over a field T. Let the normal form of $a \in R$ be the corresponding normalised monic polynomial, that is, $\operatorname{normal}(a) = a/\operatorname{lc}(a)$, where $\operatorname{lc}(a)$ denotes the leading coefficient of polynomial a. Let us summarise these important cases:

- If $R = \mathbf{Z}$ then $\text{unit}(a) = \text{sgn}(a)$ $(a \neq 0)$ and $\varphi(a) = \text{normal}(a) = \mid a \mid$,
- if $R = T[x]$ (T is a field) then $\text{unit}(a) = \text{lc}(a)$ (the leading coefficient of polynomial a with the convention $\text{unit}(0) = 1$), $\text{normal}(a) = a/\text{lc}(a)$ and $\varphi(a) = \deg a$.

The following algorithm computes the greatest common divisor of two arbitrary elements of a Euclidean domain. Note that this is one of the most ancient algorithms of the world, already known by Euclid around 300 B.C.

CLASSICAL-EUCLIDEAN(a, b)

```
1  c ← normal(a)
2  d ← normal(b)
3  while d ≠ 0
4        do r ← c rem d
5           c ← d
6           d ← r
7  return normal(c)
```

In the ring of integers, the remainder in line 4 becomes $c - \lfloor c/d \rfloor$. When $R = T[x]$, where T is a field, the remainder in line 4 can be calculated by the algorithm EUCLIDEAN-DIVISION-UNIVARIATE-POLYNOMIALS(c, d), the analysis of which is left to Exercise 6.2-1.

Figure 6.2 shows the operation of the CLASSICAL-EUCLIDEAN algorithm in $\mathbf{Z}$ and $\mathbb{Q}[x]$. Note that in $\mathbf{Z}$ the program only enters the **while** loop with non-negative numbers and the remainder is always non-negative, so the normalisation in line 7 is not needed.

Before examining the running time of the CLASSICAL-EUCLIDEAN algorithm, we deal with an extended version of it.

EXTENDED-EUCLIDEAN(a, b)

```
1  (r0, u0, v0) ← (normal(a), 1, 0)
2  (r1, u1, v1) ← (normal(b), 0, 1)
3  while r1 ≠ 0
4        do q ← r0 quo r1
5           r ← r0 − q r1
6           u ← (u0 − q u1)
7           v ← (v0 − q v1)
8           (r0, u0, v0) ← (r1, u1, v1)
9           (r1, u1, v1) ← (r, u, v)
10 return (normal(r0), u0/(unit(a) · unit(r0)), v0/(unit(b) · unit(r0)))
```

It is known that in the Euclidean domain R, the greatest common divisor of elements $a, b \in R$ can be expressed in the form $\gcd(a, b) = au + bv$ with appropriate elements $u, v \in R$. However, this pair u, v is not unique. For if u_0, v_0 are appropriate, then so are $u_1 = u_0 + bt$ and $v_1 = v_0 - at$ for all $t \in R$:

$$au_1 + bv_1 = a(u_0 + bt) + b(v_0 - at) = au_0 + bv_0 = \gcd(a, b) \ .$$

iteration	r	c	d
–	–	18	30
1	18	30	18
2	12	18	12
3	6	12	6
4	0	6	0

(a) The operation of CLASSICAL-EUCLIDEAN$(-18, 30)$.

iteration	r	c	d
–	–	$x^4 - \frac{17}{3}x^3 + \frac{13}{3}x^2 - \frac{23}{3}x + \frac{14}{3}$	$x^3 - \frac{20}{3}x^2 + 7x - 2$
1	$4x^2 - \frac{38}{3}x + \frac{20}{3}$	$x^3 - \frac{20}{3}x^2 + 7x - 2$	$4x^2 - \frac{38}{3}x + \frac{20}{3}$
2	$-\frac{23}{4}x + \frac{23}{6}$	$4x^2 - \frac{38}{3}x + \frac{20}{3}$	$-\frac{23}{4}x + \frac{23}{6}$
3	0	$-\frac{23}{4}x + \frac{23}{6}$	0

(b) The operation of
CLASSICAL-EUCLIDEAN$(12x^4 - 68x^3 + 52x^2 - 92x + 56 - 12x^3 + 80x^2 - 84x + 24)$.

Figure 6.2. Illustration of the operation of the CLASSICAL-EUCLIDEAN algorithm in $\mathbf{Z}$ and $\mathbb{Q}[x]$. In case **(a)**, the input is $a = -18, b = 30, a, b \in \mathbf{Z}$. The first two lines of the pseudocode compute the absolute values of the input numbers. The loop between lines 3 and 6 is executed four times, values r, c and d in these iterations are shown in the table. The CLASSICAL-EUCLIDEAN$(-18,30)$ algorithm outputs 6 as result. In case **(b)**, the input parameters are $a = 12x^4 - 68x^3 + 52x^2 - 92x + 56, b = -12x^3 + 80x^2 - 84x + 24 \in \mathbb{Q}[x]$. The first two lines compute the normal form of the polynomials, and the **while** loop is executed three times. The output of the algorithm is the polynomial normal$(c) = x - 2/3$.

The CLASSICAL-EUCLIDEAN algorithm is completed in a way that beside the greatest common divisor it outputs an appropriate pair $u, v \in R$ as discussed above.

Let $a, b \in R$, where R is a Euclidean domain together with the function φ. The equations

$$r_0 = u_0 a + v_0 b \quad \text{and} \quad r_1 = u_1 a + v_1 b \tag{6.3}$$

are obviously fulfilled due to the initialisation in the first two lines of the pseudocode EXTENDED-EUCLIDEAN. We show that equations (6.3) are invariant under the transformations of the **while** loop of the pseudocode. Let us presume that the conditions (6.3) are fulfilled before an iteration of the loop. Then lines 4–5 of the pseudocode imply

$$r = r_0 - q r_1 = u_0 a + v_0 b - q(a u_1 + b v_1) = a(u_0 - q u_1) + b(v_0 - q v_1) \ ,$$

hence, because of lines 6–7,

$$r = a(u_0 - q u_1) + b(v_0 - q v_1) = au + bv.$$

Lines 8–9 perform the following operations: u_0, v_0 take the values of u_1 and v_1, then u_1, v_1 take the values of u and v, while r_0, r_1 takes the value of r_1 and r. Thus, the equalities in (6.3) are also fulfilled after the iteration of the **while** loop. Since $\varphi(r_1) < \varphi(r_0)$ in each iteration of the loop, the series $\{\varphi(r_i)\}$ obtained in lines 8–9 is a strictly decreasing series of natural numbers, so sooner or later the control steps

out of the **while** loop. The greatest common divisor is the last non-zero remainder in the series of Euclidean divisions, that is, r_0 in lines 8–9.

Example 6.3 Let us examine the series of remainders in the case of polynomials

$$a(x) = 63x^5 + 57x^4 - 59x^3 + 45x^2 - 8\,, \tag{6.4}$$
$$b(x) = -77x^4 + 66x^3 + 54x^2 - 5x + 99\,. \tag{6.5}$$

$$\begin{aligned}
r_0 &= x^5 + \frac{19}{21}x^4 - \frac{59}{63}x^3 + \frac{5}{7}x^2 - \frac{8}{63}\,,\\
r_1 &= x^4 - \frac{6}{7}x^3 - \frac{54}{77}x^2 + \frac{5}{77}x - \frac{9}{7}\,,\\
r_2 &= \frac{6185}{4851}x^3 + \frac{1016}{539}x^2 + \frac{1894}{1617}x + \frac{943}{441}\,,\\
r_3 &= \frac{771300096}{420796475}x^2 + \frac{224465568}{420796475}x + \frac{100658427}{38254225}\,,\\
r_4 &= -\frac{125209969836038125}{113868312759339264}x - \frac{3541728593586625}{101216278008301568}\,,\\
r_5 &= \frac{47175801636356999274360 5121}{1803229860333151158054368 75}\,.
\end{aligned}$$

The values of the variables u_0, v_0 before the execution of line 10 are

$$\begin{aligned}
u_0 &= \frac{113868312759339264}{125209969836038125}x^3 - \frac{66263905285897833785656224}{819649936515068708206531 25}x^2\\
&\quad - \frac{172214445262403690128205666 1}{901614930166575579027184375}x + \frac{145175798748706922498167895 4}{901614930166575579027184375}\,,\\
v_0 &= -\frac{113868312759339264}{125209969836038125}x^4 - \frac{650693816081118388878813536}{819649936515068708206531 25}x^3\\
&\quad + \frac{178270505434627626751446079}{819649936515068708206531 25}x^2 + \frac{638085922305129542614635 3}{819649936515068708206531 25}x\\
&\quad - \frac{179818001183413133012445617}{819649936515068708206531 25}\,.
\end{aligned}$$

The return values are:

$$\begin{aligned}
\gcd(a,b) &= 1,\\
u &= \frac{2580775248128}{467729710968369}x^3 - \frac{3823697946464}{779549518280615}x^2\\
&\quad - \frac{27102209423483}{2338648554841845}x + \frac{7615669511954}{779549518280615}\,,\\
v &= \frac{703847794944}{155909903656123}x^4 + \frac{3072083769824}{779549518280615}x^3\\
&\quad - \frac{25249752472633}{2338648554841845}x^2 - \frac{301255883677}{779549518280615}x + \frac{25468935587159}{2338648554841845}\,.
\end{aligned}$$

We can see that the size of the coefficients show a drastic growth. One might ask why we do not normalise in *every* iteration of the **while** loop? This idea leads to the normalised version of the Euclidean algorithm for polynomials.

EXTENDED-EUCLIDEAN-NORMALISED(a, b)

```
1  e_0 ← unit(a)
2  (r_0, u_0, v_0) ← (normal(a), e_0^{-1}, 0)
3  e_1 ← unit(b)
4  (r_1, u_1, v_1) ← (normal(b), 0, e_1^{-1})
5  while r_1 ≠ 0
6        do q ← r_0 quo r_1
7           s ← r_0 rem r_1
8           e ← unit(s)
9           r ← normal(s)
10          u ← (u_0 − q u_1)/e
11          v ← (v_0 − q v_1)/e
12          (r_0, u_0, v_0) ← (r_1, u_1, v_1)
13          (r_1, u_1, v_1) ← (r, u, v)
14 return (r_0, u_0, v_0)
```

Example 6.4 Let us look at the series of remainders and series e obtained in the EXTENDED-EUCLIDEAN-NORMALISED algorithm in case of the polynomials (6.4) and (6.5)

$$\begin{aligned}
r_0 &= x^5 + \frac{19}{21}x^4 - \frac{59}{63}x^3 + \frac{5}{7}x^2 - \frac{8}{63}, & e_0 &= 63\,,\\
r_1 &= x^4 - \frac{6}{7}x^3 - \frac{54}{77}x^2 + \frac{5}{77}x - \frac{9}{7}, & e_1 &= -77\,,\\
r_2 &= x^3 + \frac{9144}{6185}x^2 + \frac{5682}{6185}x + \frac{10373}{6185}, & e_2 &= \frac{6185}{4851}\,,\\
r_3 &= x^2 + \frac{2338183}{8034376}x + \frac{369080899}{257100032}, & e_3 &= \frac{771300096}{420796475}\,,\\
r_4 &= x + \frac{166651173}{5236962760}, & e_4 &= -\frac{222685475860375}{258204790837504}\,,\\
r_5 &= 1, & e_5 &= \frac{156579848512133360531}{109703115798507270400}\,.
\end{aligned}$$

Before the execution of line 14 of the pseudocode, the values of the variables $\gcd(a, b) = r_0, u = u_0, v = v_0$ are

$$\begin{aligned}
\gcd(a,b) &= 1,\\
u &= \frac{2580775248128}{467729710968369}x^3 - \frac{3823697946464}{779549518280615}x^2\\
&\quad - \frac{27102209423483}{2338648554841845}x + \frac{7615669511954}{779549518280615}\,,\\
v &= \frac{703847794944}{155909903656123}x^4 + \frac{3072083769824}{779549518280615}x^3\\
&\quad - \frac{25249752472633}{2338648554841845}x^2 - \frac{301255883677}{779549518280615}x + \frac{25468935587159}{2338648554841845}\,.
\end{aligned}$$

Looking at the size of the coefficients in $\mathbb{Q}[x]$, the advantage of the normalised version is obvious, but we could still not avoid the growth. To get a machine architecture-dependent description and analysis of the EXTENDED-EUCLIDEAN-NORMALISED al-

gorithm, we introduce the following notation. Let

$$\begin{aligned}
\lambda(a) &= \lfloor \log_2 |a|/w \rfloor + 1 \text{ if } a \in \mathbf{Z} \setminus \{0\}, \text{ and } \lambda(0) = 0 \ , \\
\lambda(a) &= \max\{\lambda(b), \lambda(c)\} \text{ if } a = b/c \in \mathbb{Q},\ b, c \in \mathbf{Z},\ \gcd(b, c) = 1 \ , \\
\lambda(a) &= \max\{\lambda(b), \lambda(a_0), \ldots, \lambda(a_n)\} \text{ if } a = \sum_{0 \le i \le n} a_i x^i / b \in \mathbb{Q}[x] \ , \\
&\quad a_i \in \mathbf{Z},\ b \in \mathbb{N}^+,\ \gcd(b, a_0, \ldots, a_n) = 1 \ ,
\end{aligned}$$

where w is the word length of the computer in bits. It is easy to verify that if $a, b \in \mathbf{Z}[x]$ and $c, d \in \mathbb{Q}$, then

$$\begin{aligned}
\lambda(c + d) &\le \lambda(c) + \lambda(d) + 1 \ , \\
\lambda(a + b) &\le \max\{\lambda(a), \lambda(b)\} + 1 \ , \\
\lambda(cd), \lambda(c/d) &\le \lambda(c) + \lambda(d) \ , \\
\lambda(ab) &\le \lambda(a) + \lambda(b) + \lambda(\min\{\deg a, \deg b\} + 1) \ .
\end{aligned}$$

We give the following theorems without proof.

Theorem 6.1 *If $a, b \in \mathbf{Z}$ and $\lambda(a) = m \ge n = \lambda(b)$, then the* Classical-Euclidean *and* Extended-Euclidean *algorithms require $O(mn)$ machine-word arithmetic operations.*

Theorem 6.2 *If F is a field and $a, b \in F[x]$, $\deg(a) = m \ge n = \deg(b)$, then the* Classical-Euclidean, Extended-Euclidean *and* Extended-Euclidean-Normalised *algorithms require $O(mn)$ elementary operations in F.*

Can the growth of the coefficients be due to the choice of our polynomials? Let us examine a single Euclidean division in the Extended-Euclidean-Normalised algorithm. Let $a = bq + e^* r$, where

$$\begin{aligned}
a &= x^m + \frac{1}{c} \sum_{i=0}^{m-1} a_i x^i \in \mathbb{Q}[x] \ , \\
b &= x^n + \frac{1}{d} \sum_{i=0}^{n-1} b_i x^i \in \mathbb{Q}[x] \ ,
\end{aligned}$$

and $r \in \mathbb{Q}[x]$ are monic polynomials, $a_i, b_i \in \mathbf{Z}$, $e^* \in \mathbb{Q}$, $c, d \in \mathbb{N}^+$, and consider the case $n = m - 1$. Then

$$\begin{aligned}
q &= x + \frac{a_{m-1} d - b_{n-1} c}{cd} \ , \\
\lambda(q) &\le \lambda(a) + \lambda(b) + 1 \ , \\
e^* r &= a - qb = \frac{acd^2 - xbcd^2 - (a_{m-1} d - b_{n-1} c) bd}{cd^2} \ , \\
\lambda(e^* r) &\le \lambda(a) + 2\lambda(b) + 3 \ . \qquad (6.6)
\end{aligned}$$

Note that the bound (6.6) is valid for the coefficients of the remainder polynomial r as well, that is, $\lambda(r) \le \lambda(a) + 2\lambda(b) + 3$. So in case $\lambda(a) \sim \lambda(b)$, the size of the

coefficients may only grow by a factor of around three in each Euclidean division. This estimate seems accurate for pseudorandom polynomials, the interested reader should look at Problem 6-1 The worst case estimate suggests that

$$\lambda(r_l) = O(3^l \cdot \max\{\lambda(a), \lambda(b)\}) ,$$

where l denotes the running time of the EXTENDED-EUCLIDEAN-NORMALISED algorithm, practically, the number of times the **while** loop is executed. Luckily, this exponential growth is not achieved in each iteration of the loop, and altogether the growth of the coefficients is bounded polynomially in terms of the input. Later we will see that the growth can be eliminated using modular techniques.

Summarising: after computing the greatest common divisor of the polynomials $f, g \in R[x]$ (R is a field), f and g have a common root if and only if their greatest common divisor is not a constant. For if $\gcd(f, g) = d \in R[x]$ is not a constant, then the roots of d are also roots of f and g, since d divides f and g. On the other hand, if f and g have a root in common, then their greatest common divisor cannot be a constant, since the common root is also a root of it.

6.2.2. Primitive Euclidean algorithm

If R is a UFD (unique factorisation domain, where every non-zero, non-unit element can be written as a product of irreducible elements in a unique way up to reordering and multiplication by units) but not necessarily a Euclidean domain then, the situation is more complicated, since we may not have a Euclidean algorithm in $R[x]$. Luckily, there are several useful methods due to: (1) unique factorisation in $R[x]$, (2) the existence of a greatest common divisor of two or more arbitrary elements.

The first possible method is to perform the calculations in the field of fractions of R. The polynomial $p(x) \in R[x]$ is called a ***primitive polynomial*** if there is no prime in R that divides all coefficients of $p(x)$. A famous lemma by Gauss says that the product of primitive polynomials is also primitive, hence, for the primitive polynomials f, g, $d = \gcd(f, g) \in R[x]$ if and only if $d = \gcd(f, g) \in H[x]$, where H denotes the field of fractions of R. So we can calculate greatest common divisors in $H[x]$ instead of $R[x]$. Unfortunately, this approach is not really effective because arithmetic in the field of fractions H is much more expensive than in R.

A second possibility is an algorithm similar to the Euclidean algorithm: in the ring of polynomials in one variable over an integral domain, a so-called ***pseudo-division*** can be defined. Using the polynomials (6.1), (6.2), if $m \geq n$, then there exist $q, r \in R[x]$, such that

$$g_n^{m-n+1} f = gq + r ,$$

where $r = 0$ or $\deg r < \deg g$. The polynomial q is called the ***pseudo-quotient*** of f and g and r is called the ***pseudo-remainder***. The notation is $q = \mathrm{pquo}(f, g), r = \mathrm{prem}(f, g)$.

Example 6.5 Let

$$f(x) = 12x^4 - 68x^3 + 52x^2 - 92x + 56 \in \mathbf{Z}[x] , \tag{6.7}$$

$$g(x) = -12x^3 + 80x^2 - 84x + 24 \in \mathbf{Z}[x] . \tag{6.8}$$

iteration	r	c	d
–	–	$3x^4 - 17x^3 + 13x^2 - 23x + 14$	$-3x^3 + 20x^2 - 21x + 6$
1	$108x^2 - 342x + 108$	$-3x^3 + 20x^2 - 21x + 6$	$6x^2 - 19x + 10$
2	$621x - 414$	$6x^2 - 19x + 10$	$3x - 2$
3	0	$3x - 2$	0

Figure 6.3. The illustration of the operation of the PRIMITIVE-EUCLIDEAN algorithm with input $a(x) = 12x^4 - 68x^3 + 52x^2 - 92x + 56$, $b(x) = -12x^3 + 80x^2 - 84x + 24 \in \mathbf{Z}[x]$. The first two lines of the program compute the primitive parts of the polynomials. The loop between lines 3 and 6 is executed three times, the table shows the values of r, c and d in the iterations. In line 7, variable γ equals $\gcd(4, 4) = 4$. The PRIMITIVE-EUCLIDEAN(a, b) algorithm returns $4 \cdot (3x - 2)$ as result.

Then $\operatorname{pquo}(f, g) = -144(x + 1)$, $\operatorname{prem}(f, g) = 1152(6x^2 - 19x + 10)$.

On the other hand, each polynomial $f(x) \in R[x]$ can be written in a unique form

$$f(x) = \operatorname{cont}(f) \cdot \operatorname{pp}(f)$$

up to a unit factor, where $\operatorname{cont}(f) \in R$ and $\operatorname{pp}(f) \in R[x]$ are primitive polynomials. In this case, $\operatorname{cont}(f)$ is called the ***content,*** $\operatorname{pp}(f)$ is called the ***primitive part*** of $f(x)$. The uniqueness of the form can be achieved by the normalisation of units. For example, in the case of integers, we always choose the positive ones from the equivalence classes of $\mathbf{Z}$.

The following algorithm performs a series of pseudo-divisions. The algorithm uses the function prem(), which computes the pseudo-remainder, and it assumes that we can calculate greatest common divisors in R, contents and primitive parts in $R[x]$. The input is $a, b \in R[x]$, where R is a UFD. The output is the polynomial $\gcd(a, b) \in R[x]$.

PRIMITIVE-EUCLIDEAN(a, b)

```
1  c ← pp(f)
2  d ← pp(g)
3  while d ≠ 0
4        do r ← prem(c, d)
5           c ← d
6           d ← pp(r)
7  γ ← gcd(cont(a), cont(b))
8  δ ← γc
9  return δ
```

The operation of the algorithm is illustrated by Figure 6.3. The running time of the PRIMITIVE-EUCLIDEAN algorithm is the same as the running time of the previous versions of the Euclidean algorithm.

The PRIMITIVE-EUCLIDEAN algorithm is very important because the ring $R[x_1, x_2, \ldots, x_t]$ of multivariate polynomials is a UFD, so we apply the algorithm recursively, e.g. in $R[x_2, \ldots, x_t][x_1]$, using computations in the UFDs $R[x_2, \ldots, x_t], \ldots,$

$R[x_t]$. In other words, the recursive view of multivariate polynomial rings leads to the recursive application of the PRIMITIVE-EUCLIDEAN algorithm in a straightforward way.

We may note that, like above, the algorithm shows a growth in the coefficients.

Let us take a detailed look at the UFD $\mathbf{Z}[x]$. The bound on the size of the coefficients of the greatest common divisor is given by the following theorem, which we state without proof.

Theorem 6.3 (Landau-Mignotte). *Let* $a(x) = \sum_{i=0}^{m} a_i x^i$, $b(x) = \sum_{i=0}^{n} b_i x^i \in \mathbf{Z}[x]$, $a_m \neq 0 \neq b_n$ *and* $b(x) \mid a(x)$. *Then*

$$\sum_{i=1}^{n} |b_i| \leq 2^n \left| \frac{b_n}{a_m} \right| \sqrt{\sum_{i=0}^{m} a_i^2} \ .$$

Corollary 6.4 *With the notations of the previous theorem, the absolute value of any coefficient of the polynomial* $gcd(a, b) \in \mathbf{Z}[x]$ *is smaller than*

$$2^{\min\{m,n\}} \cdot gcd(a_m, b_n) \cdot \min\left\{ \frac{1}{|a_m|} \sqrt{\sum_{i=1}^{m} a_i^2}, \frac{1}{|b_n|} \sqrt{\sum_{i=1}^{n} b_i^2} \right\} .$$

Proof The greatest common divisor of a and b obviously divides both a and b, and its degree is at most the minimum of their degrees. Furthermore, the leading coefficient of the greatest common divisor divides a_m and b_n, so it also divides $\gcd(a_m, b_n)$. ■

Example 6.6 Corollary 6.4 implies that the absolute value of the coefficients of the greatest common divisor is at most $\lfloor 32/9\sqrt{3197} \rfloor = 201$ for the polynomials (6.4), (6.5), and at most $\lfloor 32\sqrt{886} \rfloor = 952$ for the polynomials (6.7) and (6.8).

6.2.3. The resultant

The following method describes the necessary and sufficient conditions for the common roots of (6.1) and (6.2) in the most general context. As a further advantage, it can be applied to solve algebraic equation systems of higher degree.

Let R be an integral domain and H its field of fractions. Let us consider the smallest extension K of H over which both $f(x)$ of (6.1) and $g(x)$ of (6.2) splits into linear factors. Let us denote the roots (in K) of the polynomial $f(x)$ by $\alpha_1, \alpha_2, \ldots, \alpha_m$, and the roots of $g(x)$ by $\beta_1, \beta_2, \ldots, \beta_n$. Let us form the following product:

$$\begin{aligned} \operatorname{res}(f, g) &= f_m^n g_n^m (\alpha_1 - \beta_1)(\alpha_1 - \beta_2) \cdots (\alpha_1 - \beta_n) \\ &\quad \cdot (\alpha_2 - \beta_1)(\alpha_2 - \beta_2) \cdots (\alpha_2 - \beta_n) \\ &\quad \vdots \\ &\quad \cdot (\alpha_m - \beta_1)(\alpha_m - \beta_2) \cdots (\alpha_m - \beta_n) \\ &= f_m^n g_n^m \prod_{i=1}^{m} \prod_{j=1}^{n} (\alpha_i - \beta_j) \ . \end{aligned}$$

It is obvious that $\mathrm{res}(f,g)$ equals to 0 if and only if $\alpha_i = \beta_j$ for some i and j, that is, f and g have a common root. The product $\mathrm{res}(f,g)$ is called the ***resultant*** of the polynomials f and g. Note that the value of the resultant depends on the order of f and g, but the resultants obtained in the two ways can only differ in sign.

$$\begin{aligned}\mathrm{res}(g,f) &= g_n^m f_m^n \prod_{j=1}^{n}\prod_{i=1}^{m}(\beta_j - \alpha_i) \\ &= (-1)^{mn} f_m^n g_n^m \prod_{i=1}^{m}\prod_{j=1}^{n}(\alpha_i - \beta_j) = (-1)^{mn}\mathrm{res}(f,g)\ .\end{aligned}$$

Evidently, this form of the resultant cannot be applied in practice, since it presumes that the roots are known. Let us examine the different forms of the resultant. Since

$$\begin{aligned}f(x) &= f_m(x-\alpha_1)(x-\alpha_2)\cdots(x-\alpha_m) \quad (f_m \neq 0)\ , \\ g(x) &= g_n(x-\beta_1)(x-\beta_2)\cdots(x-\beta_n) \quad (g_n \neq 0)\ ,\end{aligned}$$

hence,

$$\begin{aligned}g(\alpha_i) &= g_n(\alpha_i-\beta_1)(\alpha_i-\beta_2)\cdots(\alpha_i-\beta_n) \\ &= g_n \prod_{j=1}^{n}(\alpha_i - \beta_j)\ .\end{aligned}$$

Thus,

$$\begin{aligned}\mathrm{res}(f,g) &= f_m^n \prod_{i=1}^{m}\left(g_n \prod_{j=1}^{n}(\alpha_i - \beta_j)\right) \\ &= f_m^n \prod_{i=1}^{m} g(\alpha_i) = (-1)^{mn} g_n^m \prod_{j=1}^{n} f(\beta_j)\ .\end{aligned}$$

Although it looks a lot more friendly, this form still requires the roots of at least one polynomial. Next we examine how the resultant may be expressed only in terms of the coefficients of the polynomials. This leads to the Sylvester form of the resultant.

Let us presume that polynomial f in (6.1) and polynomial g in (6.2) have a common root. This means that there exists a number $\alpha \in K$ such that

$$\begin{aligned}f(\alpha) &= f_m\alpha^m + f_{m-1}\alpha^{m-1} + \cdots + f_1\alpha + f_0 = 0\ , \\ g(\alpha) &= g_n\alpha^n + g_{n-1}\alpha^{n-1} + \cdots + g_1\alpha + g_0 = 0\ .\end{aligned}$$

Multiply these equations by the numbers $\alpha^{n-1}, \alpha^{n-2}, \ldots, \alpha, 1$ and α^{m-1}, $\alpha^{m-2}, \ldots, \alpha, 1$, respectively. We get n equations from the first one and m from the second one. Consider these $m+n$ equations as a homogeneous system of linear equations in $m+n$ indeterminates. This system has the obviously non-trivial solution $\alpha^{m+n-1}, \alpha^{m+n-2}, \ldots, \alpha, 1$. It is a well-known fact that a homogeneous system with as many equations as indeterminates has non-trivial solutions if and only if its determinant is zero. We get that f and g can

only have common roots if the determinant

$$D = \left|\begin{array}{ccccccccc} f_m & \cdots & \cdots & \cdots & f_0 & & & \uparrow \\ & \ddots & & & & \ddots & & n \\ & & f_m & \cdots & \cdots & \cdots & f_0 & \downarrow \\ g_n & \cdots & \cdots & g_0 & & & & \uparrow \\ & \ddots & & & \ddots & & & m \\ & & \ddots & & & \ddots & & \\ & & & g_n & \cdots & \cdots & g_0 & \downarrow \end{array}\right| \tag{6.9}$$

equals to 0 (there are 0s everywhere outside the dotted areas). Thus, a necessary condition for the existence of common roots is that the determinant D of order $(m+n)$ is 0. Below we prove that D equals to the resultant of f and g, hence, $D = 0$ is also a sufficient condition for common roots. The determinant (6.9) is called the ***Sylvester form*** of the resultant.

Theorem 6.5 *Using the above notation*

$$D = f_m^n \prod_{i=1}^{m} g(\alpha_i) \ .$$

Proof We will precede by induction on m. If $m = 0$, then $f = f_m = f_0$, so the right-hand side is f_0^n. The left-hand side is a determinant of order n with f_0 everywhere in the diagonal, and 0 everywhere else. Thus, $D = f_0^n$, so the statement is true. In the following, presume that $m > 0$ and the statement is true for $m - 1$. If we take the polynomial

$$f^*(x) = f_m(x - \alpha_1) \cdots (x - \alpha_{m-1}) = f^*_{m-1}x^{m-1} + f^*_{m-2}x^{m-2} + \cdots + f^*_1 x + f^*_0$$

instead of f, then f^* and g fulfil the condition:

$$D^* = \left|\begin{array}{ccccccc} f^*_{m-1} & \cdots & \cdots & \cdots & f^*_0 & & \\ & \ddots & & & & \ddots & \\ & & f^*_{m-1} & \cdots & \cdots & \cdots & f^*_0 \\ g_n & \cdots & \cdots & g_0 & & & \\ & \ddots & & & \ddots & & \\ & & \ddots & & & \ddots & \\ & & & g_n & \cdots & \cdots & g_0 \end{array}\right| = f^{*m}_{m-1} \prod_{i=1}^{m-1} g(\alpha_i) \ .$$

Since $f = f^*(x - \alpha_m)$, the coefficients of f and f^* satisfy

$$f_m = f^*_{m-1}, f_{m-1} = f^*_{m-2} - f^*_{m-1}\alpha_m, \ldots, f_1 = f^*_0 - f^*_1\alpha_m, f_0 = -f^*_0\alpha_m.$$

Thus,

$$D = \begin{vmatrix} f_{m-1}^* & f_{m-2}^* - f_{m-1}^*\alpha_m & \cdots & \cdots & -f_0^*\alpha_m & & \\ & \ddots & & & & \ddots & \\ & & f_{m-1}^* & \cdots & \cdots & \cdots & -f_0^*\alpha_m \\ g_n & \cdots & \cdots & g_0 & & & \\ & \ddots & & & \ddots & & \\ & & \ddots & & & \ddots & \\ & & & g_n & \cdots & \cdots & g_0 \end{vmatrix}.$$

We transform the determinant in the following way: add α_m times the first column to the second column, then add α_m times the new second column to the third column, etc. This way the α_m-s disappear from the first n lines, so the first n lines of D^* and the transformed D are identical. In the last m rows, subtract α_m times the second one from the first one, and similarly, always subtract α_m times a row from the row right above it. In the end, D becomes

$$D = \begin{vmatrix} f_{m-1}^* & \cdots & \cdots & \cdots & f_0^* & & & & \\ & \ddots & & & & \ddots & & & \\ & & f_{m-1}^* & \cdots & \cdots & \cdots & & f_0^* & \\ g_n & \cdots & \cdots & g_0 & & & & & \\ & \ddots & & & \ddots & & & & \\ & & \ddots & & & \ddots & & & \\ & & & g_n & \cdots & \cdots & & g_0 & \\ & & & & g_n & g_n\alpha_m + g_{n-1} & \cdots & g(\alpha_m) \end{vmatrix}.$$

Using the last row for expansion, we get $D = D^* g(\alpha_m)$, which implies $D = f_m^n \prod_{i=1}^m g(\alpha_i)$ by the induction hypothesis. ∎

We get that $D = \mathrm{res}(f, g)$, that is, polynomials f and g have a common root in K if and only if determinant D vanishes.

>From an algorithmic point of view, the computation of the resultant in Sylvester form for higher degree polynomials means the computation of a large determinant. The following theorem implies that pseudo-division may simplify the computation.

Theorem 6.6 *For the polynomials f of (6.1) and g of (6.2), in case of $m \geq n > 0$*

$$\begin{cases} res(f,g) = 0, & \text{if } prem(f,g) = 0\ , \\ g_n^{(m-n)(n-1)+d} res(f,g) = (-1)^{mn} res(g,r), & \text{if } r = prem(f,g) \neq 0 \text{ and } d = \deg(r)\ . \end{cases}$$

Proof Multiply the first line of the determinant (6.9) by g_n^{m-n+1}. Let $q = q_{m-n}x^{m-n} + \cdots + q_0 \in R[x]$ and $r = r_d x^d + \cdots + r_0 \in R[x]$ be the uniquely determined polynomials with

$$\begin{aligned} g_n^{m-n+1}(f_m x^m + \cdots + f_0) &= (q_{m-n}x^{m-n} + \cdots + q_0)(g_n x^n + \cdots + g_0) \\ &\quad + r_d x^d + \cdots + r_0\ , \end{aligned}$$

where $r = \operatorname{prem}(f,g)$. Then multiplying row $(n+1)$ of the resultant by q_{m-n}, row $(n+2)$ by q_{m-n-1} etc., and subtracting them from the first row we get the determinant

$$g_n^{m-n+1}\operatorname{res}(f,g) = \begin{vmatrix}
0 & \cdots & 0 & r_d & \cdots & \cdots & r_0 & & & & \\
 & f_m & \cdots & \cdots & \cdots & \cdots & \cdots & f_0 & & & \\
 & & \ddots & & & & & & \ddots & & \\
 & & & f_m & \cdots & \cdots & \cdots & \cdots & \cdots & f_0 & \\
g_n & \cdots & \cdots & \cdots & g_0 & & & & & & \\
 & \ddots & & & & \ddots & & & & & \\
 & & \ddots & & & & \ddots & & & & \\
 & & & \ddots & & & & \ddots & & & \\
 & & & & \ddots & & & & \ddots & & \\
 & & & & & g_n & \cdots & \cdots & \cdots & g_0 &
\end{vmatrix}.$$

Here r_d is in the $(m-d+1)$th column of the first row, and r_0 is in the $(m+1)$th column of the first row.

Similarly, multiply the second row by g_n^{m-n+1}, then multiply rows $(n+2)$, $(n+3)$, ... by q_{m-n}, q_{m-n-1} etc., and subtract them from the second row. Continue the same way for the third, ..., nth row. The result is

$$g_n^{n(m-n+1)}\operatorname{res}(f,g) = \begin{vmatrix}
 & & & r_d & \cdots & \cdots & r_0 & & & & \\
 & & & & \ddots & & & \ddots & & & \\
 & & & & & \ddots & & & \ddots & & \\
 & & & & & & r_d & \cdots & \cdots & r_0 & \\
g_n & \cdots & \cdots & \cdots & g_0 & & & & & & \\
 & \ddots & & & & \ddots & & & & & \\
 & & \ddots & & & & \ddots & & & & \\
 & & & \ddots & & & & \ddots & & & \\
 & & & & \ddots & & & & \ddots & & \\
 & & & & & g_n & \cdots & \cdots & \cdots & g_0 &
\end{vmatrix}.$$

After reordering the rows

$$g_n^{n(m-n+1)}\mathrm{res}(f,g) = (-1)^{mn}\begin{vmatrix} g_n & \cdots & \cdots & \cdots & g_0 & & & & & & \\ & \ddots & & & & \ddots & & & & & \\ & & \ddots & & & & \ddots & & & & \\ & & & g_n & \cdots & \cdots & \cdots & g_0 & & & \\ & & & & \ddots & & & & \ddots & & \\ & & & & & g_n & \cdots & \cdots & \cdots & g_0 & \\ & & & r_d & \cdots & \cdots & r_0 & & & & \\ & & & & \ddots & & & \ddots & & & \\ & & & & & \ddots & & & \ddots & & \\ & & & & & & r_d & \cdots & \cdots & r_0 & \end{vmatrix} .$$

Note that

$$\begin{vmatrix} g_n & \cdots & \cdots & \cdots & g_0 & & \\ & \ddots & & & & \ddots & \\ & & g_n & \cdots & \cdots & \cdots & g_0 \\ r_d & \cdots & \cdots & r_0 & & & \\ & \ddots & & & \ddots & & \\ & & \ddots & & & \ddots & \\ & & & r_d & \cdots & \cdots & r_0 \end{vmatrix} = \mathrm{res}(g,r) ,$$

thus,

$$g_n^{n(m-n+1)}\mathrm{res}(f,g) = (-1)^{mn} g_n^{m-d}\mathrm{res}(g,r) ,$$

and therefore

$$g_n^{(m-n)(n-1)+d}\mathrm{res}(f,g) = (-1)^{mn}\mathrm{res}(g,r) . \tag{6.10}$$

■

Equation (6.10) describes an important relationship. Instead of computing the possibly gigantic determinant $\mathrm{res}(f,g)$, we perform a series of pseudo-divisions and apply (6.10) in each step. We calculate the resultant only when no more pseudo-division can be done. An important consequence of the theorem is the following corollary.

Corollary 6.7 *There exist polynomials $u, v \in R[x]$ such that $res(f,g) = fu + gv$, with $\deg u < \deg g$, $\deg v < \deg f$.*

Proof Multiply the ith column of the determinant form of the resultant by x^{m+n-i}

and add it to the last column for $i = 1, \ldots, (m+n-1)$. Then

$$\operatorname{res}(f,g) = \begin{vmatrix} f_m & \cdots & \cdots & f_0 & \cdots & x^{n-1}f \\ & \ddots & & & \ddots & \vdots \\ & & f_m & \cdots & \cdots & f \\ g_n & \cdots & \cdots & g_0 & \cdots & x^{m-1}g \\ & \ddots & & & \ddots & \vdots \\ & & g_n & \cdots & \cdots & g \end{vmatrix} .$$

Using the last column for expansion and factoring f and g, we get the statement with the restrictions on the degrees. ■

The most important benefit of the resultant method, compared to the previously discussed methods, is that the input polynomials may contain symbolic coefficients as well.

Example 6.7 Let

$$\begin{aligned} f(x) &= 2x^3 - \xi x^2 + x + 3 \in \mathbb{Q}[x] , \\ g(x) &= x^2 - 5x + 6 \in \mathbb{Q}[x] . \end{aligned}$$

Then the existence of common rational roots of f and g cannot be decided by variants of the Euclidean algorithm, but we can decide it with the resultant method. Such a root exists if and only if

$$\operatorname{res}(f,g) = \begin{vmatrix} 2 & -\xi & 1 & 3 & \\ & 2 & -\xi & 1 & 3 \\ 1 & -5 & 6 & & \\ & 1 & -5 & 6 & \\ & & 1 & -5 & 6 \end{vmatrix} = 36\xi^2 - 429\xi + 1260 = 3(4\xi - 21)(3\xi - 20) = 0 ,$$

that is, when $\xi = 20/3$ or $\xi = 21/4$.

The significance of the resultant is not only that we can decide the existence of common roots of polynomials, but also that using it we can reduce the solution of algebraic equation systems to solving univariate equations.

Example 6.8 Let

$$f(x,y) = x^2 + xy + 2x + y - 1 \in \mathbf{Z}[x,y] , \tag{6.11}$$

$$g(x,y) = x^2 + 3x - y^2 + 2y - 1 \in \mathbf{Z}[x,y] . \tag{6.12}$$

Consider polynomials f and g as elements of $(\mathbf{Z}[x])[y]$. They have a common root if and only if

$$\operatorname{res}_y(f,g) = \begin{vmatrix} x+1 & x^2+2x-1 & 0 \\ 0 & x+1 & x^2+2x-1 \\ -1 & 2 & x^2+3x-1 \end{vmatrix} = -x^3 - 2x^2 + 3x = 0 .$$

Common roots in $\mathbf{Z}$ can exist for $x \in \{-3, 0, 1\}$. For each x, we substitute into equations (6.11) and (6.12) (already in $\mathbf{Z}[y]$) and get that the integer solutions are

$(-3,1),(0,1),(1,-1)$.

We note that the resultant method can also be applied to solve polynomial equations in several variables, but it is not really effective. One problem is that computational space explosion occurs in the computation of the determinant. Note that computing the resultant of two univariate polynomials in determinant form using the usual Gauss-elimination requires $O((m+n)^3)$ operations, while the variants of the Euclidean algorithm are quadratic. The other problem is that computational complexity depends strongly on the order of the indeterminates. *Eliminating all variables together* in a polynomial equation system is much more effective. This leads to the introduction of multivariate resultants.

6.2.4. Modular greatest common divisor

All methods considered so far for the existence and calculation of common roots of polynomials are characterised by an explosion of computational space. The natural question arises: can we apply modular techniques? Below we examine the case $a(x), b(x) \in \mathbf{Z}[x]$ with $(a, b \neq 0)$. Let us consider the polynomials (6.4), (6.5) $\in \mathbf{Z}[x]$ and let $p = 13$ a prime number. Then the series of remainders in $\mathbf{Z}_p[x]$ in the CLASSICAL-EUCLIDEAN algorithm is

$$\begin{aligned} r_0 &= 11x^5 + 5x^4 + 6x^3 + 6x^2 + 5 \,, \\ r_1 &= x^4 + x^3 + 2x^2 + 8x + 8 \,, \\ r_2 &= 3x^3 + 8x^2 + 12x + 1 \,, \\ r_3 &= x^2 + 10x + 10 \,, \\ r_4 &= 7x \,, \\ r_5 &= 10 \,. \end{aligned}$$

We get that polynomials a and b are relatively prime in $\mathbf{Z}_p[x]$. The following theorem describes the connection between greatest common divisors in $\mathbf{Z}[x]$ and $\mathbf{Z}_p[x]$.

Theorem 6.8 *Let $a, b \in \mathbf{Z}[x], a, b \neq 0$. Let p be a prime such that $p \not| \; lc(a)$ and $p \not| \; lc(b)$. Let furthermore $c = gcd(a,b) \in \mathbf{Z}[x]$, $a_p = a\,remp$, $b_p = b\,remp$ and $c_p = c\,remp$. Then*

(1) $\deg\big(gcd(a_p, b_p)\big) \geq \deg\big(gcd(a,b)\big)$,

(2) if $p \not| \; res(a/c, b/c)$, then $gcd(a_p, b_p) = c_p$.

Proof (1): Since $c_p \mid a_p$ and $c_p \mid b_p$, thus $c_p \mid \gcd(a_p, b_p)$. So

$$\deg\big(\gcd(a_p, b_p)\big) \geq \deg\big(\gcd(a,b) \bmod p\big).$$

By the hypothesis $p \not| \; \mathrm{lc}\big(\gcd(a,b)\big)$, which implies

$$\deg\big(\gcd(a,b) \bmod p\big) = \deg\big(\gcd(a,b)\big) \,.$$

(2): Since $\gcd(a/c, b/c) = 1$ and c_p is non-trivial,

$$\gcd(a_p, b_p) = c_p \cdot \gcd(a_p/c_p, b_p/c_p) \,. \tag{6.13}$$

If $\gcd(a_p, b_p) \neq c_p$, then the right-hand side of (6.13) is non-trivial, thus $\operatorname{res}(a_p/c_p, b_p/c_p) = 0$. But the resultant is the sum of the corresponding products of the coefficients, so $p \mid \operatorname{res}(a/c, b/c)$, contradiction. ■

Corollary 6.9 *There are at most a finite number of primes p such that $p \not| lc(a)$, $p \not| lc(b)$ and $\deg\big(gcd(a_p, b_p)\big) > \deg\big(gcd(a, b)\big)$.* ■

In case statement (1) of Theorem 6.8 is fulfilled, we call p a „lucky prime". We can outline a modular algorithm for the computation of the gcd.

MODULAR-GCD-BIGPRIME(a, b)

```
1  M ← the Landau-Mignotte constant (from Corollary 6.4)
2  H ← {}
3  while TRUE
4        do p ← a prime with p ≥ 2M, p ∉ H, p ∤ lc(a) and p ∤ lc(b)
5           c_p ← gcd(a_p, b_p)
6           if c_p | a and c_p | b
7              then return c_p
8              else H ← H ∪ {p}
```

The first line of the algorithm requires the calculation of the Landau-Mignotte bound. The fourth line requires a „sufficiently large" prime p which does not divide the leading coefficient of a and b. The fifth line computes the greatest common divisor of polynomials a and b modulo p (for example with the CLASSICAL-EUCLIDEAN algorithm in $\mathbf{Z}_p[x]$). We store the coefficients of the resulting polynomials with symmetrical representation. The sixth line examines whether $c_p \mid a$ and $c_p \mid b$ are fulfilled, in which case c_p is the required greatest common divisor. If this is not the case, then p is an „unlucky prime", so we choose another prime. Since, by Theorem 6.8, there are only finitely many „unlucky primes", the algorithm eventually terminates. If the primes are chosen according to a given strategy, set H is not needed.

The disadvantage of the MODULAR-GCD-BIGPRIME algorithm is that the Landau-Mignotte constant grows exponentially in terms of the degree of the input polynomials, so we have to work with large primes. The question is how we could modify the algorithm so that we can work with „many small primes". Since the greatest common divisor in $\mathbf{Z}_p[x]$ is only unique up to a constant factor, we have to be careful with the coefficients of the polynomials in the new algorithm. So, before applying the Chinese remainder theorem for the coefficients of the modular greatest common divisors taken modulo different primes, we have to normalise the leading coefficient of $\gcd(a_p, b_p)$. If a_m and b_n are the leading coefficients of a and b, then the leading coefficient of $\gcd(a, b)$ divides $\gcd(a_m, b_n)$. Therefore, we normalise the leading coefficient of $\gcd(a_p, b_p)$ to $\gcd(a_m, b_n) \bmod p$ in case of primitive polynomials a and b; and finally take the primitive part of the resulting polynomial. Just like in the MODULAR-GCD-BIGPRIME algorithm, modular values are stored with symmetrical representation. These observations lead to the following modular gcd algorithm using small primes.

MODULAR-GCD-SMALLPRIMES(a, b)

```
 1  d ← gcd( lc(a), lc(b))
 2  p ← a prime such that p ∤ d
 3  H ← {p}
 4  P ← p
 5  c_p ← gcd(a_p, b_p)
 6  g_p ← (d mod p) · c_p
 7  (n, i, j) ← (3, 1, 1)
 8  while TRUE
 9        do if j = 1
10              then if deg g_p = 0
11                      then return 1
12              (g, j, P) ← (g_p, 0, p)
13           if n ≤ i
14              then g ← pp(g)
15                   if g | a and g | b
16                      then return g
17           p ← a prime such that p ∤ d and p ∉ H
18           H ← H ∪ {p}
19           c_p ← gcd(a_p, b_p)
20           g_p ← (d mod p) · c_p
21           if deg g_p < deg g
22              then (i, j) ← (1, 1)
23           if j = 0
24              then if deg g_p = deg g
25                      then g_1 = COEFF-BUILD(g, g_p, P, p)
26                           if g_1 = g
27                              then i ← i + 1
28                              else i ← 1
29                           P ← P · p
30                           g ← g_1
```

COEFF-BUILD(a, b, m_1, m_2)

```
1  p ← 0
2  c ← 1/m_1 mod m_2
3  for i ← deg a downto 0
4      do r ← a_i mod m_1
5         s ← (b_i − r) mod m_2
6         p ← p + (r + s · m_1)x^i
7  return p
```

We may note that the algorithm MODULAR-GCD-SMALLPRIMES does not require as many small primes as the Landau–Mignotte bound tells us. When the value of polynomial g does not change during a few iterations, we test in lines 13–16 if g is a greatest common divisor. The number of these iterations is stored in the

variable n of line six. Note that the value of n could vary according to the input polynomial. The primes used in the algorithms could preferably be chosen from an (architecture-dependent) prestored list containing primes that fit in a machine word, so the use of set H becomes unnecessary. Corollary 6.9 implies that the MODULAR-GCD-SMALLPRIMES algorithm always terminates.

The COEFF-BUILD algorithm computes the solution of the equation system obtained by taking congruence relations modulo m_1 and m_2 for the coefficients of identical degree in the input polynomials a and b. This is done according to the Chinese remainder theorem. It is very important to store the results in symmetrical modular representation form.

Example 6.9 Let us examine the operation of the MODULAR-GCD-SMALLPRIMES algorithm for the previously seen polynomials (6.4), (6.5). For simplicity, we calculate with small primes. Recall that

$$\begin{aligned} a(x) &= 63x^5 + 57x^4 - 59x^3 + 45x^2 - 8 \in \mathbf{Z}[x] \,, \\ b(x) &= -77x^4 + 66x^3 + 54x^2 - 5x + 99 \in \mathbf{Z}[x] \,. \end{aligned}$$

After the execution of the first six lines of the algorithm with $p = 5$, we have $d = 7, c_p = x^2 + 3x + 2$ and $g_p = 2x^2 + x - 1$. Since $j = 1$ due to line 7, lines 10–12 are executed. Polynomial g_p is not zero, so $g = 2x^2 + x - 1, j = 0$, and $P = 5$ will be the values after the execution. The condition in line 13 is not fulfilled, so we choose another prime, $p = 7$ is a bad choice, but $p = 11$ is allowed. According to lines 19–20, $c_p = 1$, $g_p = -4$. Since $\deg g_p < \deg g$, we have $j = 1$ and lines 25–30 are not executed. Polynomial g_p is constant, so the return value in line 11 is 1, which means that polynomials a and b are relatively prime.

Example 6.10 In our second example, consider the already discussed polynomials

$$\begin{aligned} a(x) &= 12x^4 - 68x^3 + 52x^2 - 92x + 56 \in \mathbf{Z}[x] \,, \\ b(x) &= -12x^3 + 80x^2 - 84x + 24 \in \mathbf{Z}[x] \,. \end{aligned}$$

Let again $p = 5$. After the first six lines of the polynomials $d = 12, c_p = x + 1, g_p = 2x + 2$. After the execution of lines 10–12, we have $P = 5, g = 2x + 2$. Let the next prime be $p = 7$. So the new values are $c_p = x + 4, g_p = -2x - 1$. Since $\deg g_p = \deg g$, $P = 35$ and the new value of g is $12x - 8$ after lines 25–30. The value of the variable i is still 1. Let the next prime be 11. Then $c_p = g_p = x + 3$. Polynomials g_p and g have the same degree, so we modify the coefficients of g. Then $g_1 = 12x - 8$ and since $g = g_1$, we get $i = 2$ and $P = 385$. Let the new prime be 13. Then $c_p = x + 8, g_p = -x + 5$. The degrees of g_p and g are still equal, thus lines 25–30 are executed and the variables become $g = 12x - 8, P = 4654, i = 3$.

After the execution of lines 17–18, it turns out that $g \mid a$ and $g \mid b$, so $g = 12x - 8$ is the greatest common divisor.

We give the following theorem without proof.

Theorem 6.10 *The* MODULAR-GCD-SMALLPRIMES *algorithm works correctly. The computational complexity of the algorithm is* $O(m^3(\lg m + \lambda(K))^2)$ *machine word operations, where* $m = \min\{\deg a, \deg b\}$*, and* K *is the Landau–Mignotte bound for polynomials* a *and* b*.*

Exercises

6.2-1 Let R be a commutative ring with identity element, $a = \sum_{i=0}^{m} a_i x^i \in R[x]$, $b = \sum_{i=0}^{n} b_i x^i \in R[x]$, furthermore, b_n a unit, $m \geq n \geq 0$. The following algorithm performs Euclidean division for a and b and outputs polynomials $q, r \in R[x]$ for which $a = qb + r$ and $\deg r < n$ or $r = 0$ holds.

EUCLIDEAN-DIVISION-UNIVARIATE-POLYNOMIALS(a, b)

```
1  r ← a
2  for i ← m − n downto 0
3      do if deg r = n + i
4            then q_i ← lc(r)/b_n
5                 r ← r − q_i x^i b
6            else q_i ← 0
7  q ← Σ_{i=0}^{m−n} q_i x^i and r
8  return q
```

Prove that the algorithm uses at most

$$(2 \deg b + 1)(\deg q + 1) = O(m^2)$$

operations in R.

6.2-2 What is the difference between the algorithms EXTENDED-EUCLIDEAN and EXTENDED-EUCLIDEAN-NORMALISED in $\mathbf{Z}[x]$?

6.2-3 Prove that $\operatorname{res}(f \cdot g, h) = \operatorname{res}(f, h) \cdot \operatorname{res}(g, h)$.

6.2-4 The ***discriminant*** of polynomial $f(x) \in R[x]$ ($\deg f = m$, $\operatorname{lc}(f) = f_m$) is the element

$$\operatorname{discr} f = \frac{(-1)^{\frac{m(m-1)}{2}}}{f_m} \operatorname{res}(f, f') \in R \,,$$

where f' denotes the derivative of f with respect to x. Polynomial f has a multiple root if and only if its discriminant is 0. Compute $\operatorname{discr} f$ for general polynomials of second and third degree.

6.3. Gröbner basis

Let F be a field and $R = F[x_1, x_2, \ldots, x_n]$ be a multivariate polynomial ring in n variables over F. Let $f_1, f_2, \ldots, f_s \in R$. First we determine a necessary and sufficient condition for the polynomials $f_1, f_2, \ldots, f_s$ having common roots in R. We can see that the problem is a generalisation of the case $s = 2$ from the previous subsection. Let

$$I = \langle f_1, \ldots, f_s \rangle = \left\{ \sum_{1 \leq i \leq s} q_i f_i : q_i \in R \right\}$$

denote the ***ideal generated by polynomials*** $f_1, \ldots, f_s$. Then the polynomials $f_1, \ldots, f_s$ form a ***basis*** of ideal I. The ***variety*** of an ideal I is the set

$$V(I) = \Big\{ u \in F^n : f(u) = 0 \ \text{ for all } f \in I \Big\}.$$

The knowledge of the variety $V(I)$ means that we also know the common roots of $f_1, \ldots, f_s$. The most important questions about the variety and ideal I are as follows.

- $V(I) \neq \emptyset$?
- How „big" is $V(I)$?
- Given $f \in R$, in which case is $f \in I$?
- $I = R$?

Fortunately, in a special basis of ideal I, in the so-called Gröbner basis, these questions are easy to answer. First let us study the case $n = 1$. Since $F[x]$ is a Euclidean ring,

$$\langle f_1, \ldots, f_s \rangle = \langle \, \gcd(f_1, \ldots, f_s) \, \rangle \, . \tag{6.14}$$

We may assume that $s = 2$. Let $f, g \in F[x]$ and divide f by g with remainder. Then there exist unique polynomials $q, r \in F[x]$ with $f = gq + r$ and $\deg r < \deg g$. Hence,

$$f \in \langle g \rangle \Leftrightarrow r = 0 \, .$$

Moreover, $V(g) = \{u_1, \ldots, u_d\}$ if $x - u_1, \ldots, x - u_d$ are the distinct linear factors of $g \in F[x]$. Unfortunately, equality (6.14) is not true in case of two or more variables. Indeed, a multivariate polynomial ring over an arbitrary field is not necessary Euclidean, therefore we have to find a new interpretation of division with remainder. We proceed in this direction.

6.3.1. Monomial order

Recall that a partial order $\rho \subseteq S \times S$ is a total order (or simply order) if either $a \rho b$ or $b \rho a$ for all $a, b \in S$. The total order '$\preceq$' $\subseteq \mathbb{N}^n$ is ***allowable*** if

(i) $(0, \ldots, 0) \preceq v$ for all $v \in \mathbb{N}^n$,
(ii) $v_1 \preceq v_2 \Rightarrow v_1 + v \preceq v_2 + v$ for all $v_1, v_2, v \in \mathbb{N}^n$.

It is easy to prove that any allowable order on $\mathbb{N}^n$ is a well-order (namely, every nonempty subset of $\mathbb{N}^n$ has a least element). With the notation already adopted consider the set

$$T = \{x_1^{i_1} \cdots x_n^{i_n} \mid i_1, \ldots, i_n \in \mathbb{N}\} \, .$$

The elements of T are called ***monomials***. Observe that T is closed under multiplication in $F[x_1, \ldots, x_n]$, constituting a commutative monoid. The map $\mathbb{N}^n \to T$, $(i_1, \ldots, i_n) \mapsto x_1^{i_1} \cdots x_n^{i_n}$ is an isomorphism, therefore, for an allowable total order $\preceq$ on T, we have that

(i) $1 \preceq t$ for all $t \in T$,
(ii) $\forall\, t_1, t_2, t \in T \;\; t_1 \prec t_2 \Rightarrow t_1 t \prec t_2 t$.

The allowable orders on T are called ***monomial orders***. If $n = 1$, the natural order is a monomial order, and the corresponding univariate monomials are ordered by their degree. Let us see some standard examples of higher degree monomial orders. Let

$$\alpha = x_1^{i_1} \cdots x_n^{i_n},\ \beta = x_1^{j_1} \cdots x_n^{j_n} \in T\ ,$$

where the variables are ordered as $x_1 \succ x_2 \succ \cdots \succ x_{n-1} \succ x_n$.

- ***Pure lexicographic order.***
  $\alpha \prec_{plex} \beta \Leftrightarrow \exists l \in \{1, \ldots, n\}\ i_l < j_l$ and $i_1 = j_1, \ldots, i_{l-1} = j_{l-1}$.
- ***Graded lexicographic order.***
  $\alpha \prec_{grlex} \beta \Leftrightarrow i_1 + \cdots + i_n < j_1 + \cdots + j_n$ or $(i_1 + \cdots + i_n = j_1 + \cdots + j_n$ and $\alpha \prec_{plex} \beta)$.
- ***Graded reverse lexicographic order.***
  $\alpha \prec_{grevlex} \beta \Leftrightarrow i_1 + \cdots + i_n < j_1 + \cdots + j_n$ or $(i_1 + \cdots + i_n = j_1 + \cdots + j_n$ and $\exists l \in \{1, \ldots, n\}\ i_l > j_l$ and $i_{l+1} = j_{l+1}, \ldots, i_n = j_n)$.

The proof that these orders are monomial orders is left as an exercise. Observe that if $n = 1$, then $\prec_{plex} = \prec_{grlex} = \prec_{grevlex}$. The graded reverse lexicographic order is often called a total degree order and it is denoted by $\prec_{tdeg}$.

Example 6.11
Let $\prec = \prec_{plex}$ and let $z \prec y \prec x$. Then

$$\begin{aligned} 1 \;\prec\;& z \prec z^2 \prec \cdots \prec y \prec yz \prec yz^2 \prec \cdots \\ \prec\;& y^2 \prec y^2 z \prec y^2 z^2 \prec \cdots x \prec xz \prec xz^2 \prec \cdots \\ \prec\;& xy \prec xy^2 \prec \cdots \prec x^2 \prec \cdots\ . \end{aligned}$$

Let $\prec = \prec_{tdeg}$ and again, $z \prec y \prec x$. Then

$$\begin{aligned} 1 \;\prec\;& z \prec y \prec x \\ \prec\;& z^2 \prec yz \prec xz \prec y^2 \prec xy \prec x^2 \\ \prec\;& z^3 \prec yz^2 \prec xz^2 \prec y^2 z \prec xyz \\ \prec\;& x^2 z \prec y^3 \prec xy^2 \prec x^2 y \prec x^3 \prec \cdots\ . \end{aligned}$$

Let a monomial order $\prec$ be given. Furthermore, we identify the vector $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{N}^n$ with the monomial $x^\alpha = x_1^{\alpha_1} \cdots x_n^{\alpha_n} \in R$. Let $f = \sum_{\alpha \in \mathbb{N}^n} c_\alpha x^\alpha \in R$ be a non-zero polynomial, $c_\alpha \in F$. Then $c_\alpha x^\alpha$ $(c_\alpha \neq 0)$ are the ***terms*** of polynomial f, $\mathrm{mdeg}(f) = \max\{\alpha \in \mathbb{N}^n : c_\alpha \neq 0\}$ is the ***multidegree*** of the polynomial (where the maximum is with respect to the monomial order), $\mathrm{lc}(f) = c_{\mathrm{mdeg}(f)} \in F \setminus \{0\}$ is the ***leading coefficient*** of f, $\mathrm{lm}(f) = x^{\mathrm{mdeg}(f)} \in R$ is the ***leading monomial*** of f, and $\mathrm{lt}(f) = \mathrm{lc}(f) \cdot \mathrm{lm}(f) \in R$ is the ***leading term*** of f. Let $\mathrm{lt}(0) = \mathrm{lc}(0) = \mathrm{lm}(0) = 0$ and $\mathrm{mdeg}(0) = -\infty$.

Example 6.12 Consider the polynomial $f(x, y, z) = 2xyz^2 - 3x^3 + 4y^4 - 5xy^2 z \in \mathbb{Q}[x, y, z]$.

Let $\prec = \prec_{plex}$ and $z \prec y \prec x$. Then

$$\mathrm{mdeg}(f) = (3,0,0),\ \mathrm{lt}(f) = -3x^3,\ \mathrm{lm}(f) = x^3,\ \mathrm{lc}(f) = -3\ .$$

If $\prec = \prec_{tdeg}$ and $z \prec y \prec x$, then

$$\mathrm{mdeg}(f) = (0,4,0),\ \mathrm{lt}(f) = 4y^4,\ \mathrm{lm}(f) = y^4,\ \mathrm{lc}(f) = 4\ .$$

6.3.2. Multivariate division with remainder

In this subsection, our aim is to give an algorithm for division with remainder in R. Given multivariate polynomials $f, f_1, \ldots, f_s \in R$ and monomial order $\prec$, we want to compute the polynomials $q_1, \ldots, q_s \in R$ and $r \in R$ such that $f = q_1 f_1 + \cdots + q_s f_s + r$ and no monomial in r is divisible by any of $\mathrm{lt}(f_1), \ldots, \mathrm{lt}(f_s)$.

MULTIVARIATE-DIVISION-WITH-REMAINDER$(f, f_1, \ldots, f_s)$

```
1  r ← 0
2  p ← f
3  for i ← 1 to s
4       do q_i ← 0
5  while p ≠ 0
6        do if lt(f_i) divides lt(p) for some i ∈ {1,...,s}
7              then choose such an i and q_i ← q_i + lt(p)/lt · (f_i)
8                   p ← p − lt(p)/lt(f_i) · f_i
9              else r ← r + lt(p)
10                  p ← p − lt(p)
11 return q_1,...,q_s and r
```

The correctness of the algorithm follows from the fact that in every iteration of the **while** cycle of lines 5–10, the following invariants hold

(i) $\mathrm{mdeg}(p) \preceq \mathrm{mdeg}(f)$ and $f = p + q_1 f_1 + \cdots + q_s f_s + r$,
(ii) $q_i \neq 0 \Rightarrow \mathrm{mdeg}(q_i f_i) \preceq \mathrm{mdeg}(f)$ for all $1 \leq i \leq s$,
(iii) no monomial in r is divisible by any $\mathrm{lt}(f_i)$.

The algorithm has a weakness, namely, the multivariate division with remainder is not deterministic. In line 7, we can choose arbitrarily from the appropriate values of i.

Example 6.13 Let $f = x^2y + xy^2 + y^2 \in \mathbb{Q}[x,y]$, $f_1 = xy - 1$, $f_2 = y^2 - 1$, the monomial order $\prec_{plex}$, $y \prec_{plex} x$, and in line 7, we always choose the smallest possible i. Then the result of the algorithm is $q_1 = x + y$, $q_2 = 1$, $r = x + y + 1$. But if we change the order of the functions f_1 and f_2 (that is, $f_1 = y^2 - 1$ and $f_2 = xy - 1$), then the output of the algorithm is $q_1 = x + 1$, $q_2 = x$ and $r = 2x + 1$.

As we have seen in the previous example, we can make the algorithm deterministic by always choosing the smallest possible i in line 7. In this case, the ***quotients*** $q_1, \ldots, q_s$ and the ***remainder*** r are unique, which we can express as $r = f \text{ rem } (f_1, \ldots, f_s)$.

Observe that if $s = 1$, then the algorithm gives the answer to the ideal membership problem: $f \in \langle f_1 \rangle$ if and only if the remainder is zero. Unfortunately, if $s \geq 2$, then this is not true anymore. For example, with the monomial order $\prec_{plex}$

$$xy^2 - x \text{ rem } (xy + 1, y^2 - 1) = -x - y \ ,$$

and the quotients are $q_1 = y$, $q_2 = 0$. On the other hand, $xy^2 - x = x \cdot (y^2 - 1) + 0$, which shows that $xy^2 - x \in \langle xy + 1, y^2 - 1 \rangle$.

6.3.3. Monomial ideals and Hilbert's basis theorem

Our next goal is to find a special basis for an arbitrary polynomial ideal such that the remainder on division by that basis is unique, which gives the answer to the ideal membership problem. But does such a basis exist at all? And if it does, is it finite?

The ideal $I \subseteq R$ is a ***monomial ideal*** if there exists a subset $A \subseteq \mathbb{N}^n$ such that

$$I = \langle x^A \rangle = \langle \{x^\alpha \in T : \alpha \in A\} \rangle \ ,$$

that is, ideal I is generated by monomials.

Lemma 6.11 *Let $I = \langle x^A \rangle \subseteq R$ be a monomial ideal, and $\beta \in \mathbb{N}^n$. Then*

$$x^\beta \in I \Leftrightarrow \exists \alpha \in A \quad x^\alpha \mid x^\beta \ .$$

Proof The $\Leftarrow$ direction is obvious. Conversely, let $\alpha_1, \ldots, \alpha_s \in A$ and $q_1, \ldots, q_s \in R$ such that $x^\beta = \sum_i q_i x^{\alpha_i}$. Then the sum has at least one member $q_i x^{\alpha_i}$ which contains x^β, therefore $x^{\alpha_i} \mid x^\beta$. ∎

The most important consequence of the lemma is that two monomial ideals are identical if and only if they contain the same monomials.

Lemma 6.12 (Dickson's lemma). *Every monomial ideal is finitely generated, namely, for every $A \subseteq \mathbb{N}^n$, there exists a finite subset $B \subseteq A$ such that $\langle x^A \rangle = \langle x^B \rangle$.*

Lemma 6.13 *Let I be an ideal in $R = F[x_1, \ldots, x_n]$. If $G \subseteq I$ is a finite subset such that $\langle lt(G) \rangle = \langle lt(I) \rangle$, then $\langle G \rangle = I$.*

Proof Let $G = \{g_1, \ldots, g_s\}$. If $f \in I$ is an arbitrary polynomial, then division with remainder gives $f = q_1 g_1 + \cdots + q_s g_s + r$, with $q_1, \ldots, q_s, r \in R$, such that either $r = 0$ or no term of r is divisible by the leading term of any g_i. But $r = f - q_1 g_1 - \cdots - q_s g_s \in I$, hence, $\mathrm{lt}(r) \in \mathrm{lt}(I) \subseteq \langle \mathrm{lt}(g_1), \ldots, \mathrm{lt}(g_s) \rangle$. This, together with Lemma (6.11), implies that $r = 0$, therefore $f \in \langle g_1, \ldots, g_s \rangle = \langle G \rangle$. ∎

Together with Dickson's lemma applied to $\langle \mathrm{lt}(I) \rangle$, and the fact that the zero polynomial generates the zero ideal, we obtain the following famous result.

Theorem 6.14 (Hilbert's basis theorem). *Every ideal $I \subseteq R = F[x_1, \ldots, x_n]$ is finitely generated, namely, there exists a finite subset $G \subseteq I$ such that $\langle G \rangle = I$ and $\langle lt(G) \rangle = \langle lt(I) \rangle$.*

Corollary 6.15 (ascending chain condition). *Let $I_1 \subseteq I_2 \subseteq \cdots$ be an ascending chain of ideals in R. Then there exists an $n \in \mathbb{N}$ such that $I_n = I_{n+1} = \cdots$.*

Proof Let $I = \cup_{j \geq 1} I_j$. Then I is an ideal, which is finitely generated by Hilbert's basis theorem. Let $I = \langle g_1, \ldots, g_s \rangle$. With $n = \min\{j \geq 1 : g_1, \ldots, g_s \in I_j\}$, we have $I_n = I_{n+1} = \cdots = I$. ■

A ring satisfying the ascending chain condition is called ***Noetherian***. Specifically, if F is a field, then $F[x_1, \ldots, x_n]$ is Noetherian.

Let $\prec$ be a monomial order on R and $I \subseteq R$ an ideal. A finite set $G \subseteq I$ is a ***Gröbner basis*** of ideal I with respect to $\prec$ if $\langle \mathrm{lt}(G) \rangle = \langle \mathrm{lt}(I) \rangle$. Hilbert's basis theorem implies the following corollary

Corollary 6.16 *Every ideal I in $R = F[x_1, \ldots, x_n]$ has a Gröbner basis.*

It is easy to show that the remainder on division by the Gröbner basis G does not depend on the order of the elements of G. Therefore, we can use the notation $f \text{ rem } G = r \in R$. Using the Gröbner basis, we can easily answer the ideal membership problem.

Theorem 6.17 *Let G be a Gröbner basis of ideal $I \subseteq R$ with respect to a monomial order $\prec$ and let $f \in R$. Then $f \in I \Leftrightarrow f \text{ rem } G = 0$.*

Proof We prove that there exists a unique $r \in R$ such that (1) $f - r \in I$, (2) no term of r is divisible by any monomial of $\mathrm{lt}(G)$. The existence of such an r comes from division with remainder. For the uniqueness, let $f = h_1 + r_1 = h_2 + r_2$ for arbitrary $h_1, h_2 \in I$ and suppose that no term of r_1 or r_2 is divisible by any monomial of $\mathrm{lt}(G)$. Then $r_1 - r_2 = h_2 - h_1 \in I$, and by Lemma 6.11, $\mathrm{lt}(r_1 - r_2)$ is divisible by $\mathrm{lt}(g)$ for some $g \in G$. This means that $r_1 - r_2 = 0$. ■

Thus, if G is a Gröbner basis of R, then for all $f, g, h \in R$,

$$g = f \text{ rem } G \text{ and } h = f \text{ rem } G \Rightarrow g = h.$$

6.3.4. Buchberger's algorithm

Unfortunately, Hilbert's basis theorem is not constructive, since it does not tell us how to compute a Gröbner basis for an ideal I and basis G. In the following, we investigate how the finite set G can fail to be a Gröbner basis for an ideal I.

Let $g, h \in R$ be nonzero polynomials, $\alpha = (\alpha_1, \ldots, \alpha_n) = \mathrm{mdeg}(g)$, $\beta = (\beta_1, \ldots, \beta_n) = \mathrm{mdeg}(h)$, and $\gamma = (\max\{\alpha_1, \beta_1\}, \ldots, \max\{\alpha_n, \beta_n\})$. The ***S-polynomial*** of g and h is

$$S(g, h) = \frac{x^\gamma}{\mathrm{lt}(g)} g - \frac{x^\gamma}{\mathrm{lt}(h)} h \in R.$$

It is easy to see that $S(g, h) = -S(h, g)$, moreover, since $x^\gamma / \mathrm{lt}(g), x^\gamma / \mathrm{lt}(h) \in R$, therefore $S(g, h) \in \langle g, h \rangle$.

The following theorem yields an easy method to test whether a given set G is a Gröbner basis of the ideal $\langle G \rangle$.

Theorem 6.18 *The set $G = \{g_1, \ldots, g_s\} \subseteq R$ is the Gröbner basis of the ideal $\langle G \rangle$ if and only if*

$$S(g_i, g_j) \text{ rem } (g_1, \ldots, g_s) = 0 \quad \text{for all } i \ (1 \leq i < j \leq s) \ .$$

Using the S-polynomials, it is easy to give an algorithm for constructing the Gröbner basis. We present a simplified version of Buchberger's method (1965): given a monomial order $\prec$ and polynomials $f_1, \ldots, f_s \in R = F[x_1, \ldots, x_n]$, the algorithm yields a $G \subseteq R$ Gröbner basis of the ideal $I = \langle f_1, \ldots, f_s \rangle$.

GRÖBNER-BASIS$(f_1, \ldots, f_s)$

```
1  G ← {f_1, ..., f_s}
2  P ← {(f_i, f_j) | f_i, f_j ∈ G, i < j, f_i ≠ f_j}
3  while P ≠ ∅
4        do (f, g) ← an arbitrary pair from P
5           P ← P \ (f, g)
6           r ← S(f, g) rem G
7           if r ≠ 0
8              then G ← G ∪ {r}
9                   P ← P ∪ {(f, r) | f ∈ G}
10 return G
```

First we show the correctness of the GRÖBNER-BASIS algorithm assuming that the procedure terminates. At any stage of the algorithm, set G is a basis of ideal I, since initially it is, and at any other step only those elements are added to G that are remainders of the S-polynomials on division by G. If the algorithm terminates, the remainders of all S-polynomials on division by G are zero, and by Theorem (6.18), G is a Gröbner basis.

Next we show that the algorithm terminates. Let G and G^* be the sets corresponding to successive iterations of the **while** cycle (lines $3-9$). Clearly, $G \subseteq G^*$ and $\langle \mathrm{lt}(G) \rangle \subseteq \langle \mathrm{lc}(G^*) \rangle$. Hence, ideals $\langle \mathrm{lt}(G) \rangle$ in successive iteration steps form an ascending chain, which stabilises by Corollary (6.15). Thus, after a finite number of steps, we have $\langle \mathrm{lt}(G) \rangle = \langle \mathrm{lc}(G^*) \rangle$. We state that $G = G^*$ in this case. Let $f, g \in G$ and $r = S(f, g)$ rem G. Then $r \in G^*$ and either $r = 0$ or $\mathrm{lt}(r) \in \langle \mathrm{lt}(G^*) \rangle = \langle \mathrm{lt}(G) \rangle$, and using the definition of the remainder, we conclude that $r = 0$.

Example 6.14 Let $F = \mathbb{Q}$, $\prec = \prec_{plex}$, $z \prec y \prec x$, $f_1 = x - y - z, f_2 = x + y - z^2, f_3 = x^2 + y^2 - 1$.

$G = \{f_1, f_2, f_3\}$ by step one, and $P = \{(f_1, f_2), (f_1, f_3), (f_2, f_3)\}$ by step two.

At the first iteration of the **while** cycle, let us choose the pair (f_1, f_2). Then $P = \{(f_1, f_3), (f_2, f_3)\}$, $S(f_1, f_2) = -2y - z + z^2$ and $r = f_4 = S(f_1, f_2)$ rem $G = -2y - z + z^2$. Therefore, $G = \{f_1, f_2, f_3, f_4\}$ and $P = \{(f_1, f_3), (f_2, f_3), (f_1, f_4), (f_2, f_4), (f_3, f_4)\}$.

At the second iteration of the **while** cycle, let us choose the pair (f_1, f_3). Then $P = P \setminus \{f_1, f_3\}$, $S(f_1, f_3) = -xy - xz - y^2 + 1$, $r = f_5 = S(f_1, f_3)$ rem $G = -1/2z^4 - 1/2z^2 + 1$, hence, $G = \{f_i \mid 1 \leq i \leq 5\}$ and $P = \{(f_2, f_3), (f_1, f_4), \ldots, (f_3, f_4), (f_1, f_5), \ldots, (f_4, f_5)\}$.

At the third iteration of the **while** cycle, let us choose the pair (f_2, f_3). Then $P = P \setminus \{(f_2, f_3)\}$, $S(f_2, f_3) = xy - xz^2 - y^2 + 1$, $r = S(f_2, f_3)$ rem $G = 0$.

At the fourth iteration, let us choose the pair (f_1, f_4). Then $P = P \setminus \{f_1, f_4\}$, $S(f_1, f_4) =$

$2y^2 + 2yz + xz - xz^2$, $r = S(f_1, f_4)$ rem $G = 0$.

In the same way, the remainder of the S-polynomials of all the remaining pairs on division by G are zero hence, the algorithm returns with $G = \{x - y - z, x + y - z^2, x^2 + y^2 - 1, -2y - z + z^2, -1/2z^4 - 1/2z^2 + 1\}$ which constitutes a Gröbner basis.

6.3.5. Reduced Gröbner basis

In general, the Gröbner basis computed by Buchberger's algorithm is neither unique nor minimal. Fortunately, both can be achieved by a little finesse.

Lemma 6.19 *If G is a Gröbner basis of $I \subseteq R$, $g \in G$ and $lt(g) \in \langle lt(G \setminus \{g\})\rangle$, then $G \setminus \{g\}$ is a Gröbner basis of I as well.*

We say that the set $G \subseteq R$ is a ***minimal*** Gröbner basis for ideal $I = \langle G\rangle$ if it is a Gröbner basis, and for all $g \in G$,

- $\mathrm{lc}(g) = 1$,
- $\mathrm{lt}(g) \notin \langle \mathrm{lt}(G \setminus \{g\})\rangle$.

An element $g \in G$ of a Gröbner basis G is said to be ***reduced*** with respect to G if no monomial of g is in the ideal $\langle \mathrm{lt}(G \setminus \{g\})\rangle$. A minimal Gröbner basis G for $I \subseteq R$ is reduced if all of its elements are reduced with respect to G.

Theorem 6.20 *Every ideal has a unique reduced Gröbner basis.*

Example 6.15 In Example 6.14 not only G but also $G' = \{x-y-z, -2y-z+z^2, -1/2z^4 - 1/2z^2 + 1\}$ is a Gröbner basis. It is not hard to show that $G_r = \{x - 1/2z^2 - 1/2z, y - 1/2z^2 - 1/2z, z^4 + z^2 - z\}$ is a reduced Gröbner basis.

6.3.6. The complexity of computing Gröbner bases

The last forty years (since Buchberger's dissertation) was not enough to clear up entirely the algorithmic complexity of Gröbner basis computation. Implementation experiments show that we are faced with the intermediate expression swell phenomenon. Starting with a few polynomials of low degree and small coefficients, the algorithm produces a large number of polynomials with huge degrees and enormous coefficients. Moreover, in contrast to the variants of the Euclidean algorithm, the explosion cannot be kept under control. In 1996, Kühnle and Mayr gave an exponential space algorithm for computing a reduced Gröbner basis. The polynomial ideal membership problem over $\mathbb{Q}$ is *EXPSPACE*-complete.

Let $f, f_1, \ldots, f_s \in F[x_1, \ldots, x_n]$ be polynomials over a field F with $\deg f_i \leq d$ ($\prec = \prec_{tdeg}$). If $f \in \langle f_1, f_2, \ldots f_s\rangle$, then

$$f = f_1 g_1 + \cdots + f_s g_s$$

for polynomials $g_1, \ldots, g_s \in F[x_1, \ldots, x_n]$ for which their degrees are bounded by $\beta = \beta(n, d) = (2d)^{2^n}$. The double exponential bound is essentially unavoidable,

which is shown by several examples. Unfortunately, in case $F = \mathbb{Q}$, the ideal membership problem falls into this category. Fortunately, in special cases better results are available. If $f = 1$ (Hilbert's famous Nullstellensatz), then in case $d = 2$, the bound is $\beta = 2^{n+1}$, while for $d > 2$, the bound is $\beta = d^n$. But the variety $V(f_1, \ldots, f_s)$ is empty if and only if $1 \in \langle f_1, f_2, \ldots f_s \rangle$, therefore the solvability problem of a polynomial system is in *PSPACE*. Several results state that under specific circumstances, the (general) ideal membership problem is also in *PSPACE*. Such a criterion is for example that $\langle f_1, f_2, \ldots f_s \rangle$ is zero-dimensional (contains finitely many isolated points).

In spite of the exponential complexity, there are many successful stories for the application of Gröbner bases: geometric theorem proving, robot kinematics and motion planning, solving polynomial systems of equations are the most widespread application areas. In the following, we enumerate some topics where the Gröbner basis strategy has been applied successfully.

- *Equivalence of polynomial equations.* Two sets of polynomials generate the same ideal if and only if their Gröbner bases are equal with arbitrary monomial order.
- *Solvability of polynomial equations.* The polynomial system of equations $f_i(x_1, \ldots, x_n) = 0,\ 1 \leq i \leq s$ is solvable if and only if $1 \notin \langle f_1, \ldots, f_s \rangle$.
- *Finitely many solutions of polynomial equations.* The polynomial system of equations $f_i(x_1, \ldots, x_n) = 0,\ 1 \leq i \leq s$ has a finite number of solutions if and only if in any Gröbner basis of $\langle f_1, \ldots, f_s \rangle$ for every variable x_i, there is a polynomial such that its leading term with respect to the chosen monomial order is a power of x_i.
- *The number of solutions.* Suppose that the system of polynomial equations $f_i(x_1, \ldots, x_n) = 0,\ 1 \leq i \leq s$ has a finite number of solutions. Then the number of solutions counted with multiplicityes is equal to the cardinality of the set of monomials that are not multiples of the leading monomials of the polynomials in the Gröbner basis, where any monomial order can be chosen.
- *Simplification of expressions.*

We show an example for the last item.

Example 6.16 Let $a, b, c \in \mathbb{R}$ be given such that

$$a + b + c = 3, \quad a^2 + b^2 + c^2 = 9, \quad a^3 + b^3 + c^3 = 24\ .$$

Compute the value of $a^4 + b^4 + c^4$. So let $f_1 = a + b + c - 3, f_2 = a^2 + b^2 + c^2 - 9$ and $a^3 + b^3 + c^3 - 24$ be elements of $\mathbb{R}[a, b, c]$ and let $\prec = \prec_{plex}$, $c \prec b \prec a$. Then the Gröbner basis of $\langle f_1, f_2, f_3 \rangle$ is

$$G = \{a + b + c - 3, b^2 + c^2 - 3b - 3c + bc, 1 - 3c^2 + c^3\}\ .$$

Since $a^4 + b^4 + c^4 \text{ rem } G = 69$, the answer to the question follows.

Exercises

6.3-1 Prove that the orders $\prec_{plex}, \prec_{grlex}$ and $\prec_{tdeg}$ are monomial orders.

6.3-2 Let $\prec$ be a monomial order on R, $f, g \in R \setminus \{0\}$. Prove the following:

a. $\mathrm{mdeg}(fg) = \mathrm{mdeg}(f) + \mathrm{mdeg}(g)$,

b. if $f + g \neq 0$, then $\mathrm{mdeg}(f + g) \preceq \max\{\mathrm{mdeg}(f), \mathrm{mdeg}(g)\}$, where equality holds if $\mathrm{mdeg}(f) \neq \mathrm{mdeg}(g)$.

6.3-3 Let $f = 2x^4y^2z - 3x^4yz^2 + 4xy^4z^2 - 5xy^2z^4 + 6x^2y^4z - 7x^2yz^4 \in \mathbb{Q}[x, y, z]$.

a. Determine the order of the monomials in f for the monomial orders $\prec_{plex}$, $\prec_{grlex}$ and $\prec_{tdeg}$ with $z \prec y \prec x$ in all cases.

b. For each of the three monomial orders from (*a.*), determine $\mathrm{mdeg}(f)$, $\mathrm{lc}(f)$, $\mathrm{lm}(f)$ and $\mathrm{lt}(f)$.

6.3-4* Prove Dickson's lemma.

6.3-5 Compute the Gröbner basis and the reduced Gröbner basis of the ideal $I = \langle x^2+y-1, xy-x\rangle \subseteq \mathbb{Q}[x, y]$ using the monomial order $\prec = \prec_{lex}$, where $y \prec x$. Which of the following polynomials belong to I: $f_1 = x^2+y^2-y$, $f_2 = 3xy^2-4xy+x+1$?

6.4. Symbolic integration

The problem of indefinite integration is the following: given a function f, find a function g the derivative of which is f, that is, $g'(x) = f(x)$; for this relationship the notation $\int f(x)\,dx = g(x)$ is also used. In introductory calculus courses, one tries to solve indefinite integration problems by different methods, among which one tries to choose in a heuristic way: substitution, trigonometric substitution, integration by parts, etc. Only the integration of rational functions is usually solved by an algorithmic method.

It can be shown that indefinite integration in the general case is algorithmically unsolvable. So we only have the possibility to look for a reasonably large part that can be solved using algorithms.

The first step is the algebraisation of the problem: we discard every analytical concept and consider differentiation as a new (unary) algebraic operation connected to addition and multiplication in a given way, and we try to find the „inverse" of this operation. This approach leads to the introduction of the concept of differential algebra.

The integration routines of computer algebra systems (e.g. MAPLE®), similarly to us, first try a few heuristic methods. Integrals of polynomials (or a bit more generally, finite Laurent-series) are easy to determine. This is followed by a simple table lookup process (e.g. in case of MAPLE 35 basic integrals are used). One can, of course, use integral tables entered from books as well. Next we may look for special cases where appropriate methods are known. For example, for integrands of the form

$$e^{ax+b}\sin(cx+d)\cdot p(x)\ ,$$

where p is a polynomial, the integral can be determined using integration by parts. When the above methods fail, a form of substitution called the „derivative-divides" method is tried: if the integrand is a composite expression, then for every subexpression $f(x)$, we divide by the derivative of f, and check if x vanishes from the result after the substitution $u = f(x)$. Using these simple methods we can determine a surprisingly large number of integrals. To their great advantage, they can solve

simple problems very quickly. If they do not succeed, we try algorithmic methods. The first one is for the integration of rational functions. As we will see, there is a significant difference between the version used in computer algebra systems and the version used in hand computations, since the aims are short running times even in complicated cases, and the simplest possible form of the result. The Risch algorithm for integrating elementary functions is based on the algorithm for the integration of rational functions. We describe the Risch algorithm, but not in full detail. In most cases, we only outline the proofs.

6.4.1. Integration of rational functions

In this subsection, we introduce the notion of differential field and differential extension field, then we describe Hermite's method.

Differential fields Let K be a field of characteristic 0, with a mapping $f \mapsto f'$ of K into itself satisfying:

(1) $(f+g)' = f' + g'$ (additivity);

(2) $(fg)' = f'g + g'f$ (Leibniz-rule).

The mapping $f \mapsto f'$ is called a ***differential operator***, ***differentiation*** or ***derivation***, and K is called a ***differential field***. The set $C = \{c \in K : c' = 0\}$ is the ***field of constants*** or ***constant subfield*** in K. If $f' = g$, we also write $f = \int g$. Obviously, for any constant $c \in C$, we have $f + c = \int g$. The ***logarithmic derivative*** of an element $0 \neq f \in K$ is defined as f'/f (the „derivative of $\log(f)$").

Theorem 6.21 *With the notations of the previous definition, the usual rules of derivation hold:*

(1) $0' = 1' = (-1)' = 0$;

(2) *derivation is C-linear:* $(af + bg)' = af' + bg'$ *for* $f, g \in K$, $a, b \in C$;

(3) *if* $g \neq 0$, f *is arbitrary, then* $(f/g)' = (f'g - g'f)/g^2$;

(4) $(f^n)' = nf'f^{n-1}$ *for* $0 \neq f \in K$ *and* $n \in \mathbf{Z}$;

(5) $\int fg' = fg - \int gf'$ *for* $f, g \in K$ (***integration by parts***).

Example 6.17 (1) With the notations of the previous definition, the mapping $f \mapsto 0$ on K is the *trivial derivation*, for this we have $C = K$.

(2) Let $K = \mathbb{Q}(x)$. There exists a single differential operator on $\mathbb{Q}(x)$ with $x' = 1$, it is the usual differentiation. For this the constants are the elements of $\mathbb{Q}$. Indeed, $n' = 0$ for $n \in \mathbb{N}$ by induction, so the elements of $\mathbf{Z}$ and $\mathbb{Q}$ are also constants. We have by induction that the derivative of power functions is the usual one, thus, by linearity, this is true for polynomials, and by the differentiation rule of quotients, we get the statement. It is not difficult to calculate that for the usual differentiation, the constants are the elements of $\mathbb{Q}$.

(3) If $K = C(x)$, where C is an arbitrary field of characteristic 0, then there exists a single differential operator on K with constant subfield C and $x' = 1$: it is the usual differentiation. This statement is obtained similarly to the previous one.

If C is an arbitrary field of characteristic 0, and $K = C(x)$ with the usual differentiation, then $1/x$ is not the derivative of anything. (The proof of the statement is very much like the proof of the irrationality of $\sqrt{2}$, but we have to work with divisibility by x rather than by 2.)

The example shows that for the integration of $1/x$ and other similar functions, we have to extend the differential field. In order to integrate rational functions, an extension by logarithms will be sufficient.

Extensions of differential fields Let L be a differential field and $K \subset L$ a subfield of L. If differentiation doesn't lead out of K, then we say that K is a ***differential subfield*** of L, and L is a ***differential extension field*** of K. If for some $f, g \in L$ we have $f' = g'/g$, that is, the derivative of f is the logarithmic derivative of g, then we write $f = \log g$. (We note that log, just like $\int$ is a relation rather than a function. In other words, log is an abstract concept here and not a logarithm function to a given base.) If we can choose $g \in K$, we say that f is ***logarithmic over*** K.

Example 6.18 (1) Let $g = x \in K = \mathbb{Q}(x)$, $L = \mathbb{Q}(x, f)$, where f is a new indeterminate, and let $f' = g'/g = 1/x$, that is, $f = \log(x)$. Then $\int 1/x\,dx = \log(x)$.

(2) Analogically,

$$\int \frac{1}{x^2 - 2} = \frac{\sqrt{2}}{4}\log(x - \sqrt{2}) - \frac{\sqrt{2}}{4}\log(x + \sqrt{2})$$

is in the differential field $\mathbb{Q}(\sqrt{2})\ x, \log(x - \sqrt{2}), \log(x + \sqrt{2})$.

(3) Since

$$\int \frac{1}{x^3 + x} = \log(x) - \frac{1}{2}\log(x + i) - \frac{1}{2}\log(x - i) = \log(x) - \frac{1}{2}\log(x^2 + 1)\ ,$$

the integral can be considered as an element of

$$\mathbb{Q}\ x, \log(x), \log(x^2 + 1)$$

or an element of

$$\mathbb{Q}(i)\ x, \log(x), \log(x - i), \log(x + i)$$

as well. Obviously, it is more reasonable to choose the first possibility, because in this case there is no need to extend the base field.

Hermite's method Let K be a field of characteristic 0, $f, g \in K[x]$ non-zero relatively prime polynomials. To compute the integral $\int f/g$ using Hermite's method, we can find polynomials $a, b, c, d \in K[x]$ with

$$\int \frac{f}{g} = \frac{c}{d} + \int \frac{a}{b}\ , \tag{6.15}$$

where $\deg a < \deg b$ and b is monic and square-free. The rational function c/d is called the ***rational part,*** the expression $\int a/b$ is called the ***logarithmic part*** of the integral. The method avoids the factorisation of g into linear factors (in a factor

field or some larger field), and even its decomposition into irreducible factors over K.

Trivially, we may assume that g is monic. By Euclidean division we have $f = pg + h$, where $\deg h < \deg g$, thus, $f/g = p + h/g$. The integration of the polynomial part p is trivial. Let us determine the square-free factorisation of g, that is, find monic and pairwise relatively prime polynomials $g_1, \ldots, g_m \in K[x]$ such that $g_m \neq 1$ and $g = g_1 g_2^2 \cdots g_m^m$. Let us construct the partial fraction decomposition (this can be achieved by the Euclidean algorithm):

$$\frac{h}{g} = \sum_{i=1}^{m} \sum_{j=1}^{i} \frac{h_{i,j}}{g_i^j} \ ,$$

where every $h_{i,j}$ has smaller degree than g_i.

The Hermite-reduction is the iteration of the following step: if $j > 1$, then the integral $\int h_{i,j}/g_i^j$ is reduced to the sum of a rational function and an integral similar to the original one, with j reduced by 1. Using that g_i is square-free, we get $\gcd(g_i, g_i') = 1$, thus, we can obtain polynomials $s, t \in K[x]$ by the application of the extended Euclidean algorithm such that $sg_i + tg_i' = h_{i,j}$ and $\deg s, \deg t < \deg g_i$. Hence, using integration by parts,

$$\begin{aligned} \int \frac{h_{i,j}}{g_i^j} &= \int \frac{t \cdot g_i'}{g_i^j} + \int \frac{s}{g_i^{j-1}} \\ &= \frac{-t}{(j-1)g_i^{j-1}} + \int \frac{t'}{(j-1)g_i^{j-1}} + \int \frac{s}{g_i^{j-1}} \\ &= \frac{-t}{(j-1)g_i^{j-1}} + \int \frac{s + t'/(j-1)}{g_i^{j-1}} \ . \end{aligned}$$

It can be shown that using fast algorithms, if $\deg f, \deg g < n$, then the procedure requires $O\big(M(n) \log n\big)$ operations in the field K, where $M(n)$ is a bound on the number of operations needed to multiply to polynomials of degree at most n.

Hermite's method has a variant that avoids the partial fraction decomposition of h/g. If $m = 1$, then g is square-free. If $m > 1$, then let

$$g^* = g_1 g_2^2 \cdots g_{m-1}^{m-1} = \frac{g}{g_m^m} \ .$$

Since $\gcd(g_m, g^* g_m') = 1$, there exist polynomials $s, t \in K[x]$ such that

$$sg_m + tg^* g_m' = h \ .$$

Dividing both sides by $g = g^* g_m^m$ and integrating by parts,

$$\int \frac{h}{g} = \frac{-t}{(m-1)g_m^{m-1}} + \int \frac{s + g^* t'/(m-1)}{g^* g_m^{m-1}} \ ,$$

thus, m is reduced by one.

Note that a and c can be determined by the method of undetermined coefficients (Horowitz's method). After division, we may assume $\deg f < \deg g$. As it can be

seen from the algorithm, we can choose $d = g_2 g_3^2 \cdots g_m^{m-1}$ and $b = g_1 g_2 \cdots g_m$. Differentiating (6.15), we get a system of linear equations on $\deg b$ coefficients of a and $degd$ coefficients of c, altogether n coefficients. This method in general is not as fast as Hermite's method.

The algorithm below performs the Hermite-reduction for a rational function f/g of variable x.

HERMITE-REDUCTION(f, g)

```
 1  p ← quo(f, g)
 2  h ← rem(f, g)
 3  (g[1], ..., g[m]) ← SQUARE-FREE(g)
 4  construct the partial fraction decomposition of (h/g), compute numerators
    h[i, j] belonging to g[i]^j
 5  rac ← 0
 6  int ← 0
 7  for i ← 1 to m
 8      do int ← int + h[i, 1]/g[i]
 9      for j ← 2 to i
10          do n ← j
11          while n > 1
12                  do determine s and t from the equation s · g[i] + t · g[i]' = h[i, j]
13                  n ← n − 1
14                  rac ← rac − (t/n)/g[i]^n
15                  h[i, n] ← s + t'/n
16          int ← int + h[i, 1]/g[i]
17  red ← rac + ∫ p + ∫ int
18  return red
```

If for some field K of characteristic 0, we want to compute the integral $\int a/b$, where $a, b \in K[x]$ are non-zero relatively prime polynomials with $\deg a < \deg b$, b square-free and monic, we can proceed by decomposing polynomial b into linear factors, $b = \prod_{k=1}^{n}(x - a_k)$, in its splitting field L, then constructing the partial fraction decomposition, $a/b = \sum_{k=1}^{n} c_k/(x - a_k)$, over L, finally integrating we get

$$\int \frac{a}{b} = \sum_{k=1}^{n} c_k \log(x - a_k) \in L\big(x, \log(x - a_1), \ldots, \log(x - a_n)\big) \ .$$

The disadvantage of this method, as we have seen in the example of the function $1/(x^3 + x)$, is that the degree of the extension field L can be too large. An extension degree as large as $n!$ can occur, which leads to totally unmanageable cases. On the other hand, it is not clear either if a field extension is needed at all: for example, in case of the function $1/(x^2 - 2)$, can we not compute the integral without extending the base field? The following theorem enables us to choose the degree of the field extension as small as possible.

Theorem 6.22 (Rothstein-Trager integration algorithm). *Let K be a field of characteristic 0, $a, b \in K[x]$ non-zero relatively prime polynomials,* $\deg a < \deg b$, b

square-free and monic. If L is an algebraic extension of K, $c_1, \ldots, c_k \in L \setminus K$ are square-free and pairwise relatively prime monic polynomials, then the following statements are equivalent:

(1) $$\int \frac{a}{b} = \sum_{i=1}^{k} c_i \log v_i \ ,$$

(2) *The polynomial $r = res_x(b, a - yb') \in K[y]$ can be decomposed into linear factors over L, $c_1, \ldots, c_k$ are exactly the distinct roots of r, and $v_i = gcd(b, a - c_i b')$ if $i = 1, \ldots, k$. Here, res_x is the resultant taken in the indeterminate x.*

Example 6.19 Let us consider again the problem of computing the integral $\int 1/(x^3+x)\,dx$. In this case,

$$r = \mathrm{res}_x\ x^3 + x, 1 - y(3x^2 + 1)\ = -4z^3 + 3y + 1 = -(2y + 1)^2(y - 1)\ ,$$

the roots of which are $c_1 = 1$ and $c_2 = -1/2$. Thus,

$$\begin{aligned} v_1 &= \gcd\ x^3 + x, 1 - (3x^2 + 1)\ = x\ , \\ v_2 &= \gcd\ x^3 + x, 1 + \frac{1}{2}(3x^2 + 1)\ = x^2 + 1\ . \end{aligned}$$

The algorithm, which can be easily given based on the previous theorem, can be slightly improved: instead of computing $v_i = \gcd(b, a - c_i b')$ (by calculations over the field L), v_i can also be computed over K, applying the Extended-Euclidean-Normalised algorithm. This was discovered by Trager, , and independently by Lazard and Rioboo. It is not difficult to show that the running time of the complete integration algorithm obtained this way is $O\big(nM(n)\lg n\big)$ if $\deg f, \deg g < n$.

Theorem 6.23 (Lazard-Rioboo-Trager-formula). *Using the notations of the previous theorem, let e denote the multiplicity of c_i as a root of the polynomial $r = res_x(b, a - yb')$. Then*

(1) $\deg v_i = e$;

(2) *if $w(x, y) \in K(y)[x]$ denotes the remainder of degree e in the* Extended-Euclidean-Normalised *algorithm performed in $K(y)[x]$ on b and $a - yb'$, then $v_i = w(x, c_i)$.*

The algorithm below is the improved Lazard-Rioboo-Trager version of the Rothstein-Trager method. We compute $\int a/b$ for the rational function a/b of indeterminate x, where b is square-free and monic, and $\deg a < \deg b$.

INTEGRATE-LOGARITHMIC-PART(a, b, x)

```
1  Let r(y) ← res_x(b, a − yb′) by the subresultant algorithm, furthermore
2  let w_e(x, y) be the remainder of degree e during the computation
3  (r_1(y), ..., r_k(y)) ←SQUARE-FREE(r(y))
4  int ← 0
5  for i ← 1 to k
6      do if r_i(y) ≠ 1
7            then w(y) ← the gcd of the coefficients of w_i(x, y)
8                 (l(y), s(y), t(y)) ←
                  EXTENDED-EUCLIDEAN-NORMALISED(w(y), r_i(y))
9                 w_i(x, y) ←PRIMITIVE-PART(rem(s(y) · w_i(x, y), r_i(y)))
10                (r_{i,1}, ..., r_{i,k}) ←FACTORS(r_i)
11                for j ← 1 to k
12                    do d ← deg(r_{i,j})
13                       c ←SOLVE(r_{i,j}(y) = 0, y)
14                       if d = 1
15                          then int ← int + c · log(w_i(x, c))
16                          else for n ← 1 to d
17                                  do int ← int + c[n] · log(w_i(x, c[n]))
18 return int
```

Example 6.20 Let us consider again the problem of computing the integral $\int 1/(x^2-2)\,dx$. In this case,

$$r = \mathrm{res}_x(x^2 - 2, 1 - y \cdot 2x) = -8y^2 + 1 \ .$$

The polynomial is irreducible in $\mathbb{Q}[x]$, thus, we cannot avoid the extension of $\mathbb{Q}$. The roots of r are $\pm 1/\sqrt{8}$. From the EXTENDED-EUCLIDEAN-NORMALISED-algorithm over $\mathbb{Q}(y)$, $w_1(x, y) = x - 1/(2y)$, thus, the integral is

$$\int \frac{1}{x^2-2}\,dx = \frac{1}{\sqrt{8}}\log(x - \sqrt{2}) - \frac{1}{\sqrt{8}}\log(x + \sqrt{2}) \ .$$

6.4.2. The Risch integration algorithm

Surprisingly, the methods found for the integration of rational functions can be generalised for the integration of expressions containing basic functions (sin, exp etc.) and their inverse. Computer algebra systems can compute the integral of remarkably complicated functions, but sometimes they fail in seemingly very simple cases, for example the expression $\int x/(1+e^x)\,dx$ is returned unevaluated, or the result contains a special non-elementary function, for example the logarithmic integral. This is due to the fact that in such cases, the integral cannot be given in „closed form".

Although the basic results for integration in „closed form" had been discovered by Liouville in 1833, the corresponding algorithmic methods were only developed by Risch in 1968.

Elementary functions The functions usually referred to as functions in „closed form" are the ones composed of rational functions, exponential functions, logarithmic functions, trigonometric and hyperbolic functions, their inverses and n-th roots (or more generally „inverses" of polynomial functions, that is, solutions of polynomial equations); that is, any nesting of the above functions is also a function in „closed form".

One might note that while $\int 1/(1+x^2)\,dx$ is usually given in the form $\operatorname{arctg}(x)$, the algorithm for the integration of rational functions returns

$$\int \frac{1}{1+x^2}\,dx = \frac{i}{2}\log(x+i) - \frac{i}{2}\log(x-i)$$

as solution. Since trigonometric and hyperbolic functions and their inverses over $\mathbb{C}$ can be expressed in terms of exponentials and logarithms, we can restrict our attention to exponentials and logarithms. Surprisingly, it also turns out that the only extensions needed are logarithms (besides algebraic numbers) in the general case.

Exponential elements Let L be a differential extension field of the differential field K. If for a $\theta \in L$, there exists a $u \in K$ such that $\theta'/\theta = u'$, that is, the logarithmic derivative of θ equals the derivative of an element of K, then we say that θ is ***exponential*** over K and we write $\theta = \exp(u)$. If only the following is true: for an element $\theta \in L$, there is a $u \in K$ such that $\theta'/\theta = u$, that is, the logarithmic derivative of θ is an element of K, then θ is called ***hyperexponential*** over K.

Logarithmic, exponential or hyperexponential elements may be algebraic or transcendent over K.

Elementary extensions Let L be a differential extension field of the differential field K. If

$$L = K(\theta_1, \theta_2, \ldots, \theta_n)\ ,$$

where for $j = 1, 2, \ldots, n$, θ_j is logarithmic, exponential or algebraic over the field

$$K_{j-1} = K(\theta_1, \ldots, \theta_{j-1})$$

($K_0 = K$), then L is called an ***elementary extension*** of K. If for $j = 1, 2, \ldots, n$, θ_j is either transcendental and logarithmic, or transcendental and exponential over K_{j-1}, then L is a ***transcendental elementary extension*** of K.

Let $C(x)$ be the differential field of rational functions with the usual differentiation and constant subfield $\mathbb{C}$. An elementary extension of $C(x)$ is called a field of elementary functions, a transcendental elementary extension of $C(x)$ is called a field of transcendental elementary functions.

Example 6.21 The function $f = \exp(x) + \exp(2x) + \exp(x/2)$ can be written in the form $f = \theta_1 + \theta_2 + \theta_3 \in \mathbb{Q}(x, \theta_1, \theta_2, \theta_3)$, where $\theta_1 = \exp(x)$, $\theta_2 = \exp(2x)$, $\theta_3 = \exp(x/2)$. Trivially, θ_1 is exponential over $\mathbb{Q}(x)$, θ_2 is exponential over $\mathbb{Q}(x, \theta_1)$ and θ_3 is exponential over $\mathbb{Q}(x, \theta_1, \theta_2)$. Since $\theta_2 = \theta_1^2$ and $\mathbb{Q}(x, \theta_1, \theta_2) = \mathbb{Q}(\theta_1)$, f can be written in the simpler form $f = \theta_1 + \theta_1^2 + \theta_3$. The function θ_3 is not only exponential but also algebraic over

$\mathbb{Q}(x,\theta_1)$, since $\theta_3^2-\theta_1=0$, that is, $\theta_3=\theta_1^{1/2}$. So $f=\theta_1+\theta_1^2+\theta_1^{1/2}\in\mathbb{Q}(x,\theta_1,\theta_1^{1/2})$. But f can be put in an even simpler form:

$$f=\theta_3^2+\theta_3^4+\theta_3\in\mathbb{Q}(x,\theta_3)\ .$$

Example 6.22 The function

$$f=\sqrt{\log(x^2+3x+2)\ \log(x+1)+\log(x+2)}$$

can be written in form $f=\theta_4\in\mathbb{Q}(x,\theta_1,\theta_2,\theta_3,\theta_4)$, where $\theta_1=\log(x^2+3x+2)$, $\theta_2=\log(x+1)$, $\theta_3=\log(x+2)$, and θ_4 satisfies the algebraic equation $\theta_4^2-\theta_1(\theta_2+\theta_3)=0$. But f can also be given in the much simpler form $f=\theta_1\in\mathbb{Q}(x,\theta_1)$.

Example 6.23 The function $f=\exp\log(x)/2$ can be written in the form $f=\theta_2\in\mathbb{Q}(x,\theta_1,\theta_2)$, where $\theta_1=\log(x)$ and $\theta_2=\exp(\theta_1/2)$, so θ_1 is logarithmic over $\mathbb{Q}(x)$, and θ_2 is exponential over $\mathbb{Q}(x,\theta_1)$. But $\theta_2^2-x=0$, so θ_2 is algebraic over $\mathbb{Q}(x)$, and $f(x)=x^{1/2}$.

The integration of elementary functions The integral of an element of a field of elementary functions will be completely characterised by Liouville's Principle in case it is an elementary function. Algebraic extensions, however, cause great difficulty if not only the constant field is extended.

Here we only deal with the integration of elements of fields of transcendental elementary functions by the Risch integration algorithm.

In practice, this means an element of the field of transcendental elementary functions $\mathbb{Q}(\alpha_1,\ldots,\alpha_k)(x,\theta_1,\ldots,\theta_n)$, where $\alpha_1,\ldots,\alpha_k$ are algebraic over $\mathbb{Q}$ and the integral is an element of the field

$$\mathbb{Q}(\alpha_1,\ldots,\alpha_k,\ldots,\alpha_{k+h})(x,\theta_1,\ldots,\theta_n,\ldots,\theta_{n+m})$$

of elementary functions. In principle, it would be simpler to choose $\mathbb{C}$ as constant subfield but, as we have seen in the case of rational functions, this is impossible, for we can only compute in an exact way in special fields like algebraic number fields; and we even have to keep the number and degrees of $\alpha_{k+1},\ldots,\alpha_{k+h}$ as small as possible. Nevertheless, we will deal with algebraic extensions of the constant subfield dynamically: we can imagine that the necessary extensions have already been made, while in practice, we only perform extensions when they become necessary.

After the conversion of trigonometric and hyperbolic functions (and their inverses) to exponentials (and logarithms, respectively), the integrand becomes an element of a field of elementary functions. Examples 6.21 and 6.22 show that there are functions that do not seem to be elements of a transcendental elementary extension „at first sight", and yet they are; while Example 6.23 shows that there are functions that seem to be elements of such an extension „at first sight", and yet they are not. The first step is to represent the integrand as an element of a field of transcendental elementary functions using the algebraic relationships between the different exponential and logarithmic functions. We will not consider how this can be done. It can be verified whether we succeeded by the following structure theorem

by Risch. We omit the proof of the theorem. We will need a definition.

An element θ is ***monomial*** over a differential field K if K and $K(\theta)$ have the same constant field, θ is transcendental over K and it is either exponential or logarithmic over K.

Theorem 6.24 (Structure theorem). *Let K be the field of constants and $K_n = K(x, \theta_1, \ldots, \theta_n)$ a differential extension field of $K(x)$, which has constant field K. Let us assume that for all j, either θ_j is algebraic over $K_{j-1} = K(x, \theta_1, \ldots, \theta_{j-1})$, or $\theta_j = w_j$, with $w_j = \log(u_j)$ and $u_j \in K_{j-1}$, or $\theta_j = u_j$, with $u_j = exp(w_j)$ and $w_j \in K_{j-1}$. Then*

1. *$g = \log(f)$, where $f \in K_n \setminus K$, is monomial over K_n if and only if there is no product*
$$f^k \cdot \prod u_j^{k_j}, \quad k, k_j \in \mathbf{Z}, k \neq 0$$
which is an element of K;

2. *$g = exp(f)$, where $f \in K_n \setminus K$, is monomial over K_n if and only if there is no linear combination*
$$f + \sum c_j w_j, \quad c_j \in \mathbb{Q}$$
which is an element of K.

Product and summation is only taken for logarithmic and exponential steps.

The most important classic result of the entire theory is the following theorem.

Theorem 6.25 (Liouville's Principle). *Let K be a differential field with constant field C. Let L be a differential extension field of K with the same constant field. Let us assume that $g' = f \in K$. Then there exist constants $c_1, \ldots, c_m \in C$ and elements $v_0, v_1, \ldots, v_m \in K$ such that*

$$f = v_0' + \sum_{j=1}^{m} c_j \frac{v_j'}{v_j} \ ,$$

that is,

$$g = \int f = v_0 + \sum_{j=1}^{m} c_j \log(v_j) \ .$$

Note that the situation is similar to the case of rational functions.

We will not prove this theorem. Although the proof is lengthy, the idea of the proof is easy to describe. First we show that a transcendental exponential extension cannot be „eliminated", that is differentiating a rational function of it, the new element does not vanish. This is due to the fact that differentiating a polynomial of an element of the transcendental exponential extension, we get a polynomial of the same degree, and the original polynomial does not divide the derivative, except for the case when the original polynomial is monomial. Next we show that no algebraic extension is needed to express the integral. This is essentially due to the fact that

substituting an element of an algebraic extension into its minimal polynomial, we get zero, and differentiating this equation, we get the derivative of the extending element as a rational function of the element. Finally, we have to examine extensions by transcendental logarithmic elements. We show that such an extending element can be eliminated by differentiation if and only if it appears in a linear polynomial with constant leading coefficient. This is due to the fact that differentiating a polynomial of such an extending element, we get a polynomial the degree of which is either the same or is reduced by one, the latter case only being possible when the leading coefficient is constant.

The Risch algorithm Let K be an algebraic number field over $\mathbb{Q}$, and $K_n = K(x, \theta_1, \ldots, \theta_n)$ a field of transcendental elementary functions. The algorithm is recursive in n: using the notation $\theta = \theta_n$, we will integrate a function $f(\theta)/g(\theta) \in K_n = K_{n-1}(\theta)$, where $K_{n-1} = K(x, \theta_1, \ldots, \theta_{n-1})$. (The case $n = 0$ is the integration of rational functions.) We may assume that f and g are relatively prime and g is monic. Besides differentiation with respect to x, we will also use differentiation with respect to θ, which we denote by $d/d\theta$. In the following, we will only present the algorithms.

Risch algorithm: logarithmic case Using the notations of the previous paragraph, we first presume that θ is transcendental and logarithmic, $\theta' = u'/u$, $u \in K_{n-1}$. By Euclidean division, $f(\theta) = p(\theta)g(\theta) + h(\theta)$, hence,

$$\int \frac{f(\theta)}{g(\theta)} = \int p(\theta) + \int \frac{h(\theta)}{g(\theta)} \, .$$

Unlike the integration of rational functions, here the integration of the polynomial part is more difficult. Therefore, we begin with the integration of the rational part.

Logarithmic case, rational part Let $g(\theta) = g_1(\theta) g_2^2(\theta) \cdots g_m^m(\theta)$ be the square-free factorisation of $g(\theta)$. Then

$$\gcd\left(g_j(\theta), \frac{d}{d\theta} g_j(\theta))\right) = 1$$

is obvious. It can be shown that the much stronger condition $\gcd\big(g_j(\theta), g_j(\theta)'\big) = 1$ is also fulfilled. By partial fraction decomposition,

$$\frac{h(\theta)}{g(\theta)} = \sum_{i=1}^{m} \sum_{j=1}^{i} \frac{h_{i,j}(\theta)}{g_i(\theta)^j} \, .$$

We use Hermite-reduction: using the extended Euclidean algorithm we get polynomials $s(\theta), t(\theta) \in K_{n-1}[\theta]$ that satisfy $s(\theta)g_i(\theta) + t(\theta)g_i(\theta)' = h_{i,j}(\theta)$ and $\deg s(\theta), \deg t(\theta) < \deg g_i(\theta)$. Using integration by parts,

$$\int \frac{h_{i,j}(\theta)}{g_i^j(\theta)} \quad = \quad \int \frac{t(\theta) \cdot g_i(\theta)'}{g_i(\theta)^j} + \int \frac{s(\theta)}{g_i(\theta)^{j-1}}$$

$$\begin{aligned} &= \frac{-t(\theta)}{(j-1)g_i(\theta)^{j-1}} + \int \frac{t(\theta)'}{(j-1)g_i(\theta)^{j-1}} + \int \frac{s(\theta)}{g_i(\theta)^{j-1}} \\ &= \frac{-t(\theta)}{(j-1)g_i(\theta)^{j-1}} + \int \frac{s(\theta) + t(\theta)'/(j-1)}{g_i(\theta)^{j-1}} \ . \end{aligned}$$

Continuing this procedure while $j > 1$, we get

$$\int \frac{h(\theta)}{g(\theta)} = \frac{c(\theta)}{d(\theta)} + \int \frac{a(\theta)}{b(\theta)} \ ,$$

where $a(\theta), b(\theta), c(\theta), d(\theta) \in K_{n-1}[\theta]$, $\deg a(\theta) < \deg b(\theta)$ and $b(\theta)$ is a square-free and monic polynomial.

It can be shown that the Rothstein-Trager method can be applied to compute the integral $\int a(\theta)/b(\theta)$. Let us calculate the resultant

$$r(y) = \mathrm{res}\theta\big(b(\theta), a(\theta) - y \cdot b(\theta)'\big) \ .$$

It can be shown that the integral is elementary if and only if $r(y)$ is of the form $r(y) = r(y)s$, where $r(y) \in K[y]$ and $s \in K_{n-1}$. If we compute the primitive part of $r(y)$, choose it as $r(y)$ and any coefficient of $r(y)$ is not a constant, then there is no elementary integral. Otherwise, let $c_1, \ldots, c_k$ be the distinct roots of $r(y)$ in its factor field and let

$$v_i(\theta) = \gcd\big(b(\theta), a(\theta) - c_i b(\theta)'\big) \in K_{n-1}(c_1, \ldots, c_k)[\theta]$$

for $i = 1, \ldots, k$. It can be shown that

$$\int \frac{a(\theta)}{b(\theta)} = \sum_{i=1}^{k} c_i \log\big(v_i(\theta)\big) \ .$$

Let us consider a few examples.

Example 6.24 The integrand of the integral $\int 1/\log(x)$ is $1/\theta \in \mathbb{Q}(x, \theta)$, where $\theta = \log(x)$. Since

$$r(y) = \mathrm{res}\theta(\theta, 1 - y/x) = 1 - y/x \in \mathbb{Q}(x)[y]$$

is a primitive polynomial and it has a coefficient that is, not constant, the integral is not elementary.

Example 6.25 The integrand of the integral $\int 1/\ x\log(x)$ is $1/(x\theta) \in \mathbb{Q}(x, \theta)$, where $\theta = \log(x)$. Here,

$$r(y) = \mathrm{res}\theta(\theta, 1/x - y/x) = 1/x - y/x \in \mathbb{Q}(x)[y],$$

which has primitive part $1 - y$. Every coefficient of this is constant, so the integral is elementary, $c_1 = 1$, $v_1(\theta) = \gcd(\theta, 1/x - 1/x) = \theta$, so

$$\int \frac{1}{x\log(x)} = c_1 \log\ v_1(\theta) \ = \log\ \log(x) \ .$$

Logarithmic case, polynomial part The remaining problem is the integration of the polynomial part

$$p(\theta) = p_k\theta^k + p_{k-1}\theta^{k-1} + \cdots + p_0 \in K_{n-1}[\theta] \ .$$

According to Liouville's Principle $\int p(\theta)$ is elementary if and only if

$$p(\theta) = v_0(\theta)' + \sum_{j=1}^{k} c_j \frac{v_j(\theta)'}{v_j(\theta)} \ ,$$

where $c_j \in K$ and $v_i \in K_{n-1}(\theta)$ for $j = 0, 1, \ldots, m$, $K_{\mathbb{C}}$ is an extension of K and $K_{n-1} = K(x, \theta_1, \ldots, \theta_{n-1})$. We will show that K can be an algebraic extension of K. A similar reasoning to the proof of Liouville's Principle shows that $v_0(\theta) \in K_{n-1}[\theta]$ and $v_j(\theta) \in K_{n-1}$ (that is, independent of θ) for $j = 1, 2, \ldots, m$. Thus,

$$p(\theta) = v_0(\theta)' + \sum_{j=1}^{m} c_j \frac{v_j'}{v_j} \ .$$

We also get, by the reasoning used in the proof of Liouville's Principle, that the degree of $v_0(\theta)$ is at most $k+1$. So if $v_0(\theta) = q_{k+1}\theta^{k+1} + q_k\theta^k + \cdots + q_0$, then

$$p_k\theta^k + p_{k-1}\theta^{k-1} + \cdots + p_0 = (q_{k+1}\theta^{k+1} + q_k\theta^k + \cdots + q_0)' + \sum_{j=1}^{m} c_j \frac{v_j'}{v_j} \ .$$

Hence, we get the following system of equations:

$$\begin{aligned} 0 &= q_{k+1}' \ , \\ p_k &= (k+1)q_{k+1}\theta' + q_k' \ , \\ p_{k-1} &= kq_k\theta' + q_{k-1}' \ , \\ &\vdots \\ p_1 &= 2q_2\theta' + q_1' \ , \\ p_0 &= q_1\theta' + q_0' \ , \end{aligned}$$

where $q_0 = q_0 + \sum_{j=1}^{m} c_j \log(v_j)$ in the last equation. The solution of the first equation is simply a constant b_{k+1}. Substituting this into the next equation and integrating both sides, we get

$$\int p_k = (k+1)b_{k+1} \cdot \theta + q_k \ .$$

Applying the integration procedure recursively, the integral of $p_k \in K_{n-1}$ can be computed, but this equation can only be solved if the integral is elementary, it uses at most one logarithmic extension, and it is exactly $\theta = \log(u)$. If this is not fulfilled, then $\int p(\theta)$ cannot be elementary. If it is fulfilled, then $\int p_k = c_k\theta + d_k$ for some $c_k \in K$ and $d_k \in K_{n-1}$, hence, $b_{k+1} = c_{k+1}/(k+1) \in K$ and $q_k = d_k + b_k$ with an arbitrary integration constant b_k. Substituting for q_k into the next equation and rearranging, we get

$$p_{k-1} - kd_k\theta' = kb_k\theta' + q_{k-1}' \ ,$$

so we have

$$\int \left(p_{k-1} - kd_k \frac{u'}{u}\right) = kb_k\theta + q_{k-1}$$

after integration. The integrand on the right hand side is in K_{n-1}, so we can call the integration procedure in a recursive way. Just like above, the equation can only be solved when the integral is elementary, it uses at most one logarithmic extension and it is exactly $\theta = \log(u)$. Let us assume that this is the case and

$$\int \left(p_{k-1} - kd_k \frac{u'}{u}\right) = c_{k-1}\theta + d_{k-1} \ ,$$

where $c_{k-1} \in K$ and $d_{k-1} \in K_{n-1}$. Then the solution is $b_k = c_{k-1}/k \in K$ and $q_{k-1} = d_{k-1} + b_{k-1}$, where b_{k-1} is an arbitrary integration constant. Continuing the procedure, the solution of the penultimate equation is $b_2 = c_1/2 \in K$ and $q_1 = d_1 + b_1$ with an integration constant b_1. Substituting for q_1 into the last equation, after rearrangement and integration, we get

$$\int \left(p_0 - d_1 \frac{u'}{u}\right) = b_1\theta + q_0 \ .$$

This time the only condition is that the integral should be an elementary function. If it is elementary, say

$$\int \left(p_0 - d_1 \frac{u'}{u}\right) = d_0 \in K_{n-1} \ ,$$

then $b_1 \in K$ is the coefficient of $\theta = \log(u)$ in d_0, $q_0 = d_0 - b_1 \log(u)$, and the result is

$$\int p(\theta) = b_{k+1}\theta^{k+1} + q_k\theta^k + \cdots + q_1\theta + q_0 \ .$$

Let us consider a few examples.

Example 6.26 The integrand of the integral $\int \log(x)$ is $\theta \in \mathbb{Q}(x, \theta)$, where $\theta = \log(x)$. If the integral is elementary, then

$$\int \theta = b_2\theta^2 + q_1\theta + q_0$$

and $0 = b_2'$, $1 = 2b_2\theta' + q_1'$, $0 = q_1\theta' + q_0'$. With the unknown constant b_2 from the second equation, $\int 1 = 2b_2\theta + q_1$. Since $\int 1 = x + b_1$, we get $b_2 = 0$, $q_1 = x + b_1$. From the third equation $-x\theta' = b_1\theta' + q_0'$. Since $\theta' = 1/x$, after integration $\int -1 = b_1\theta + q_0$ and $\int -1 = -x$, we get $b_1 = 0$, $q_0 = -x$, hence, $\int \log(x) = x\log(x) - x$.

Example 6.27 The integrand of the integral $\int \log\log(x)$ is $\theta_2 \in \mathbb{Q}(x, \theta_1, \theta_2)$, where $\theta_1 = \log(x)$ and $\theta_2 = \log(\theta_1)$. If the integral is elementary, then

$$\int \theta_2 = b_2\theta_2^2 + q_1\theta_2 + q_0$$

and $0 = b_2'$, $1 = 2b_2\theta_2' + q_1'$, $0 = q_1\theta_2' + q_0'$. With the unknown constant b_2 from the second equation, $\int 1 = 2b_2\theta + q_1$. Since $\int 1 = x + b_1$, we get $b_2 = 0$, $q_1 = x + b_1$. From the third

equation $-x\theta_2' = b_1\theta_2' + q_0'$. Since $\theta_2' = \theta_1'/\theta_1 = 1/\ x\log(x)\ $, the equation

$$\int \frac{-1}{\log(x)} = b_1\theta_2 + q_0$$

must hold but we know from Example 6.24 that the integral on the left hand side is not elementary.

Risch algorithm: exponential case Now we assume that θ is transcendental and exponential, $\theta'/\theta = u'$, $u \in K_{n-1}$. By Euclidean division, $f(\theta) = q(\theta)g(\theta)+h(\theta)$, hence

$$\int \frac{f(\theta)}{g(\theta)} = \int q(\theta) + \int \frac{h(\theta)}{g(\theta)}\ .$$

We plan using Hermite's method for the rational part. But we have to face an unpleasant surprise: although for the square-free factors $g_j(\theta)$

$$\gcd\left(g_j(\theta), \frac{d}{d\theta}g_j(\theta)\right) = 1$$

is obviously satisfied, the much stronger condition $\gcd\big(g_j(\theta), g_j(\theta)'\big) = 1$ is not. For example, if $g_j(\theta) = \theta$, then

$$\gcd\big(g_j(\theta), g_j(\theta)'\big) = \gcd(\theta, u'\theta) = \theta\ .$$

It can be shown, however, that this unpleasant phenomenon does not appear if $\theta \not| g_j(\theta)$, in which case $\gcd\big(g_j(\theta), g_j(\theta)'\big) = 1$. Thus, it will be sufficient to eliminate θ from the denominator. Let $g(\theta) = \theta^\ell g(\theta)$, where $\theta \not| g(\theta)$, and let us look for polynomials $h(\theta), s(\theta) \in K_{n-1}[\theta]$ with $h(\theta)\theta^\ell + t(\theta)g(\theta) = h(\theta)$, $\deg h(\theta) < \deg g(\theta)$ and $\deg s(\theta) < \ell$. Dividing both sides by $g(\theta)$, we get

$$\frac{f(\theta)}{g(\theta)} = q(\theta) + \frac{t(\theta)}{\theta^l} + \frac{h(\theta)}{g(\theta)}\ .$$

Using the notation $p(\theta) = q(\theta)+t(\theta)/\theta^l$, $p(\theta)$ is a finite Laurent-series the integration of which will be no harder than the integration of a polynomial. This is not surprising if we note $\theta^{-1} = \exp(-u)$. Even so, the integration of the „polynomial part" is more difficult here, too. We start with the other one.

Exponential case, rational part Let $g(\theta) = g_1(\theta)g_2^2(\theta)\cdots g_m^m(\theta)$ be the square-free factorisation of $g(\theta)$. Then, since $\theta \not| g_j(\theta)$, $\gcd\big(g_j(\theta), g_j(\theta)'\big) = 1$. Using partial fraction decomposition

$$\frac{h(\theta)}{g(\theta)} = \sum_{i=1}^{m}\sum_{j=1}^{i}\frac{h_{i,j}(\theta)}{g_i(\theta)^j}\ .$$

Hermite-reduction goes the same way as in the logarithmic case. We get

$$\int \frac{h(\theta)}{g(\theta)} = \frac{c(\theta)}{d(\theta)} + \int \frac{a(\theta)}{b(\theta)}\ ,$$

where $a(\theta), b(\theta), c(\theta), d(\theta) \in K_{n-1}[\theta]$, $\deg a(\theta) < \deg b(\theta)$ and $b(\theta)$ is a square-free and monic polynomial, $\theta \not| b(\theta)$.

It can be shown that the Rothstein-Trager method can be applied to compute the integral $\int a(\theta)/b(\theta)$. Let us calculate the resultant

$$r(y) = \mathrm{res}\theta\big(b(\theta), a(\theta) - y \cdot b(\theta)'\big) \ .$$

It can be shown that the integral is elementary if and only if $r(y)$ is of form $r(y) = r(y)s$, where $r(y) \in K[y]$ and $s \in K_{n-1}$. If we compute the primitive part of $r(y)$, choose it as $r(y)$ and any coefficient of $r(y)$ is not a constant, then there is no elementary integral. Otherwise, let $c_1, \ldots, c_k$ be the distinct roots of $r(y)$ in its factor field and let

$$v_i(\theta) = \gcd\big(b(\theta), a(\theta) - c_i b(\theta)'\big) \in K_{n-1}(c_1, \ldots, c_k)[\theta]$$

for $i = 1, \ldots, k$. It can be shown that

$$\int \frac{a(\theta)}{b(\theta)} = -\left(\sum_{i=1}^{k} c_i \deg v_i(\theta)\right) + \sum_{i=1}^{k} c_i \log\big(v_i(\theta)\big) \ .$$

Let us consider a few examples.

Example 6.28 The integrand of the integral $\int 1/\ 1+\exp(x)$ is $1/(1+\theta) \in \mathbb{Q}(x, \theta)$, where $\theta = \exp(x)$. Since

$$r(y) = \mathrm{res}\theta(\theta + 1, 1 - y\theta) = -1 - y \in \mathbb{Q}(x)[y]$$

is a primitive polynomial which only has constant coefficients, the integral is elementary, $c_1 = -1$, $v_1(\theta) = \gcd(\theta + 1, 1 + \theta) = 1 + \theta$, thus,

$$\int \frac{1}{1 + \exp(x)} = -c_1 x \deg v_1(\theta) + c_1 \log\ v_1(\theta)\ = x - \log\ \exp(x) + 1 \quad .$$

Example 6.29 The integrand of the integral $\int x/\ 1 + \exp(x)$ is $x/(1 + \theta) \in \mathbb{Q}(x, \theta)$, where $\theta = \exp(x)$. Since

$$r(y) = \mathrm{res}\theta(\theta + 1, x - y\theta) = -x - y \in \mathbb{Q}(x)[y]$$

is a primitive polynomial that has a non-constant coefficient, the integral is not elementary.

Exponential case, polynomial part The remaining problem is the integration of the „polynomial part"

$$p(\theta) = \sum_{i=-\ell}^{k} p_i \theta^i \in K_{n-1}(\theta) \ .$$

According to Liouville's Principle $\int p(\theta)$ is elementary if and only if

$$p(\theta) = v_0(\theta)' + \sum_{j=1}^{m} c_j \frac{v_j(\theta)'}{v_j(\theta)} \ ,$$

where $c_j \in K$ and $v_j \in K_{n-1}(\theta)$ for $j = 0, 1, \ldots, m$, $K_{\mathbb{C}}$ is an extension of K and $K_{n-1} = K(x, \theta_1, \ldots, \theta_{n-1})$. It can be shown that K can be an algebraic extension of K. A similar reasoning to the proof of Liouville's Principle shows that we may assume without breaking generality that $v_j(\theta)$ is either an element of K_{n-1} (that is, independent of θ), or it is monic and irreducible in $K_{n-1}[\theta]$ for $j = 1, 2, \ldots, m$. Furthermore, it can be shown that there can be no non-monomial factor in the denominator of $v_0(\theta)$, since such a factor would also be present in the derivative. Similarly, the denominator of $v_j(\theta)$ $(j = 1, 2, \ldots, m)$ can have no non-monomial factor either. So we get that either $v_j(\theta) \in K_{n-1}$, or $v_j(\theta) = \theta$, since this is the only irreducible monic monomial. But if $v_j(\theta) = \theta$, then the corresponding term of the sum is $c_j v_j'(\theta)/v_j(\theta) = c_j u'$, which can be incorporated into $v_0(\theta)'$. Hence, we get that if $p(\theta)$ has an elementary integral, then

$$p(\theta) = \left(\sum_{j=-\ell}^{k} q_j \theta^j \right)' + \sum_{j=1}^{m} c_j \frac{v_j'}{v_j} \ ,$$

where $q_j, v_j \in K_{n-1}$ and $c_j \in K$. The summation should be taken over the same range as in $p(\theta)$ since

$$(q_j \theta^j)' = (q_j' + j u' g_j) \theta^j \ .$$

Comparing the coefficients, we get the system

$$\begin{array}{rcl} p_j & = & q_j' + j u' q_j, \quad \text{ha } -\ell \le j \le k,\ j \ne 0 \ , \\ p_0 & = & q_0' \ , \end{array}$$

where $q_0 = q_0 + \sum_{j=1}^{m} c_j \log(v_j)$. The solution of the equation $p_0 = q_0'$ is simply $q_0 = \int p_0$; if this integral is not elementary, then $\int p(\theta)$ cannot be elementary either, but if it is, then we have determined q_0. In the case $j \ne 0$, we have to solve a differential equation called ***Risch differential equation*** to determine q_j. The differential equation is of the form $y' + fy = g$, where the given functions f and g are elements of K_{n-1}, and we are looking for solutions in K_{n-1}. At first sight it looks as if we had replaced the problem of integration with a more difficult problem, but the linearity of the equations and that the solution has to be in K_{n-1} means a great facility. If any Risch differential equation fails to have a solution in K_{n-1}, then $\int p(\theta)$ is not elementary, otherwise

$$\int p(\theta) = \sum_{j \ne 0} q_j \theta^j + q_0 \ .$$

The Risch differential equation can be solved algorithmically, but we will not go into details.

Let us consider a few examples.

Example 6.30 The integrand of the integral $\int \exp(-x^2)$ is $\theta \in \mathbb{Q}(x, \theta)$, where $\theta = \exp(-x^2)$. If the integral is elementary, then $\int \theta = q_1 \theta$, where $q_1 \in \mathbb{C}(x)$. It is not difficult to show that the differential equation has no rational solutions, so $\int \exp(-x^2)$ is not elementary.

Example 6.31 The integrand of the integral $\int x^x$ is $\exp(x\log(x)) = \theta_2 \in \mathbb{Q}(x, \theta_1, \theta_2)$, where $\theta_1 = \log(x)$ and $\theta_2 = \exp(x\theta_1)$. If the integral is elementary, then $\int \theta_2 = q_1\theta_2$, where $q_1 \in \mathbb{C}(x, \theta_1)$. Differentiating both sides, $\theta_2 = q_1'\theta_2 + q_1(\theta_1+1)\theta_2$, thus, $1 = q_1' + (\theta_1+1)q_1$. Since θ_1 is transcendental over $\mathbb{C}(x)$, by comparing the coefficients $1 = q_1' + q_1$ and $0 = q_1$, which has no solutions. Therefore, $\int x^x$ is not elementary.

Example 6.32 The integrand of the integral

$$\int \frac{(4x^2+4x-1)\ \exp(x^2)+1}{(x+1)^2}\quad \exp(x^2)-1$$

is

$$f(\theta) = \frac{4x^2+4x-1}{(x+1)^2}(\theta^2-1) \in \mathbb{Q}(x,\theta)\ ,$$

where $\theta = \exp(x^2)$. If the integral is elementary, then it is of the form $\int f(\theta) = q_2\theta^2 + q_0$, where

$$\begin{aligned} q_2' + 4xq_2 &= \frac{4x^2+4x-1}{(x+1)^2}\ , \\ q_0' &= -\frac{4x^2+4x-1}{(x+1)^2}\ . \end{aligned}$$

The second equation can be integrated and q_0 is elementary. The solution of the first equation is $q_2 = 1/(1+x)$. Hence,

$$\int f(\theta) = \frac{1}{x+1}\exp^2(x^2) - \frac{(2x+1)^2}{x+1} + 4\log(x+1)\ .$$

Exercises

6.4-1 Apply Hermite-reduction to the following function $f(x) \in \mathbb{Q}(x)$:

$$f(x) = \frac{441x^7+780x^6-286x^5+4085x^4+769x^3+3713x^2-43253x+24500}{9x^6+6x^5-65x^4+20x^3+135x^2-154x+49}\ .$$

6.4-2 Compute the integral $\int f$, where

$$f(x) = \frac{36x^6+126x^5+183x^4+13807/6x^3-407x^2-3242/5x+3044/15}{(x^2+7/6x+1/3)^2(x-2/5)^3} \in \mathbb{Q}(x)\ .$$

6.4-3 Apply the Risch integration algorithm to compute the following integral:

$$\int \frac{x(x+1)\{(x^2e^{2x^2}-\log(x+1)^2)+2xe^{3x^2}(x-(2x^3+2x^2+x+1)\log(x+1))\}}{((x+1)\log^2(x+1)-(x^3+x^2)e^{2x^2})^2}dx\ .$$

6.5. Theory and practice

So far in the chapter we tried to illustrate the algorithm design problems of computer algebra through the presentation of a few important symbolic algorithms. Below the interested reader will find an overview of the wider universe of the research of symbolic algorithms.

6.5.1. Other symbolic algorithms

Besides the resultant method and the theory of Gröbner-bases presented in this chapter, there also exist algorithms for finding *real* symbolic roots of non-linear equations and inequalities. (Collins).

There are some remarkable algorithms in the area of symbolic solution of differential equations. There exists a decision procedure similar to the Risch algorithm for the computation of solutions in closed form of a homogeneous ordinary differential equation of second degree with rational function coefficients. In case of higher degree linear equations, Abramov's procedure gives closed rational solutions of an equation with polynomial coefficients, while Bronstein's algorithm gives solutions of the form $\exp(\int f(x)dx)$. In the case of partial differential equations Lie's symmetry methods can be used. There also exists an algorithm for the factorisation of linear differential operators over formal power series and rational functions.

Procedures based on factorisation are of great importance in the research of computer algebra algorithms. They are so important that many consider the birth of the entire research field with Berlekamp's publication on an effective algorithm for the factorisation of polynomials of one variable over finite fields of small characteristic p. Later, Berlekamp extended his results for larger characteristic. In order to have similarly good running times, he introduced probabilistic elements into the algorithm. Today's computer algebra systems use Berlekamp's procedure even for large finite fields as a routine, perhaps without most of the users knowing about the probabilistic origin of the algorithm. The method will be presented in another chapter of the book. We note that there are numerous algorithms for the factorisation of polynomials over finite fields.

Not much time after polynomial factorisation over finite fields was solved, Zassenhaus, taking van der Waerden's book *Moderne Algebra* from 1936 as a base, used Hensel's lemma for the arithmetic of p-adic numbers to extend factorisation. „Hensel-lifting" – as his procedure is now called – is a general approach for the reconstruction of factors from their modular images. Unlike interpolation, which needs multiple points from the image, Hensel-lifting only needs one point from the image. The Berlekamp–Zassenhaus-algorithm for the factorisation of polynomials with integer coefficients is of fundamental importance but it has two hidden pitfalls. First, for a certain type of polynomial the running time is exponential. Unfortunately, many „bad" polynomials appear in the case of factorisation over algebraic number fields. Second, a representation problem appears for multivariable polynomials, similar to what we have encountered at the Gauss-elimination of sparse matrices. The first problem was solved by a Diophantine optimisation based on the geometry of numbers, a so-called lattice reduction algorithm by Lenstra-Lenstra-Lovász [149]; it is used together with Berlekamp's method. This polynomial algorithm is completed by a procedure which ensures that the Hensel-lifting will start from a „good" modular image and that it will end „in time". Solutions have been found for the mentioned representation problem of the factorisation of multivariable polynomials as well. This is the second area where randomisation plays a crucial role in the design of effective algorithms. We note that in practice, the Berlekamp-Zassenhaus-Hensel-algorithm proves more effective than the Lenstra-Lenstra-Lovász-procedure. As a contrast, the

problem of polynomial factorisation can be solved in polynomial time, while the best proved algorithmic bound for the factorisation of the integer N is $\widetilde{O}(N^{1/4})$ (Pollard and Strassen) in the deterministic case and $L(N)^{1+o(1)}$ (Lenstra and Pomerance) in the probabilistic case, where $L(N) = e^{\sqrt{\ln N \ln \ln N}}$.

In fact, a new theory of heuristic or probabilistic methods in computer algebra is being born to avoid computational or memory explosion and to make algorithms with deterministically large running times more effective. In the case of probabilistic algorithms, the probability of inappropriate operations can be positive, which may result in an incorrect answer (Monte Carlo algorithms) or – although we always get the correct answer (Las Vegas algorithms) – we may not get anything in polynomial time. Beside the above examples, nice results have been achieved in testing polynomial identity, irreducibility of polynomials, determining matrix normal forms (Frobenius, Hilbert, Smith), etc. Their role is likely to increase in the future.

So far in the chapter we gave an overview of the most important symbolic algorithms. We mentioned in the introduction that most computer algebra systems are also able to perform numeric computations: unlike traditional systems, the precision can be set by the user. In many cases, it is useful to combine the symbolic and numeric computations. Let us consider for example the symbolically computed power series solution of a differential equation. After truncation, evaluating the power series with the usual floating point arithmetics in certain points, we get a numerical approximation of the solution. When the problem is an approximation of a physical problem, the attractivity of the symbolic computation is often lost, simply because they are too complicated or too slow and they are not necessary or useful, since we are looking for a numerical solution. In other cases, when the problem cannot be dealt with using symbolic computation, the only way is the numerical approximation. This may be the case when the existing symbolic algorithm does not find a closed solution (e.g. the integral of non-elementary functions, etc.), or when a symbolic algorithm for the specified problem does not exist. Although more and more numerical algorithms have symbolic equivalents, numerical procedures play an important role in computer algebra. Let us think of differentiation and integration: sometimes traditional algorithms – integral transformation, power series approximation, perturbation methods – can be the most effective.

In the design of computer algebra algorithms, parallel architectures will play an increasing role in the future. Although many existing algorithms can be parallelised easily, it is not obvious that good sequential algorithms will perform optimally on parallel architectures as well: the optimal performance might be achieved by a completely different method.

6.5.2. An overview of computer algebra systems

The development of computer algebra systems is linked with the development of computer science and algorithmic mathematics. In the early period of computers, researchers of different fields began the development of the first computer algebra systems to facilitate and accelerate their symbolic computations; these systems, reconstructed and continuously updated, are present in their manieth versions today. ***General purpose*** computer algebra systems appeared in the seventies, and pro-

vide a wide range of built-in data structures, mathematical functions and algorithms, trying to cover a wide area of users. Because of their large need of computational resources, their expansion became explosive in the beginning of the eighties when microprocessor-based workstations appeared. Better hardware environments, more effective resource management, the use of system-independent high-level languages and, last but not least social-economic demands gradually transformed general purpose computer algebra systems into market products, which also resulted in a better user interface and document preparation.

Below we list the most widely known general and special purpose computer algebra systems and libraries.

- General purpose computer algebra systems: Axiom, Derive, Form, GNUcalc, Jacal, Macsyma, Maxima, Maple, Distributed Maple, MathCAD, Matlab Symbolic Math Toolbox, Scilab, MAS, Mathematica, Mathview, Mock-Mma, MuPAD, Reduce, Risa.
- Algebra and number theory: Bergman, CoCoA, Felix, Fermat, GRB, Kan, Macaulay, Magma, Numbers, Pari, Simath, Singular.
- Algebraic geometry: Casa, Ganith.
- Group theory: Gap, LiE, Magma, Schur.
- Tensor analysis: Cartan, FeynCalc, GRG, GRTensor, MathTensor, RedTen, Ricci, TTC.
- Computer algebra libraries: Apfloat, BigNum, GNU MP, Kant, LiDiA, NTL, Saclib, Ubasic, Weyl, Zen.

Most general purpose computer algebra systems are characterised by

- interactivity,
- knowledge of mathematical facts,
- a high-level, declarative[2] programming language with the possibility of functional programming and the knowledge of mathematical objects,
- expansibility towards the operational system and other programs,
- integration of symbolic and numeric computations,
- automatic (optimised) C and Fortran code generation,
- graphical user interface,
- 2- and 3-dimensional graphics and animation,
- possibility of editing text and automatic LaTeX conversion,
- on-line help.

Computer algebra systems are also called ***mathematical expert systems***. Today we can see an astonishing development of general purpose computer algebra systems, mainly because of their knowledge and wide application area. But it would be a mistake to underestimate special systems, which play a very important role in many research fields, besides, in many cases are easier to use and more effective due to

[2] Declarative programming languages specify the desired result unlike imperative languages, which describe how to get the result.

their system of notation and the low-level programming language implementation of their algorithms. It is essential that we choose the most appropriate computer algebra system to solve a specific problem.

Problems

6-1 The length of coefficients in the series of remainders in a Euclidean division

Generate two pseudorandom polynomials of degree $n = 10$ in $\mathbf{Z}[x]$ with coefficients of $l = 10$ decimal digits. Perform a single Euclidean division in (in $\mathbb{Q}[x]$) and compute the ratio of the maximal coefficients of the remainder and the original polynomial (determined by the function λ). Repeat the computation $t = 20$ times and compute the average. What is the result? Repeat the experiment with $l = 100, 500, 1000$.

6-2 Simulation of the* MODULAR-GCD-SMALLPRIMES *algorithm

Using simulation, give an estimation for the optimal value of the variable n in the MODULAR-GCD-SMALLPRIMES algorithm. Use random polynomials of different degrees and coefficient magnitudes.

6-3 Modified pseudo-euclidean division

Let $f, g \in \mathbf{Z}[x]$, $\deg f = m \geq n = \deg g$. Modify the pseudo-euclidean division in such a way that in the equation

$$g_n^s f = gq + r$$

instead of the exponent $s = m - n + 1$ put the smallest value $s \in \mathbb{N}$ such that $q, r \in \mathbf{Z}[x]$. Replace the procedures pquo() and prem() in the PRIMITIVE-EUCLIDEAN algorithm by the obtained procedures spquo() and sprem(). Compare the amount of memory space required by the algorithms.

6-4 Construction of reduced Gröbner basis

Design an algorithm that computes a reduced Gröbner basis from a given Gröbner-basis G.

6-5 Implementation of Hermite-reduction

Implement Hermite-reduction in a chosen computer algebra language.

6-6 Integration of rational functions

Write a program for the integration of rational functions.

Chapter notes

The algorithms CLASSICAL-EUCLIDEAN and EXTENDED-EUCLIDEAN for non-negative integers are described in [51]. A natural continuation of the theory of resultants leads to subresultants, which can help in reducing the growth of the coefficients in the EXTENDED-EUCLIDEAN algorithm (see e.g. [76, 82]).

Gröbner bases were introduced by B. Buchberger in 1965 [33]. Several authors examined polynomial ideals before this. The most well-known is perhaps Hironaka,

who used bases of ideals of power series to resolve singularities over $\mathbb{C}$. He was rewarded a Fields-medal for his work. His method was not constructive, however. Gröbner bases have been generalised for many algebraic structures in the last two decades.

The bases of differential algebra have been layed by J. F. Ritt in 1948 [199]. The square-free factorisation algorithm used in symbolic integration can be found for example in the books [76, 82]. The importance of choosing the smallest possible extension degree in Hermite-reduction is illustrated by Example 11.11 in [82], where the splitting field has very large degree but the integral can be expressed in an extension of degree 2. The proof of the Rothstein-Trager integration algorithm can be found in [76] (Theorem 22.8.). We note that the algorithm was found independently by Rothstein and Trager. The proof of the correctness of the Lazard-Rioboo-Trager formula, the analysis of the running time of the INTEGRATE-LOGARITHMIC-PART algorithm, an overview of the procedures that deal with the difficulties of algebraic extension steps, the determination of the hyperexponential integral (if exists) of a hyperexponential element over $C(x)$, the proof of Liouville's principle and the proofs of the statements connected to the Risch algorithm can be found in the book [76].

There are many books and publications available on computer algebra and related topics. The interested reader will find mathematical description in the following general works: Caviness [39], Davenport et al. [57], von zur Gathen et al. [76], Geddes et al. [82], Knuth [134, 135, 136], Mignotte [167], Mishra [170], Pavelle et al. [187], Winkler [258].

The computer-oriented reader will find further information on computer algebra in Christensen [41], Gonnet and Gruntz [90], Harper et al. [102] and on the world wide web.

A wide range of books and articles deal with applications, e.g. Akritas [9], Cohen et al. (ed.) [46, 47], Grossman (ed.) [95], Hearn (ed.) [104], Kovács [138] and Odlyzko [181].

For the role of computer algebra systems in education see for example the works of Karian [126] and Uhl [245].

Conference proceedings: AAECC, DISCO, EUROCAL, EUROSAM, ISSAC and SYMSAC.

Computer algebra journals: *Journal of Symbolic Computation* – Academic Press, *Applicable Algebra in Engineering, Communication and Computing* – Springer-Verlag, SIGSAM *Bulletin* – ACM Press.

The Department of Computer Algebra of the Eötvös Loránd University, Budapest takes works [76, 82, 167, 258] as a base in education.

7. Cryptology

This chapter introduces a number of cryptographic protocols and their underlying problems and algorithms. A typical cryptographic scenario is shown in Figure 7.1 (the design of Alice and Bob is due to Crépeau). Alice and Bob wish to exchange messages over an insecure channel, such as a public telephone line or via email over a computer network. Erich is eavesdropping on the channel. Knowing that their data transfer is being eavesdropped, Alice and Bob encrypt their messages using a cryptosystem.

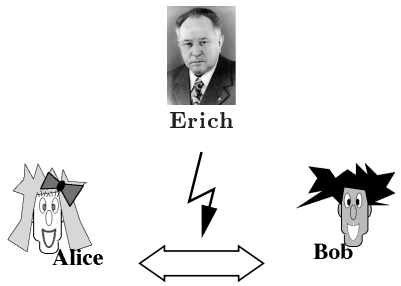


Figure 7.1. A typical scenario in cryptography.

In Section 7.1, various symmetric cryptosystems are presented. A cryptosystem is said to be symmetric if one and the same key is used for encryption and decryption. For symmetric systems, the question of key distribution is central: How can Alice and Bob agree on a joint secret key if they can communicate only via an insecure channel? For example, if Alice chooses some key and encrypts it like a message using a symmetric cryptosystem to send it to Bob, then which key should she use to encrypt this key?

This paradoxical situation is known as ***the secret-key agreement problem***, and it was considered unsolvable for a long time. Its surprisingly simple, ingenious solution by Whitfield Diffie and Martin Hellman in 1976 is a milestone in the history of cryptography. They proposed a protocol that Alice and Bob can use to exchange a few messages after which they both can easily determine their joint secret key. Eavesdropper Erich, however, does not have clue about their key, even if he was able to intercept every single bit of their transferred messages. Section 7.2 presents the Diffie-Hellman secret-key agreement protocol.

It may be considered an irony of history that this protocol, which finally solved the long-standing secret-key agreement problem that is so important in symmetric cryptography, opened the door to ***public-key cryptography*** in which there is no need to distribute joint secret keys via insecure channels. In 1978, shortly after Diffie and Hellman had published their pathbreaking work in 1976, Rivest, Shamir, and Adleman developed their famous RSA system, the first ***public-key cryptosystem*** in the open literature. Section 7.3 describes the RSA cryptosystem and the related digital signature scheme. Using the latter protocol, Alice can sign her message to Bob such that he can verify that she indeed is the sender of the message. Digital signatures prevent Erich from forging Alice's messages.

The security of the Diffie-Hellman protocol rests on the assumption that computing discrete logarithms is computationally intractable. That is why modular exponentiation (the inverse function of which is the discrete logarithm) is considered to be a candidate of a one-way function. The security of RSA similarly rests on the assumption that a certain problem is computationally intractable, namely on the assumption that factoring large integers is computationally hard. However, the authorised receiver Bob is able to efficiently decrypt the ciphertext by employing the factorisation of some integer he has chosen in private, which is his private "trapdoor" information.

Section 7.4 introduces a secret-key agreement protocol developed by Rivest and Sherman, which is based on so-called strongly noninvertible associative one-way functions. This protocol can be modified to yield a digital signature scheme as well.

Section 7.5 introduces the fascinating area of interactive proof systems and zero-knowledge protocols that has practical applications in cryptography, especially for authentication issues. In particular, a zero-knowledge protocol for the graph isomorphism problem is presented. On the other hand, this area is also central to complexity theory and will be revisited in Chapter 8, again in connection to the graph isomorphism problem.

7.1. Foundations

Cryptography is the art and science of designing secure cryptosystems, which are used to encrypt texts and messages so that they be kept secret and unauthorised decryption is prevented, whereas the authorised receiver is able to efficiently decrypt the ciphertext received. This section presents two classical symmetric cryptosystems. In subsequent sections, some important asymmetric cryptosystems and cryptographic protocols are introduced. A "protocol" is dialog between two (or more) parties, where a "party" may be either a human being or a computing machine. Cryptographic protocols can have various cryptographic purposes. They consist of algorithms that are jointly executed by more than one party.

Cryptanalysis is the art and science of (unauthorised) decryption of ciphertexts and of breaking existing cryptosystems. ***Cryptology*** captures both these fields, cryptography and cryptanalysis. In this chapter, we focus on ***cryptographic algorithms.*** Algorithms of cryptanalysis, which are used to break cryptographic protocols and systems, will be mentioned as well but will not be investigated in detail.

7.1.1. Cryptography

Figure 7.1 shows a typical scenario in cryptography: Alice and Bob communicate over an insecure channel that is eavesdropped by Erich, and thus they encrypt their messages using a cryptosystem.

Definition 7.1 (Cryptosystem). *A **cryptosystem** is a quintuple $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ with the following properties:*

1. *$\mathcal{P}$, $\mathcal{C}$, and $\mathcal{K}$ are finite sets, where $\mathcal{P}$ is the **plaintext space**, $\mathcal{C}$ is the **ciphertext space**, and $\mathcal{K}$ is the **key space**. The elements of $\mathcal{P}$ are called the **plaintexts**, and the elements of $\mathcal{C}$ are called the **ciphertexts**. A **message** is a string of plaintext symbols.*
2. *$\mathcal{E} = \{E_k \mid k \in \mathcal{K}\}$ is a family of functions $E_k : \mathcal{P} \to \mathcal{C}$, which are used for encryption. $\mathcal{D} = \{D_k \mid k \in \mathcal{K}\}$ is a family of functions $D_k : \mathcal{C} \to \mathcal{P}$, which are used for decryption.*
3. *For each key $e \in \mathcal{K}$, there exists some key $d \in \mathcal{K}$ such that for each plaintext $p \in \mathcal{P}$,*
$$D_d(E_e(p)) = p \ . \tag{7.1}$$

*A cryptosystem is said to be **symmetric** (or private-key) if either $d = e$ or if d at least can "easily" be determined from e. A cryptosystem is said to be **asymmetric** (or public-key) if $d \neq e$ and it is "computationally infeasible" to determine the private key d from the corresponding public key e.*

At times, we may use distinct key spaces for encryption and decryption, with the above definition modified accordingly.

We now introduce some easy examples of classical symmetric cryptosystems. Consider the alphabet $\Sigma = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$, which will be used both for the plaintext space and for the ciphertext space. We identify Σ with $\mathbf{Z}_{26} = \{0, 1, \ldots, 25\}$ so as to be able to perform calculations with letters as if they were numbers. The number 0 corresponds to the letter A, the 1 corresponds to B, and so on. This coding of plaintext or ciphertext symbols by nonnegative integers is not part of the actual encryption or decryption.

Messages are elements of Σ^*, where Σ^* denotes the set of strings over Σ. If some message $m \in \Sigma^*$ is subdivided into blocks of length n and is encrypted blockwise, as it is common in many cryptosystems, then each block of m is viewed as an element of $\mathbf{Z}_{26}^n$.

Example 7.1 [Shift Cipher] The first example is a monoalphabetic symmetric cryptosystem. Let $\mathcal{K} = \mathcal{P} = \mathcal{C} = \mathbf{Z}_{26}$. The ***shift cipher*** encrypts messages by shifting every plaintext symbol by the same number k of letters in the alphabet modulo 26. Shifting each letter in the ciphertext back using the same key k, the original plaintext is recovered. For each key $k \in \mathbf{Z}_{26}$, the encryption function E_k and the decryption function D_k are defined by:

$$\begin{aligned} E_k(m) &= (m + k) \bmod 26 \\ D_k(c) &= (c - k) \bmod 26 \ , \end{aligned}$$

m	S H I F T E A C H L E T T E R T O T H E L E F T
c	R G H E S D Z B G K D S S D Q S N S G D K D E S

Figure 7.2. Example of an encryption by the shift cipher.

where addition and subtraction by k modulo 26 are carried out characterwise.

Figure 7.2 shows an encryption of the message m by the shift cipher with key $k = 25$. The resulting ciphertext is c. Note that the particular shift cipher with key $k = 3$ is also known as the ***Caesar cipher***, since the Roman Emperor allegedly used this cipher during his wars to keep messages secret.[1] This cipher is a very simple substitution cipher in which each letter is substituted by a certain letter of the alphabet.

Since the key space is very small, the shift cipher can easily be broken. It is already vulnerable by attacks in which the attacker knows the ciphertext only, simply by checking which of the 26 possible keys reveals a meaningful plaintext, provided that the ciphertext is long enough to allow unique decryption.

The shift cipher is a monoalphabetic cryptosystem, since every plaintext letter is replaced by one and the same letter in the ciphertext. In contrast, a polyalphabetic cryptosystem can encrypt the same plaintext symbols by different ciphertext symbols, depending on their position in the text. Such a polyalphabetic cryptosystem that is based on the shift cipher, yet much harder to break, was proposed by the French diplomat Blaise de Vigenère (1523 until 1596). His system builds on previous work by the Italian mathematician Leon Battista Alberti (born in 1404), the German abbot Johannes Trithemius (born in 1492), and the Italian scientist Giovanni Porta (born in 1675). It works like the shift cipher, except that the letter that encrypts some plaintext letter varies with its position in the text.

Example 7.2 [Vigenère Cipher] This symmetric polyalphabetic cryptosystem uses a so-called ***Vigenère square***, a matrix consisting of 26 rows and 26 columns, see Figure 7.3. Every row has the 26 letters of the alphabet, shifted from row to row by one position. That is, the single rows can be seen as a shift cipher obtained by the keys $0, 1, \ldots, 25$. Which row of the Vigenère square is used for encryption of some plaintext symbol depends on its position in the text.

Messages are subdivided into blocks of a fixed length n and are encrypted blockwise, i.e., $\mathcal{K} = \mathcal{P} = \mathcal{C} = \mathbf{Z}_{26}^n$. The block length n is also called the *period* of the system. In what follows, the ith symbol in a string w is denoted by w_i.

For each key $k \in \mathbf{Z}_{26}^n$, the encryption function E_k and the decryption function D_k, both mapping from $\mathbf{Z}_{26}^n$ to $\mathbf{Z}_{26}^n$, are defined by:

$$\begin{aligned} E_k(b) &= (b + k) \bmod 26 \\ D_k(c) &= (c - k) \bmod 26, \end{aligned}$$

where addition and subtraction by k modulo 26 are again carried out characterwise. That is, the key $k \in \mathbf{Z}_{26}^n$ is written letter by letter above the symbols of the block $b \in \mathbf{Z}_{26}^n$ to be

[1] Historic remark: Gaius Julius Caesar reports in his book *De Bello Gallico* that he sent an encrypted message to Q. Tullius Cicero (the brother of the famous speaker) during the Gallic Wars (58 until 50 B.C.). The system used was monoalphabetic and replaced Latin letters by Greek letters; however, it is not explicitly mentioned there if the cipher used indeed was the shift cipher with key $k = 3$. This information was given later by Suetonius.

0	A	B	C	D	E	F	G	**H**	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
2	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
3	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
4	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
5	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
6	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
7	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
8	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
9	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
10	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
11	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
12	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
13	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
14	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
15	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
16	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
17	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
18	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
19	**T**	U	V	W	X	Y	Z	**A**	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
20	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
21	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
22	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
23	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
24	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
25	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Figure 7.3. Vigenère square: Plaintext "H" is encrypted as "A" by key "T".

k	T O N Y T O N Y T O N Y T O N Y T O N Y T O N Y
m	H U N G A R I A N I S A L L G R E E K T O G E R M A N S
c	A I A E T F V Y G W F Y E Z T P X S X R H U R P F O A Q

Figure 7.4. Example of an encryption by the Vigenère cipher.

encrypted. If the last plaintext block has less than n symbols, one uses less key symbols accordingly. In order to encrypt the ith plaintext symbol b_i, which has the key symbol k_i sitting on top, use the ith row of the Vigenère square as in the shift cipher.

For example, choose the block length $n = 4$ and the key $k = \text{TONY}$. Figure 7.4 shows the encryption of the message m, which consists of seven blocks, into the ciphertext c by the Vigenère cipher using key k.

To the first plaintext letter, "H", the key symbol "T" is assigned. The intersection of the "H" column with the "T" row of the Vigenère square yields "A" as the first letter of the ciphertext, see Figure 7.3.

There are many other classical cryptosystems, which will not be described in detail here. There are various ways to classify cryptosystems according to their properties or to the specific way they are designed. In Definition 7.1, the distinction between *symmetric* and *asymmetric* cryptosystems was explained. The two examples above (the shift cipher and the Vigenère cipher) demonstrated the distinction between *monoalphabetic* and *polyalphabetic* systems. Both are ***substitution ciphers***, which may be contrasted with ***permutation ciphers*** (a.k.a. ***transposition ciphers***) in which the plaintext letters are not substituted by certain ciphertext letters but go to another position in the text remaining otherwise unchanged.

Moreover, ***block ciphers*** such as the Vigenère system can be contrasted with ***stream ciphers***, which produce a continuous stream of key symbols depending on the plaintext context to be encrypted. One can also distinguish between different types of block ciphers. An important type are the ***affine linear block ciphers***, which are defined by affine linear encryption functions $E_{(A,\vec{b})}$ and decryption functions $D_{(A^{-1},\vec{b})}$, both mapping from $\mathbf{Z}_m^n$ to $\mathbf{Z}_m^n$. That is, they are of the following form:

$$\begin{aligned} E_{(A,\vec{b})}(\vec{x}) &= A\vec{x} + \vec{b} \bmod m \,, \\ D_{(A^{-1},\vec{b})}(\vec{y}) &= A^{-1}(\vec{y} - \vec{b}) \bmod m \,. \end{aligned} \tag{7.2}$$

Here, $(A, \vec{b})$ and $(A^{-1}, \vec{b})$ are the keys used for encryption and decryption, respectively; A is a $(n \times n)$ matrix with entries from $\mathbf{Z}_m$; A^{-1} is the inverse matrix for A; $\vec{x}$, $\vec{y}$, and $\vec{b}$ are vectors in $\mathbf{Z}_m^n$, and all arithmetics is carried out modulo m. Some mathematical explanations are in order (see also Definition 7.2 in Subsection 7.1.3): An $(n \times n)$ matrix A over the ring $\mathbf{Z}_m$ has a multiplicative inverse if and only if $\gcd(\det A, m) = 1$. The *inverse matrix for* A is defined by $A^{-1} = (\det A)^{-1} A_{\mathtt{adj}}$, where $\det A$ is the determinant of A, and $A_{\mathtt{adj}} = ((-1)^{i+j} \det A_{j,i})$ is the *adjunct matrix for* A. The *determinant* $\det A$ of A is recursively defined: For $n = 1$ and $A = (a)$, $\det A = a$; for $n > 1$ and each $i \in \{1, 2, \ldots, n\}$, $\det A = \sum_{j=1}^{n} (-1)^{i+j} a_{i,j} \det A_{i,j}$, where $a_{i,j}$ denotes the (i, j) entry of A and the $(n-1) \times (n-1)$ matrix $A_{i,j}$ results from A by cancelling the ith row and the jth column. The determinant of a matrix and thus its inverse (if it exists) can be computed efficiently, see Problem *7-3*.

For example, the Vigenère cipher is an affine linear cipher whose key contains the unity matrix as its first component. If $\vec{b}$ in (7.2) is the zero vector, then it is a *linear block cipher*. A classical example is the ***Hill cipher***, invented by Lester Hill in 1929. Here, the key space is the set of all $(n \times n)$ matrices A with entries in $\mathbf{Z}_m$ such that $\gcd(\det A, m) = 1$. This condition guarantees the invertibility of those matrices that are allowed as keys, since the inverse matrix A^{-1} is used for decryption of the messages encrypted by key A. For each key A, the Hill cipher is defined by the encryption function $E_A(\vec{x}) = A\vec{x} \bmod m$ and the decryption function $D_{A^{-1}}(\vec{y}) = A^{-1}\vec{y} \bmod m$. Thus, it is the most general linear cipher. The permutation cipher also is linear, and hence is a special case of the Hill cipher.

7.1.2. Cryptanalysis

Cryptanalysis aims at breaking existing cryptosystems and, in particular, at determining the decryption keys. In order to characterise the security or vulnerability of the cryptosystem considered, one distinguishes different types of attacks according to the information available for the attacker. For the shift cipher, ***ciphertext-only*** attacks were already mentioned. They are the weakest type of attack, and a cryptosystem that does not resist such attacks is not of much value.

Affine linear block ciphers such as the Vigenère and the Hill cipher are vulnerable to attacks in which the attacker knows the plaintext corresponding to some ciphertext obtained and is able to conclude the keys used. These attacks are called ***known-plaintext attacks***. Affine linear block ciphers are even more vulnerable to ***chosen-plaintext attacks***, in which the attacker can choose some plaintext and is then able to see which ciphertext corresponds to the plaintext chosen. Another type of attack is in particular relevant for asymmetric cryptosystems: In an ***encryption-key attack***, the attacker merely knows the public key but does not know any ciphertext yet, and seeks to determine the private key from this information. The difference is that the attacker now has plenty of time to perform computations, whereas in the other types of attacks the ciphertext was already sent and much less time is available to decrypt it. That is why keys of much larger size are required in public-key cryptography to guarantee the security of the system used. Hence, asymmetric cryptosystems are much less efficient than symmetric cryptosystems in many practical applications.

For the attacks mentioned above, the method of frequency counts is often useful. This method exploits the redundancy of the natural language used. For example, in many natural languages, the letter "E" occurs statistically significant most frequently. On average, the "E" occurs in long, "typical" texts with a percentage of 12.31% in English, of 15.87% in French, and even of 18.46% in German. In other languages, different letters may occur most frequently. For example, the "A" is the most frequent letter in long, "typical" Finnish texts, with a percentage of 12.06%.

That the method of frequency counts is useful for attacks on monoalphabetic cryptosystems is obvious. For example, if in a ciphertext encrypting a long German text by the shift cipher, the letter occurring most frequently is "Y", which is rather rare in German (as well as in many other languages), then it is most likely that "Y" encrypts "E". Thus, the key used for encryption is "U" ($k = 20$). In addition to counting the frequency of single letters, one can also count the frequency with which certain pairs of letters (so-called digrams) and triples of letters (so-called trigrams) occur, and so on. This kind of attack also works for polyalphabetic systems, provided the period (i.e., the block length) is known.

Polyalphabetic cryptosystems with an unknown period, however, provide more security. For example, the Vigenère cipher resisted each attempt of breaking it for a long time. No earlier than in 1863, about 300 years after its discovery, the German cryptanalyst Friedrich Wilhelm Kasiski found a method of breaking the Vigenère cipher. He showed how to determine the period, which initially is unknown, from repetitions of the same substring in the ciphertext. Subsequently, the ciphertext can be decrypted by means of frequency counts. Singh writes that the British eccentric

Charles Babbage, who was considered a genius of his time by many, presumably had discovered Kasiski's method even earlier, around 1854, although he didn't publish his work.

The pathbreaking work of Claude Shannon (1916 until 2001), the father of modern coding and information theory, is now considered a milestone in the history of cryptography. Shannon proved that there exist cryptosystems that guarantee perfect secrecy in a mathematically rigorous sense. More precisely, a cryptosystem $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ ***guarantees perfect secrecy*** if and only if $|\mathcal{P}| = |\mathcal{C}| = |\mathcal{K}|$, the keys in $\mathcal{K}$ are uniformly distributed, and for each plaintext $p \in \mathcal{P}$ and for each ciphertext $c \in \mathcal{C}$ there exists *exactly one* key $k \in \mathcal{K}$ with $E_k(p) = c$. That means that such a cryptosystem often is not useful for practical applications, since in order to guarantee perfect secrecy, every key must be at least as long as the message to be encrypted and can be used only once.

7.1.3. Algebra, number theory, and graph theory

In order to understand some of the algorithms and problems to be presented later, some fundamental notions, definitions, and results from algebra and, in particular, from number theory, group theory, and graph theory are required. This concerns both the cryptosystems and zero-knowledge protocols in Chapter 7 and some of the problems to be considered in upcoming Chapter 8. The present subsection may as well be skipped for now and revisited later when the required notions and results come up. In this section, most of the proofs are omitted.

Definition 7.2 (Group, ring, and field).

- *A* ***group*** $\mathfrak{G} = (S, \circ)$ *is defined by some nonempty set* S *and a two-ary operation* $\circ$ *on* S *that satisfy the following axioms:*
  - Closure: $(\forall x \in S)\,(\forall y \in S)\,[x \circ y \in S]$.
  - Associativity: $(\forall x \in S)\,(\forall y \in S)\,(\forall z \in S)\,[(x \circ y) \circ z = x \circ (y \circ z)]$.
  - Neutral element: $(\exists e \in S)\,(\forall x \in S)\,[e \circ x = x \circ e = x]$.
  - Inverse element: $(\forall x \in S)\,(\exists x^{-1} \in S)\,[x \circ x^{-1} = x^{-1} \circ x = e]$.

  The element e *is called the* ***neutral element of the group*** $\mathfrak{G}$*. The element* x^{-1} *is called the* ***inverse element for*** x*.* $\mathfrak{G}$ *is said to be a* ***monoid*** *if* $\mathfrak{G}$ *satisfies associativity and closure under* $\circ$*, even if* $\mathfrak{G}$ *has no neutral element or if not every element in* $\mathfrak{G}$ *has an inverse. A group* $\mathfrak{G} = (S, \circ)$ *(respectively, a monoid* $\mathfrak{G} = (S, \circ)$*) is said to be* ***commutative*** *(or abelian) if and only if* $x \circ y = y \circ x$ *for all* $x, y \in S$*. The number of elements of a finite group* $\mathfrak{G}$ *is said to be* ***the order*** $\mathfrak{G}$ *and is denoted by* $|\mathfrak{G}|$*.*
- $\mathfrak{H} = (T, \circ)$ *is said to be a* ***subgroup of a group*** $\mathfrak{G} = (S, \circ)$ *(denoted by* $\mathfrak{H} \leq \mathfrak{G}$*) if and only if* $T \subseteq S$ *and* $\mathfrak{H}$ *satisfies the group axioms.*
- *A* ***ring*** *is a triple* $\mathfrak{R} = (S, +, \cdot)$ *such that* $(S, +)$ *is an abelian group and* $(S, \cdot)$ *is a monoid and the distributive laws are satisfied:*

$$(\forall x \in S)\,(\forall y \in S)\,(\forall z \in S) \;:$$

$$(x \cdot (y+z) = (x \cdot y) + (x \cdot z)) \wedge ((x+y) \cdot z = (x \cdot z) + (y \cdot z)) \ .$$

A ring $\mathfrak{R} = (S, +, \cdot)$ is said to be ***commutative*** *if and only if the monoid $(S, \cdot)$ is commutative. The neutral element group $(S, +)$ is called the* ***zero element*** *(the zero, for short) of the ring $\mathfrak{R}$. A neutral element of the monoid $(S, \cdot)$ is called the* ***one element*** *(the one, for short) of the ring $\mathfrak{R}$.*

- *Let $\mathfrak{R} = (S, +, \cdot)$ be a ring with one. An element x of $\mathfrak{R}$ is said to be* ***invertible*** *(or a unity of $\mathfrak{R}$) if and only if it is invertible in the monoid $(S, \cdot)$.*
- *A* ***field*** *is a commutative ring with one in which each nonzero element is invertible.*

Example 7.3 [Group, ring, and field]

- Let $k \in \mathbb{N}$. The set $\mathbf{Z}_k = \{0, 1, \ldots, k-1\}$ is a finite group with respect to addition modulo k, with neutral element 0. With respect to addition and multiplication modulo k, $\mathbf{Z}_k$ is a commutative ring with one, see Problem 7-1. If p is a *prime number*, then $\mathbf{Z}_p$ is a field with respect to addition and multiplication modulo p.
- Let $\gcd n, m$ denote the greatest common divisor of two integers m and n. For $k \in \mathbb{N}$, define the set $\mathbf{Z}_k^* = \{i \mid 1 \leq i \leq k-1 \text{ and } \gcd i, k = 1\}$. With respect to multiplication modulo k, $\mathbf{Z}_k^*$ is a finite group with neutral element 1.

If the operation $\circ$ of a group $\mathfrak{G} = (S, \circ)$ is clear from the context, we omit stating it explicitly. The group $\mathbf{Z}_k^*$ from Example 7.3 will play a particular role in Section 7.3, where the RSA cryptosystem is introduced. The *Euler function* φ gives the order of this group, i.e., $\varphi(k) = |\mathbf{Z}_k^*|$. The following properties of φ follow from the definition:

- $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$ for all $m, n \in \mathbb{N}$ with $\gcd m, n = 1$, and
- $\varphi(p) = p - 1$ for all prime numbers p.

The proof of these properties is left to the reader as Exercise 7.1-3. In particular, we will apply the following fact in Subsection 7.3.1, which is a consequence of the properties above.

Proposition 7.3 *If $n = p \cdot q$ for prime numbers p and q, then $\varphi(n) = (p-1)(q-1)$.*

Euler's Theorem below is a special case (namely, for the group $\mathbf{Z}_n^*$) of Lagrange's Theorem, which says that for each group element a of a finite multiplicative group $\mathfrak{G}$ of order $|\mathfrak{G}|$ and with neutral element e, $a^{|\mathfrak{G}|} = e$. The special case of Euler's theorem, where n is a prime number not dividing a, is known as Fermat's Little Theorem.

Theorem 7.4 (Euler). *For each $a \in \mathbf{Z}_n^*$, $a^{\varphi(n)} \equiv 1 \bmod n$.*

Corollary 7.5 (Fermat's Little Theorem). *If p is a prime number and $a \in \mathbf{Z}_p^*$, then $a^{p-1} \equiv 1 \bmod p$.*

In Section 8.4, algorithms for the graph isomorphism problem will be presented. This problem, which also is related to the zero-knowledge protocols to be introduced

in Subsection 7.5.2, can be seen as a special case of certain group-theoretic problems. In particular, *permutation groups* are of interest here. Some examples for illustration will be presented later.

Definition 7.6 (Permutation group).

- *A* ***permutation*** *is a bijective mapping of a set onto itself. For each integer* $n \geq 1$*, let* $[n] = \{1, 2, \ldots, n\}$*. The set of all permutations of* $[n]$ *is denoted by* $\mathfrak{S}_n$*. For algorithmic purposes, permutations* $\pi \in \mathfrak{S}_n$ *are given as pairs* $(i, \pi(i))$ *from* $[n] \times [n]$*.*
- *If one defines the composition of permutations as an operation on* $\mathfrak{S}_n$*, then* $\mathfrak{S}_n$ *becomes a group. For two permutations* π *and* τ *in* $\mathfrak{S}_n$*, their* composition $\pi\tau$ *is defined to be that permutation in* $\mathfrak{S}_n$ *that results from first applying* π *and then* τ *to the elements of* $[n]$*, i.e.,* $(\pi\tau)(i) = \tau(\pi(i))$ *for each* $i \in [n]$*. The neutral element of the permutation group* $\mathfrak{S}_n$ *is the* ***identical permutation****, which is defined by* $\mathrm{id}(i) = i$ *for each* $i \in [n]$*. The subgroup of* $\mathfrak{S}_n$ *that contains* id *as its only element is denoted by* $\mathbf{id}$*.*
- *For any subset* $\mathfrak{T}$ *of* $\mathfrak{S}_n$*, the* ***permutation group*** $\langle\mathfrak{T}\rangle$ ***generated by*** $\mathfrak{T}$ *is defined as the smallest subgroup of* $\mathfrak{S}_n$ *containing* $\mathfrak{T}$*. Subgroups* $\mathfrak{G}$ *of* $\mathfrak{S}_n$ *are represented by their generating sets, sometimes dubbed the generators of* $\mathfrak{G}$*. In* $\mathfrak{G}$*, the* ***orbit of an element*** $i \in [n]$ *is defined as* $\mathfrak{G}(i) = \{\pi(i) \mid \pi \in \mathfrak{G}\}$*.*
- *For any subset* T *of* $[n]$*, let* $\mathfrak{S}_n^T$ *denote the subgroup of* $\mathfrak{S}_n$ *that maps every element of* T *onto itself. In particular, for* $i \leq n$ *and a subgroup* $\mathfrak{G}$ *of* $\mathfrak{S}_n$*, the* ***(pointwise) stabiliser of*** $[i]$ ***in*** $\mathfrak{G}$ *is defined by*

$$\mathfrak{G}^{(i)} = \{\pi \in \mathfrak{G} \mid \pi(j) = j \text{ for each } j \in [i]\} \ .$$

  Observe that $\mathfrak{G}^{(n)} = \mathbf{id}$ *and* $\mathfrak{G}^{(0)} = \mathfrak{G}$*.*
- *Let* $\mathfrak{G}$ *and* $\mathfrak{H}$ *be permutation groups with* $\mathfrak{H} \leq \mathfrak{G}$*. For* $\tau \in \mathfrak{G}$*,* $\mathfrak{H}\tau = \{\pi\tau \mid \pi \in \mathfrak{H}\}$ *is said to be a* ***right coset of*** $\mathfrak{H}$ ***in*** $\mathfrak{G}$*. Any two right cosets of* $\mathfrak{H}$ *in* $\mathfrak{G}$ *are either identical or disjoint. Thus, the permutation group* $\mathfrak{G}$ *is partitioned by the right cosets of* $\mathfrak{H}$ *in* $\mathfrak{G}$*:*

$$\mathfrak{G} = \mathfrak{H}\tau_1 \cup \mathfrak{H}\tau_2 \cup \cdots \cup \mathfrak{H}\tau_k \ . \tag{7.3}$$

  Every right coset of $\mathfrak{H}$ *in* $\mathfrak{G}$ *has the cardinality* $|\mathfrak{H}|$*. The set* $\{\tau_1, \tau_2, \ldots, \tau_k\}$ *in (7.3) is called the* ***complete right transversal of*** $\mathfrak{H}$ ***in*** $\mathfrak{G}$*.*

The notion of pointwise stabilisers is especially important for the design of algorithms solving problems on permutation groups. The crucial structure exploited there is the so-called ***tower of stabilisers*** of a permutation group $\mathfrak{G}$:

$$\mathbf{id} = \mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)} = \mathfrak{G} \ .$$

For each i with $1 \leq i \leq n$, let $\mathfrak{T}_i$ be the complete right transversal of $\mathfrak{G}^{(i)}$ in $\mathfrak{G}^{(i-1)}$. Then, $\mathfrak{T} = \bigcup_{i=1}^{n-1} \mathfrak{T}_i$ is said to be a ***strong generator of*** $\mathfrak{G}$, and we have $\mathfrak{G} = \langle\mathfrak{T}\rangle$. Every $\pi \in \mathfrak{G}$ then has a unique factorisation $\pi = \tau_1\tau_2\cdots\tau_n$ with $\tau_i \in \mathfrak{T}_i$. The following basic algorithmic results about permutation groups will be useful later in Section 8.4.

Theorem 7.1 *Let a permutation group $\mathfrak{G} \leq \mathfrak{S}_n$ be given by a generator. Then, we have:*

1. *For each $i \in [n]$, the orbit $\mathfrak{G}(i)$ of i in $\mathfrak{G}$ can be computed in polynomial time.*
2. *The tower of stabilisers $\mathbf{id} = \mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)} = \mathfrak{G}$ can be computed in time polynomially in n, i.e., for each i with $1 \leq i \leq n$, the complete right transversals $\mathfrak{T}_i$ of $\mathfrak{G}^{(i)}$ in $\mathfrak{G}^{(i-1)}$ and thus a strong generator of $\mathfrak{G}$ can be determined efficiently.*

The notions introduced in Definition 7.6 for permutation groups are now explained for concrete examples from graph theory. In particular, we consider the automorphism group and the set of isomorphisms between graphs. We start by introducing some useful graph-theoretical concepts.

Definition 7.7 (Graph isomorphism and graph automorphism). *A graph G consists of a finite set of vertices, $V(G)$, and a finite set of edges, $E(G)$, that connect certain vertices with each other. We assume that no multiple edges and no loops occur. In this chapter, we consider only undirected graphs, i.e., the edges are not oriented and can be seen as unordered vertex pairs. The* disjoint union *$G \cup H$* of two graphs *G* and *H is defined as the graph with vertex set $V(G) \cup V(H)$ and edge set $E(G) \cup E(H)$, where we assume that that the vertex sets $V(G)$ and $V(H)$ are made disjoint (by renaming if necessary).*

*Let G and H be two graphs with the same number of vertices. An **isomorphism between G and H** is an edge-preserving bijection of the vertex set of G onto that of H. Under the convention that $V(G) = \{1, 2, \ldots, n\} = V(H)$, G and H are isomorphic ($G \cong H$, for short) if and only if there exists a permutation $\pi \in \mathfrak{S}_n$ such that for all vertices $i, j \in V(G)$,*

$$\{i, j\} \in E(G) \iff \{\pi(i), \pi(j)\} \in E(H) \,. \tag{7.4}$$

*An **automorphism of** G is an edge-preserving bijection of the vertex set of G onto itself. Every graph has the trivial automorphism* id. *By* $\mathrm{Iso}(G, H)$ *we denote the set of all isomorphisms between G and H, and by* $\mathrm{Aut}(G)$ *we denote the set of all automorphisms of G. Define the problems **graph automorphism** (*GI*, for short) and **graph automorphism** (*GA*, for short) by:*

$$\begin{aligned} \texttt{GI} &= \{(G, H) \,|\, G \text{ and } H \text{ are isomorphic graphs}\} \,; \\ \texttt{GA} &= \{G \,|\, G \text{ has a nontrivial automorphism}\} \,. \end{aligned}$$

For algorithmic purposes, graphs are represented either by their vertex and edge lists or by an adjacency matrix, which has the entry 1 at position (i, j) if $\{i, j\}$ is an edge, and the entry 0 otherwise. This graph representation is suitably encoded over the alphabet $\Sigma = \{0, 1\}$. In order to represent pairs of graphs, we use a standard bijective pairing function $(\cdot, \cdot)$ from $\Sigma^* \times \Sigma^*$ onto Σ^* that is polynomial-time computable and has polynomial-time computable inverses.

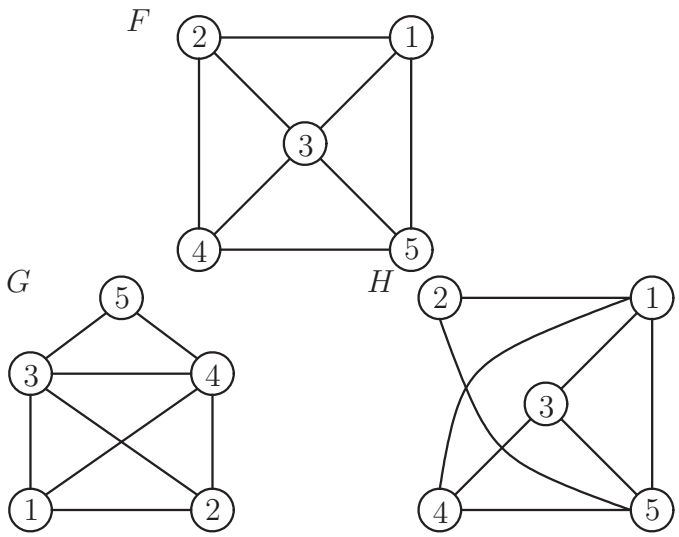


Figure 7.5. Three graphs: G is isomorphic to H, but not to F.

Example 7.4 [Graph isomorphism and graph automorphism] The graphs G and H shown in Figure 7.5 are isomorphic.

An isomorphism $\pi : V(G) \to V(H)$ preserving adjacency of the vertices according to (7.4) is given, e.g., by $\pi = \left(\begin{smallmatrix}1&2&3&4&5\\3&4&1&5&2\end{smallmatrix}\right)$, or in cycle notation by $\pi = (1\,3)(2\,4\,5)$. There are three further isomorphisms between G and H, i.e., $|\mathrm{Iso}(G,H)| = 4$, see Exercise 7.1-4. However, neither G nor H is isomorphic to F. This is immediately seen from the fact that the sequence of *vertex degrees* (the number of edges fanning out of each vertex) of G and H, respectively, differs from the sequence of vertex degrees of F: For G and H, this sequence is $(2,3,3,4,4)$, whereas it is $(3,3,3,3,4)$ for F. A nontrivial automorphism $\varphi : V(G) \to V(G)$ of G is given, e.g., by $\varphi = \left(\begin{smallmatrix}1&2&3&4&5\\2&1&4&3&5\end{smallmatrix}\right)$, or $\varphi = (1\,2)(3\,4)(5)$; another one by $\tau = \left(\begin{smallmatrix}1&2&3&4&5\\1&2&4&3&5\end{smallmatrix}\right)$, or $\tau = (1)(2)(3\,4)(5)$. There are two more automorphisms of G, i.e., $|\mathrm{Aut}(G)| = 4$, see Exercise 7.1-4.

The permutation groups $\mathrm{Aut}(F)$, $\mathrm{Aut}(G)$, and $\mathrm{Aut}(H)$ are subgroups of $\mathfrak{S}_5$. The tower $\mathrm{Aut}(G)^{(5)} \leq \mathrm{Aut}(G)^{(4)} \leq \cdots \leq \mathrm{Aut}(G)^{(1)} \leq \mathrm{Aut}(G)^{(0)}$ of stabilisers of $\mathrm{Aut}(G)$ consists of the subgroups $\mathrm{Aut}(G)^{(5)} = \mathrm{Aut}(G)^{(4)} = \mathrm{Aut}(G)^{(3)} = \mathbf{id}$, $\mathrm{Aut}(G)^{(2)} = \mathrm{Aut}(G)^{(1)} = \langle\{\mathrm{id}, \tau\}\rangle$, and $\mathrm{Aut}(G)^{(0)} = \mathrm{Aut}(G)$. In the automorphism group $\mathrm{Aut}(G)$ of G, the vertices 1 and 2 have the orbit $\{1,2\}$, the vertices 3 and 4 have the orbit $\{3,4\}$, and vertex 5 has the orbit $\{5\}$.

$\mathrm{Iso}(G,H)$ and $\mathrm{Aut}(G)$ have the same number of elements if and only if G and H are isomorphic. To wit, if G and H are isomorphic, then $|\mathrm{Iso}(G,H)| = |\mathrm{Aut}(G)|$ follows from $\mathrm{Aut}(G) = \mathrm{Iso}(G,G)$. Otherwise, if $G \ncong H$, then $\mathrm{Iso}(G,H)$ is empty but $\mathrm{Aut}(G)$ contains always the trivial automorphism id. This implies (7.5) in Lemma 7.8 below, which we will need later in Section 8.4. For proving (7.6), suppose that G and H are connected; otherwise, we simply consider instead of G and H the co-graphs $\overline{G}$ and $\overline{H}$, see Exercise 7.1-5. An automorphism of $G \cup H$ that switches the vertices of G and H, consists of an isomorphism in $\mathrm{Iso}(G,H)$ and an isomorphism in $\mathrm{Iso}(H,G)$. Thus, $|\mathrm{Aut}(G \cup H)| = |\mathrm{Aut}(G)| \cdot |\mathrm{Aut}(H)| + |\mathrm{Iso}(G,H)|^2$, which implies

(7.6) via (7.5).

Lemma 7.8 *For any two graphs G and H, we have*

$$|\mathrm{Iso}(G,H)| = \begin{cases} |\mathrm{Aut}(G)| = |\mathrm{Aut}(H)| & \text{if } G \cong H \\ 0 & \text{if } G \ncong H \end{cases} ; \tag{7.5}$$

$$|\mathrm{Aut}(G \cup H)| = \begin{cases} 2 \cdot |\mathrm{Aut}(G)| \cdot |\mathrm{Aut}(H)| & \text{if } G \cong H \\ |\mathrm{Aut}(G)| \cdot |\mathrm{Aut}(H)| & \text{if } G \ncong H \end{cases} . \tag{7.6}$$

If G and H are isomorphic graphs and if $\tau \in \mathrm{Iso}(G,H)$, then $\mathrm{Iso}(G,H) = \mathrm{Aut}(G)\tau$. Thus, $\mathrm{Iso}(G,H)$ is a right coset of $\mathrm{Aut}(G)$ in $\mathfrak{S}_n$. Since any two right cosets are either identical or disjoint, $\mathfrak{S}_n$ can be partitioned into right cosets of $\mathrm{Aut}(G)$ according to (7.3):

$$\mathfrak{S}_n = \mathrm{Aut}(G)\tau_1 \cup \mathrm{Aut}(G)\tau_2 \cup \cdots \cup \mathrm{Aut}(G)\tau_k , \tag{7.7}$$

where $|\mathrm{Aut}(G)\tau_i| = |\mathrm{Aut}(G)|$ for all i, $1 \le i \le k$. The set $\{\tau_1, \tau_2, \ldots, \tau_k\}$ of permutations in $\mathfrak{S}_n$ thus is a complete right transversal of $\mathrm{Aut}(G)$ in $\mathfrak{S}_n$. Denoting by $\pi(G)$ the graph $H \cong G$ that results from applying the permutation $\pi \in \mathfrak{S}_n$ to the vertices of G, we have $\{\tau_i(G) \mid 1 \le i \le k\} = \{H \mid H \cong G\}$. Since there are exactly $n! = n(n-1)\cdots 2 \cdot 1$ permutations in $\mathfrak{S}_n$, it follows from (7.7) that

$$|\{H \mid H \cong G\}| = k = \frac{|\mathfrak{S}_n|}{|\mathrm{Aut}(G)|} = \frac{n!}{|\mathrm{Aut}(G)|} .$$

This proves the following corollary.

Corollary 7.9 *If G is a graph with n vertices, then there are exactly $n!/|\mathrm{Aut}(G)|$ graphs isomorphic to G.*

For the graph G in Figure 7.5 from Example 7.4, there thus exist exactly $5!/4 = 30$ isomorphic graphs. The following lemma will be used later in Section 8.4.

Lemma 7.10 *Let G and H be two graphs with n vertices. Define the set*

$$A(G,H) = \{(F,\varphi) \mid F \cong G \text{ and } \varphi \in \mathrm{Aut}(F)\} \cup \{(F,\varphi) \mid F \cong H \text{ and } \varphi \in \mathrm{Aut}(F)\} .$$

Then, we have

$$|A(G,H)| = \begin{cases} n! & \text{if } G \cong H \\ 2n! & \text{if } G \ncong H \end{cases} .$$

Proof If F and G are isomorphic, then $|\mathrm{Aut}(F)| = |\mathrm{Aut}(G)|$. Hence, by Corollary 7.9, we have

$$|\{(F,\varphi) \mid F \cong G \text{ and } \varphi \in \mathrm{Aut}(F)\}| = \frac{n!}{|\mathrm{Aut}(F)|} \cdot |\mathrm{Aut}(F)| = n! .$$

Analogously, one can show that $|\{(F,\varphi) \mid F \cong H \text{ and } \varphi \in \mathrm{Aut}(F)\}| = n!$. If G and H are isomorphic, then

$$\{(F,\varphi) \mid F \cong G \text{ and } \varphi \in \mathrm{Aut}(F)\} = \{(F,\varphi) \mid F \cong H \text{ and } \varphi \in \mathrm{Aut}(F)\} .$$

c_1	W K L V V H Q W H Q F H L V H Q F U B S W H G E B F D H V D U V N H B
c_2	N U O S J Y A Z E E W R O S V H P X Y G N R J B P W N K S R L F Q E P

Figure 7.6. Two ciphertexts encrypting the same plaintext, see Exercise 7.1-1.

It follows that $|A(G,H)| = n!$. If G and H are nonisomorphic, then $\{(F,\varphi) \mid F \cong G$ and $\varphi \in \mathrm{Aut}(F)\}$ and $\{(F,\varphi) \mid F \cong H$ and $\varphi \in \mathrm{Aut}(F)\}$ are disjoint sets. Thus, $|A(G,H)| = 2n!$. ■

Exercises

7.1-1 Figure 7.6 shows two ciphertexts, c_1 and c_2. It is known that both encrypt the same plaintext m and that one was obtained using the shift cipher, the other one using the Vigenère cipher. Decrypt both ciphertexts. *Hint.* After decryption you will notice that the plaintext obtained is a true statement for one of the two ciphertexts, whereas it is a false statement for the other ciphertext. Is perhaps the method of frequency counts useful here?

7.1-2 Prove that $\mathbf{Z}$ is a ring with respect to ordinary addition and multiplication. Is it also a field? What can be said about the properties of the algebraic structures $(\mathbf{N},+)$, $(\mathbf{N},\cdot)$, and $(\mathbf{N},+,\cdot)$?

7.1-3 Prove the properties stated for Euler's φ function:

a. $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$ for all $m,n \in \mathbf{N}$ with $\gcd(m,n) = 1$.

b. $\varphi(p) = p - 1$ for all prime numbers p.

Using these properties, prove Proposition 7.3.

7.1-4 Consider the graphs F, G, and H from Figure 7.5 in Example 7.4.

a. Determine all isomorphisms between G and H.

b. Determine all automorphisms of F, G, and H.

c. For which isomorphisms between G and H is $\mathrm{Iso}(G,H)$ a right coset of $\mathrm{Aut}(G)$ in $\mathfrak{S}_5$, i.e., for which $\tau \in \mathrm{Iso}(G,H)$ is $\mathrm{Iso}(G,H) = \mathrm{Aut}(G)\tau$? Determine the complete right transversals of $\mathrm{Aut}(F)$, $\mathrm{Aut}(G)$, and $\mathrm{Aut}(H)$ in $\mathfrak{S}_5$.

d. Determine the orbit of all vertices of F in $\mathrm{Aut}(F)$ and the orbit of all vertices of H in $\mathrm{Aut}(H)$.

e. Determine the tower of stabilisers of the subgroups $\mathrm{Aut}(F) \leq \mathfrak{S}_5$ and $\mathrm{Aut}(H) \leq \mathfrak{S}_5$.

f. How many graphs with 5 vertices are isomorphic to F?

7.1-5 The *co-graph* $\overline{G}$ of a graph G is defined by the vertex set $V(\overline{G}) = V(G)$ and the edge set $E(\overline{G}) = \{\{i,j\} \mid i,j \in V(\overline{G})$ and $\{i,j\} \notin E(G)\}$. Prove the following claims: *a.* $\mathrm{Aut}(G) = \mathrm{Aut}(\overline{G})$.

b. $\mathrm{Iso}(G,H) = \mathrm{Iso}(\overline{G},\overline{H})$.

c. $\overline{G}$ is connected if G is not connected.

Step	Alice	Erich	Bob
1	Alice and Bob agree upon a large prime number p and a primitive root r of p; both p and r are public		
2	chooses a large, secret number a at random and computes $\alpha = r^a \bmod p$		chooses a large, secret number b at random and computes $\beta = r^b \bmod p$
3		$\alpha \Rightarrow$ $\Leftarrow \beta$	
4	computes $k_A = \beta^a \bmod p$		computes $k_B = \alpha^b \bmod p$

Figure 7.7. The Diffie-Hellman secret-key agreement protocol.

7.2. Diffie and Hellman's secret-key agreement protocol

The basic number-theoretic facts presented in Subsection 7.1.3 will be needed in this and the subsequent sections. In particular, recall the multiplicative group $\mathbf{Z}_k^*$ from Example 7.3 and Euler's φ function. The arithmetics in remainder class rings will be explained at the end of this chapter, see Problem *7-1*.

Figure 7.7 shows the Diffie-Hellman secret-key agreement protocol, which is based on exponentiation with base r and modulus p, where p is a prime number and r is a primitive root of p. A *primitive root of a number* n is any element $r \in \mathbf{Z}_n^*$ such that $r^d \not\equiv 1 \bmod n$ for each d with $1 \leq d < \varphi(n)$. A primitive root r of n generates the entire group $\mathbf{Z}_n^*$, i.e., $\mathbf{Z}_n^* = \{r^i \,|\, 0 \leq i < \varphi(n)\}$. Recall that for any prime number p the group $\mathbf{Z}_p^*$ has order $\varphi(p) = p - 1$. $\mathbf{Z}_p^*$ has exactly $\varphi(p-1)$ primitive roots, see also Exercise 7.2-1.

Example 7.5 Consider $\mathbf{Z}_5^* = \{1, 2, 3, 4\}$. Since $\mathbf{Z}_4^* = \{1, 3\}$, we have $\varphi(4) = 2$, and the two primitive roots of 5 are 2 and 3. Both 2 and 3 generate all of $\mathbf{Z}_5^*$, since:

$$\begin{array}{llll} 2^0 = 1; & 2^1 = 2; & 2^2 = 4; & 2^3 \equiv 3 \bmod 5\ ; \\ 3^0 = 1; & 3^1 = 3; & 3^2 \equiv 4 \bmod 5; & 3^3 \equiv 2 \bmod 5\ . \end{array}$$

Not every number has a primitive root; 8 is the smallest such example. It is known that a number n has a primitive root if and only if n either is from $\{1, 2, 4\}$, or has the form $n = q^k$ or $n = 2q^k$ for some odd prime number q.

Definition 7.11 (Discrete logarithm). *Let p be a prime number, and let r be a primitive root of p. The* modular exponential function with base r and modulus p *is the function $\mathtt{exp}_{r,p}$ that maps from $\mathbf{Z}_{p-1}$ to $\mathbf{Z}_p^*$, and is defined by $\mathtt{exp}_{r,p}(a) = r^a \bmod p$. Its inverse function is called the* discrete logarithm, *and maps for fixed p and r the value $\mathtt{exp}_{r,p}(a)$ to a. We write $a = \log_r \mathtt{exp}_{r,p}(a) \bmod p$.*

The protocol given in Figure 7.7 works, since (in the arithmetics modulo p)

$$k_A = \beta^a = r^{ba} = r^{ab} = \alpha^b = k_B\ ,$$

so Alice indeed computes the same key as Bob. Computing this key is not hard, since

modular exponentiation can be performed efficiently using the square-and-multiply method from algorithm SQUARE-AND-MULTIPLY.

Erich, however, has a hard time when he attempts to determine their key, since the discrete logarithm is considered to be a hard problem. That is why the modular exponential function $\texttt{exp}_{r,p}$ is a candidate of a *one-way function*, a function that is easy to compute but hard to invert. Note that it is not known whether or not one-way functions indeed exist; this is one of the most challenging open research questions in cryptography. The security of many cryptosystems rests on the assumption that one-way functions indeed exist.

SQUARE-AND-MULTIPLY(a, b, m)

1 $\quad\quad\quad

determine their joint secret key according to the Diffie-Hellman protocol from Figure 7.7; see Exercise 7.2-2.

Note that the protocol is far from being secure in this case, since the prime number $p = 5$ and the secret exponents $a = 17$ and $b = 23$ are much too small. This toy example was chosen just to explain how the protocol works. In practice, a and b should have at least 160 bits each.

If Erich was listening very careful, he knows the values p, r, α, and β after Alice and Bob have executed the protocol. His aim is to determine their joint secret key $k_A = k_B$. This problem is known as the *Diffie-Hellman problem*. If Erich were able to determine $a = \log_r \alpha \bmod p$ and $b = \log_r \beta \bmod p$ efficiently, he could compute the key $k_A = \beta^a \bmod p = \alpha^b \bmod p = k_B$ just like Alice and Bob and thus would have solved the Diffie-Hellman problem. Thus, this problem is no harder than the problem of computing discrete logarithms. The converse question of whether or not the Diffie-Hellman problem is *at least* as hard as solving the discrete logarithm (i.e., whether or not the two problems are equally hard) is still just an unproven conjecture. As many other cryptographic protocols, the Diffie-Hellman protocol currently has no proof of security.

However, since up to date neither the discrete logarithm nor the Diffie-Hellman problem can be solved efficiently, this direct attack is not a practical threat. On the other hand, there do exist other, indirect attacks in which the key is determined not immediately from the values α and β communicated in the Diffie-Hellman protocol. For example, Diffie-Hellman is vulnerable by the "*man-in-the-middle*" attack. Unlike the *passive* attack described above, this attack is *active*, since the attacker Erich aims at actively altering the protocol to his own advantage. He is the "man in the middle" between Alice and Bob, and he intercepts Alice's message $\alpha = r^a \bmod p$ to Bob and Bob's message $\beta = r^b \bmod p$ to Alice. Instead of α and β, he forwards his own values $\alpha_E = r^c \bmod p$ to Bob and $\beta_E = r^d \bmod p$ to Alice, where the private numbers c and d were chosen by Erich. Now, if Alice computes her key $k_A = (\beta_E)^a \bmod p$, which she falsely presumes to share with Bob, k_A in fact is a key for future communications with Erich, who determines the same key by computing (in the arithmetics modulo p)

$$k_E = \alpha^d = r^{ad} = r^{da} = (\beta_E)^a = k_A \ .$$

Similarly, Erich can share a key with Bob, who has not the slightest idea that he in fact communicates with Erich. This raised the issue of *authentication*, which we will deal with in more detail later in Section 7.5 about zero-knowledge protocols.

Exercises

7.2-1 *a.* How many primitive roots do $\mathbf{Z}_{13}^*$ and $\mathbf{Z}_{14}^*$ have?

b. Determine all primitive roots of $\mathbf{Z}_{13}^*$ and $\mathbf{Z}_{14}^*$, and prove that they indeed are primitive roots.

c. Show that every primitive root of 13 and of 14, respectively, generates all of $\mathbf{Z}_{13}^*$ and $\mathbf{Z}_{14}^*$.

7.2-2 *a.* Determine Bob's number $\beta = 3^{23} = 94143178827 \equiv 2 \bmod 5$ from Example 7.6 using the algorithm SQUARE-AND-MULTIPLY.

b. For α and β from Example 7.6, determine the joint secret key of Alice and Bob according to the Diffie-Hellman protocol from Figure 7.7.

Step	Alice	Erich	Bob
1			chooses large prime numbers p and q at random and computes $n = pq$ and $\varphi(n) = (p-1)(q-1)$, his public key (n, e) and his private key d, which satisfy (7.8) and (7.9)
2		$\Leftarrow (n, e)$	
3	encrypts m as $c = m^e \bmod n$		
4		$c \Rightarrow$	
5			decrypts c as $m = c^d \bmod n$

Figure 7.8. The RSA protocol.

7.3. RSA and factoring

7.3.1. RSA

The RSA cryptosystem, which is named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman [200], is the first *public-key* cryptosystem. It is very popular still today and is used by various cryptographic applications. Figure 7.8 shows the single steps of the RSA protocol, which we will now describe in more detail, see also Example 7.7.

1. **Key generation:** Bob chooses two large prime numbers at random, p and q with $p \neq q$, and computes their product $n = pq$. He then chooses an exponent $e \in \mathbf{N}$ satisfying

$$1 < e < \varphi(n) = (p-1)(q-1) \quad \text{and} \quad \gcd(e, \varphi(n)) = 1 \tag{7.8}$$

and computes the inverse of $e \bmod \varphi(n)$, i.e., the unique number d such that

$$1 < d < \varphi(n) \quad \text{and} \quad e \cdot d \equiv 1 \bmod \varphi(n)\ . \tag{7.9}$$

The pair (n, e) is Bob's *public key*, and d is Bob's *private key*.

2. **Encryption:** As in Section 7.1, messages are strings over an alphabet Σ. Any message is subdivided into blocks of a fixed length, which are encoded as positive integers in $|\Sigma|$-adic representation. These integers are then encrypted. Let $m < n$ be the number encoding some block of the message Alice wishes to send to Bob. Alice knows Bob's public key (n, e) and encrypts m as the number $c = E_{(n,e)}(m)$, where the encryption function is defined by

$$E_{(n,e)}(m) = m^e \bmod n\ .$$

3. **Decryption:** Let c with $0 \leq c < n$ be the number encoding one block of the ciphertext, which is received by Bob and also by the eavesdropper Erich. Bob decrypts c by using his private key d and the following decryption function

$$D_d(c) = c^d \bmod n\ .$$

Theorem 7.12 states that the RSA protocol described above indeed is a cryptosystems in the sense of Definition 7.1. The proof of Theorem 7.12 is left to the reader as Exercise 7.3-1.

Theorem 7.12 *Let (n, e) be the public key and d be the private key in the RSA protocol. Then, for each message m with $0 \leq m < n$,*

$$m = (m^e)^d \bmod n \ .$$

Hence, RSA is a public-key cryptosystem.

To make RSA encryption and (authorised) decryption efficient, the algorithm SQUARE-AND-MULTIPLY algorithm is again employed for fast exponentiation.

How should one choose the prime numbers p and q in the RSA protocol? First of all, they must be large enough, since otherwise Erich would be able to factor the number n in Bob's public key (n, e) using the extended Euclidean algorithm. Knowing the prime factors p and q of n, he could then easily determine Bob's private key d, which is the unique inverse of e mod $\varphi(n)$, where $\varphi(n) = (p-1)(q-1)$. To keep the prime numbers p and q secret, they thus must be sufficiently large. In practice, p and q should have at least 80 digits each. To this end, one generates numbers of this size randomly and then checks using one of the known randomised primality tests whether the chosen numbers are primes indeed. By the Prime Number Theorem, there are about $N/\ln N$ prime numbers not exceeding N. Thus, the odds are good to hit a prime after reasonably few trials.

In theory, the primality of p and q can be decided even in *deterministic* polynomial time. Agrawal et al. [1, 2] recently showed that the primality problem, which is defined by

$$\texttt{PRIMES} = \{\mathrm{bin}(n) \mid n \text{ is prime}\} \ ,$$

is a member of the complexity class P. Their breakthrough solved a longstanding open problem in complexity theory: Among a few other natural problems such as the graph isomorphism problem, the primality problem was considered to be one of the rare candidates of problems that are neither in P nor NP-complete.[2] For practical purposes, however, the known randomised algorithms are more useful than the deterministic algorithm by Agrawal et al. The running time of $O(n^{12})$ obtained in their original paper [1, 2] could be improved to $O(n^6)$ meanwhile, applying a more sophisticated analysis, but this running time is still not as good as that of the randomised algorithms.

[2] The complexity classes P and NP will be defined in Section 8.1 and the notion of NP-completeness will be defined in Section 8.2.

MILLER-RABIN(n)

```
1  determine the representation n − 1 = 2^k m, where m and n are odd
2  choose a number z ∈ {1, 2, ..., n − 1} at random under the uniform distribution
3  compute x = z^m mod n
4  if (x ≡ 1 mod n)
5     then return "n is a prime number"
6     else for j ← 0 to k − 1
7          do if (x ≡ −1 mod n)
8                then return "n is a prime number"
9                else  x ← x^2 mod n
10                     return "n is not a prime number"
```

One of the most popular randomised primality tests is the algorithm MILLER-RABIN developed by Rabin [195], which is based on the ideas underlying the deterministic algorithm of Miller [168]. The Miller-Rabin test is a so-called Monte Carlo algorithm, since the "no" answers of the algorithm are always reliable, whereas its "yes" answers have a certain error probability. An alternative to the Miller-Rabin test is the primality test of Solovay and Strassen [231]. Both primality tests run in time $O(n^3)$. However, the Solovay-Strassen test is less popular because it is not as efficient in practice and also less accurate than the Miller-Rabin test.

The class of problems solvable via Monte Carlo algorithms with always reliable "yes" answers is named RP, which stands for *R*andomised *P*olynomial Time. The complementary class, coRP $= \{L \mid \overline{L} \in \text{RP}\}$, contains all those problems solvable via Monte Carlo algorithms with always reliable "no" answers. Formally, RP is defined via nondeterministic polynomial-time Turing machines (NPTMs, for short; see Section 8.1 and in particular Definitions 8.1, 8.2, and 8.3) whose computations are viewed as random processes: For each nondeterministic guess, the machine flips an unbiased coin and follows each of the resulting two next configurations with probability 1/2 until a final configuration is reached. Depending on the number of accepting computation paths of the given NPTM, one obtains a certain acceptance probability for each input. Errors may occur. The definition of RP requires that the error probability must be below the threshold of 1/2 for an input to be accepted, and there must occur no error at all for an input to be rejected.

Definition 7.13 (Randomised polynomial time). *The class* RP *consists of exactly those problems A for which there exists an NPTM M such that for each input x, if $x \in A$ then $M(x)$ accepts with probability at least 1/2, and if $x \notin A$ then $M(x)$ accepts with probability 0.*

Theorem 7.14 `PRIMES` *is in* coRP.

Theorem 7.14 follows from the fact that, for example, the Miller-Rabin test is a Monte Carlo algorithm for the primality problem. We present a proof sketch only. We show that the Miller-Rabin test accepts `PRIMES` with one-sided error probability as in Definition 7.13: If the given number n (represented in binary) is a prime number then the algorithm cannot answer erroneously that n is not prime. For a contradiction,

suppose n is prime but the Miller-Rabin test halts with the output: "n is not a prime number". Hence, $z^m \not\equiv 1 \bmod n$. Since x is squared in each iteration of the `for` loop, we sequentially test the values

$$z^m, z^{2m}, \ldots, z^{2^{k-1}m}$$

modulo n. By assumption, for none of these values the algorithm says n were prime. It follows that for each j with $0 \leq j \leq k-1$,

$$z^{2^j m} \not\equiv -1 \bmod n \; .$$

Since $n - 1 = 2^k m$, Fermat's Little Theorem (see Corollary 7.5) implies $z^{2^k m} \equiv 1 \bmod n$. Thus, $z^{2^{k-1}m}$ is a square roots of 1 modulo n. Since n is prime, there are only two square roots of 1 modulo n, namely $\pm 1 \bmod n$, see Exercise 7.3-1. Since $z^{2^{k-1}m} \not\equiv -1 \bmod n$, we must have $z^{2^{k-1}m} \equiv 1 \bmod n$. But then, $z^{2^{k-2}m}$ again is a square root of 1 modulo n. By the same argument, $z^{2^{k-2}m} \equiv 1 \bmod n$. Repeating this argument again and again, we eventually obtain $z^m \equiv 1 \bmod n$, a contradiction. It follows that the Miller-Rabin test works correctly for each prime number. On the other hand, if n is not a prime number, it can be shown that the error probability of the Miller-Rabin tests does not exceed the threshold of 1/4. Repeating the number of independent trials, the error probability can be made arbitrarily close to zero, at the cost of increasing the running time of course, which still will be polynomially in $\log n$, where $\log n$ is the size of the input n represented in binary.

Example 7.7 [RSA] Suppose Bob chooses the prime numbers $p = 67$ and $q = 11$. Thus, $n = 67 \cdot 11 = 737$, so we have $\varphi(n) = (p-1)(q-1) = 66 \cdot 10 = 660$. If Bob now chooses the smallest exponent possible for $\varphi(n) = 660$, namely $e = 7$, then $(n, e) = (737, 7)$ is his public key. Using the extended Euclidean algorithm, Bob determines his private key $d = 283$, and we have $e \cdot d = 7 \cdot 283 = 1981 \equiv 1 \bmod 660$; see Exercise 7.3-2. As in Section 7.1, we identify the alphabet $\Sigma = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$ with the set $\mathbf{Z}_{26} = \{0, 1, \ldots, 25\}$. Messages are strings over Σ and are encoded in blocks of fixed length as natural numbers in 26-adic representation. In our example, the block length is $\ell = \lfloor \log_{26} n \rfloor = \lfloor \log_{26} 737 \rfloor = 2$.

More concretely, any block $b = b_1 b_2 \cdots b_\ell$ of length ℓ with $b_i \in \mathbf{Z}_{26}$ is represented by the number

$$m_b = \sum_{i=1}^{\ell} b_i \cdot 26^{\ell - i} \; .$$

Since $\ell = \lfloor \log_{26} n \rfloor$, we have

$$0 \leq m_b \leq 25 \cdot \sum_{i=1}^{\ell} 26^{\ell - i} = 26^\ell - 1 < n \; .$$

The RSA encryption function encrypts the block b (i.e., the corresponding number m_b) as $c_b = (m_b)^e \bmod n$. The ciphertext for block b then is $c_b = c_0 c_1 \cdots c_\ell$ with $c_i \in \mathbf{Z}_{26}$. Thus, RSA maps blocks of length ℓ injectively to blocks of length $\ell + 1$. Figure 7.9 shows how to subdivide a message of length 34 into 17 blocks of length 2 and how to encrypt the single blocks, which are represented by numbers. For example, the first block, "RS", is turned into a number as follows: "R" corresponds to 17 and "S" to 18, and we have

$$17 \cdot 26^1 + 18 \cdot 26^0 = 442 + 18 = 460 \; .$$

Message	R S	A I	S T	H E	K E	Y T	O P	U B	L I	C K	E Y	C R	Y P	T O	G R	A P	H Y
m_b	460	8	487	186	264	643	379	521	294	62	128	69	639	508	173	15	206
c_b	697	387	229	340	165	223	586	5	189	600	325	262	100	689	354	665	673

Figure 7.9. Example of an RSA encryption.

Step	Alice	Erich	Bob
1	chooses $n = pq$, her public key (n, e) and her private key d just as Bob does in the RSA protocol in Figure 7.8		
2		$(n, e) \Rightarrow$	
3	signs the message m by $\mathrm{sig}_A(m) = m^d \bmod n$		
4		$(m, \mathrm{sig}_A(m)) \Rightarrow$	
5			checks $m \equiv (\mathrm{sig}_A(m))^e \bmod n$ to verify Alice's signature

Figure 7.10. Digital RSA signatures.

The resulting number c_b is written again in 26-adic representation and can have the length $\ell+1$: $c_b = \sum_{i=0}^{\ell} c_i \cdot 26^{\ell-i}$, where $c_i \in \mathbf{Z}_{26}$, see also Exercise 7.3-2. So, the first block, $697 = 676 + 21 = 1 \cdot 26^2 + 0 \cdot 26^1 + 21 \cdot 26^0$, is encrypted to yield the ciphertext "BAV".

Decryption is also done blockwise. In order to decrypt the first block with the private key $d = 283$, compute $697^{283} \bmod 737$, again using fast exponentiation with Square-and-Multiply. To prevent the numbers from becoming too large, it is recommendable to reduce modulo $n = 737$ after each multiplication. The binary expansion of the exponent is $283 = 2^0 + 2^1 + 2^3 + 2^4 + 2^8$, and we obtain

$$697^{283} \equiv 697^{2^0} \cdot 697^{2^1} \cdot 697^{2^3} \cdot 697^{2^4} \cdot 697^{2^8} \equiv 697 \cdot 126 \cdot 9 \cdot 81 \cdot 15 \equiv 460 \bmod 737$$

as desired.

7.3.2. Digital RSA signatures

The public-key cryptosystem RSA from Figure 7.8 can be modified so as to produce digital signatures. This protocol is shown in Figure 7.10. It is easy to see that this protocol works as desired; see Exercise 7.3-2. This digital signature protocol is vulnerable to "*chosen-plaintext attacks*" in which the attacker can choose a plaintext and obtains the corresponding ciphertext. This attack is described, e.g., in [203].

7.3.3. Security of RSA

As mentioned above, the security of the RSA cryptosystem crucially depends on the assumption that large numbers cannot be factored in a reasonable amount of time. Despite much effort in the past, no efficient factoring algorithm has been found until now. Thus, it is widely believed that there is no such algorithm and the factoring problem is hard. A rigorous proof of this hypothesis, however, has not been found either. And even if one could prove this hypothesis, this would not imply a proof

of security of RSA. Breaking RSA is at most as hard as the factoring problem; however, the converse is not known to hold. That is, it is not known whether these two problems are equally hard. It may be possible to break RSA without factoring n.

We omit listing potential attacks on the RSA system here. Rather, the interested reader is pointed to the comprehensive literature on this subject; note also Problem 7-4 at the end of this chapter. We merely mention that for each of the currently known attacks on RSA, there are suitable countermeasures, rules of thumb that either completely prevent a certain attack or make its probability of success negligibly small. In particular, it is important to take much care when choosing the prime numbers p and q, the modulus $n = pq$, the public exponent e, and the private key d.

Finally, since the *factoring attacks* on RSA play a particularly central role, we briefly sketch two such attacks. The first one is based on Pollard's $(p-1)$ method [193]. This method is effective for composite numbers n having a prime factor p such that the prime factors of $p-1$ each are small. Under this assumption, a multiple ν of $p-1$ can be determined without knowing p. By Fermat's Little Theorem (see Corollary 7.5), it follows that $a^\nu \equiv 1 \bmod p$ for all integers a coprime with p. Hence, p divides $a^\nu - 1$. If n does not divide $a^\nu - 1$, then $\gcd(a^\nu - 1, n)$ is a nontrivial divisor of n. Thus, the number n can be factored.

How can the multiple ν of $p-1$ be determined? Pollard's $(p-1)$ method uses as candidates for ν the products of prime powers below a suitably chosen bound S:

$$\nu = \prod_{q \text{ is prime, } q^k \le S} q^k \ .$$

If all prime powers dividing $p-1$ are less than S, then ν is a multiple of $p-1$. The algorithm determines $\gcd(a^\nu - 1, n)$ for a suitably chosen base a. If no nontrivial divisor of n is found, the algorithm is restarted with a new bound $S' > S$.

Other factoring methods, such as the *quadratic sieve*, are described, e.g., in [205, 234]. They use the following simple idea. Suppose n is the number to be factored. Using the sieve, determine numbers a and b such that:

$$a^2 \equiv b^2 \bmod n \quad \text{and} \quad a \not\equiv \pm b \bmod n \ . \tag{7.10}$$

Hence, n divides $a^2 - b^2 = (a-b)(a+b)$ but neither $a-b$ nor $a+b$. Thus, $\gcd(a-b, n)$ is a nontrivial factor of n.

There are also sieve methods other than the quadratic sieve. These methods are distinguished by the particular way of how to determine the numbers a and b such that (7.10) is satisfied. A prominent example is the "general number field sieve", see [148].

Exercises

7.3-1 *a.* Prove Theorem 7.12. *Hint.* Show $(m^e)^d \equiv m \bmod p$ and $(m^e)^d \equiv m \bmod q$ using Corollary 7.5, Fermat's Little Theorem. Since p and q are prime numbers with $p \neq q$ and $n = pq$, the claim $(m^e)^d \equiv m \bmod n$ now follows from the Chinese remainder theorem.

b. The proof sketch of Theorem 7.14 uses the fact that any prime number n can have only two square roots of 1 modulo n, namely $\pm 1 \bmod n$. Prove this fact. *Hint.*

Step	Alice	Erich	Bob
1	chooses two large numbers x and y at random, keeps x secret and computes $x\sigma y$		
2		$(y, x\sigma y) \Rightarrow$	
3			chooses a large number z at random, keeps z secret and computes $y\sigma z$
4		$\Leftarrow y\sigma z$	
5	computes $k_A = x\sigma(y\sigma z)$		computes $k_B = (x\sigma y)\sigma z$

Figure 7.11. The Rivest-Sherman protocol for secret-key agreement, based on σ.

It may be helpful to note that r is a square root of 1 modulo n if and only if n divides $(r-1)(r+1)$.

7.3-2 *a.* Let $\varphi(n) = 660$ and $e = 7$ be the values from Example 7.7. Show that the extended Euclidean algorithm indeed provides the private key $d = 283$, the inverse of 7 mod 660.

b. Consider the plaintext in Figure 7.9 from Example 7.7 and its RSA encryption. Determine the encoding of this ciphertext by letters of the alphabet $\Sigma = \{\mathrm{A}, \mathrm{B}, \ldots, \mathrm{Z}\}$ for each of the 17 blocks.

c. Decrypt each of the 17 ciphertext blocks in Figure 7.9 and show that the original message is obtained indeed.

d. Prove that the RSA digital signature protocol from Figure 7.10 works.

7.4. The protocols of Rivest, Rabi, and Sherman

Rivest, Rabi, and Sherman proposed protocols for secret-key agreement and digital signatures. The secret-key agreement protocol given in Figure 7.11 is due to Rivest and Sherman. Rabi and Sherman modified this protocol to a digital signature protocol, see Exercise 7.4-1.

The Rivest-Sherman protocol is based on a *total, strongly noninvertible, associative one-way function*. Informally put, a *one-way function* is a function that is easy to compute but hard to invert. One-way functions are central cryptographic primitives and many cryptographic protocols use them as their key building blocks. To capture the notion of noninvertibility, a variety of models and, depending on the model used, various candidates for one-way functions have been proposed. In most cryptographic applications, noninvertibility is defined in the average-case complexity model. Unfortunately, it is not known whether such one-way functions exist; the security of the corresponding protocols is merely based on the assumption of their existence. Even in the less challenging worst-case model, in which so-called "complexity-theoretic" one-way functions are usually defined, the question of whether any type of one-way function exists remains an open issue after many years of research.

A total (i.e., everywhere defined) function σ mapping from $\mathbf{N} \times \mathbf{N}$ to $\mathbf{N}$ is ***associative*** if and only if $(x\sigma y)\sigma z = x\sigma(y\sigma z)$ holds for all $x, y, z \in \mathbf{N}$, where

we use the infix notation $x\sigma y$ instead of the prefix notation $\sigma(x, y)$. This property implies that the above protocol works:

$$k_A = x\sigma(y\sigma z) = (x\sigma y)\sigma z = k_B \ ,$$

so Alice and Bob indeed compute the same secret key.

The notion of strong noninvertibility is not to be defined formally here. Informally put, σ is said to be *strongly noninvertible* if σ is not only a one-way function, but even if in addition to the function value one of the corresponding arguments is given, it is not possible to compute the other argument efficiently. This property implies that the attacker Erich, knowing y and $x\sigma y$ and $y\sigma z$, is not able to compute the secret numbers x and z, from which he could easily determine the secret key $k_A = k_B$.

Exercises

7.4-1 Modify the Rivest-Sherman protocol for secret-key agreement from Figure 7.11 to a protocol for digital signatures.

7.4-2 *(a.* Try to give a formal definition of the notion of "strong noninvertibility" that is defined only informally above. Use the worst-case complexity model.

b. Suppose σ is a *partial* function from $\mathbf{N} \times \mathbf{N}$ to $\mathbf{N}$, i.e., σ may be undefined for some pairs in $\mathbf{N} \times \mathbf{N}$. Give a formal definition of "associativity" for partial functions. What is wrong with the following (flawed) attempt of a definition: "A partial function $\sigma : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ is said to be *associative* if and only if $x\sigma(y\sigma z) = (x\sigma y)\sigma z$ holds for all $x, y, z \in \mathbf{N}$ for which each of the four pairs (x, y), (y, z), $(x, y\sigma z)$, and $(x\sigma y, z)$ is in the domain of σ." *Hint.* A comprehensive discussion of these notions can be found in [105, 107, 108].

7.5. Interactive proof systems and zero-knowledge

7.5.1. Interactive proof systems, Arthur-Merlin games, and zero-knowledge protocols

In Section 7.2, the "*man-in-the-middle*" attack on the Diffie-Hellman protocol was mentioned. The problem here is that Bob has not verified the true identity of his communication partner before executing the protocol. While he assumes to communicate with Alice, he in fact exchanges messages with Erich. In other words, Alice's task is to convince Bob of her true identity without any doubt. This cryptographic task is called *authentication*. Unlike digital signatures, whose purpose is to authenticate electronically transmitted *documents* such as emails, electronic contracts, etc., the goal now is to authenticate *individuals* such as human or computer parties participating in a cryptographic protocol.[3]

In order to authenticate herself, Alice might try to prove her identity by a secret information known to her alone, say by giving her PIN ("*P*ersonal *I*dentifaction *N*umber") or any other private information that no one knows but her. However,

[3] Here, an "individual" or a "party" is not necessarily a human being; it may also be a computer program that automatically executes a protocol with another computer program.

there is a catch. To prove her identity, she would have to give her secret away to Bob. But then, it no longer is a secret! Bob, knowing her secret, might pretend to be Alice in another protocol he executes with Chris, a third party. So the question is how to prove knowledge of a secret without giving it away. This is what zero-knowledge is all about. Zero-knowledge protocols are special interactive proof systems, which were introduced by Goldwasser, Micali, and Rackoff and, independently, by Babai and Moran. Babai and Moran's notion (which is essentially equivalent to the interactive proof systems proposed by Goldwasser et al.) is known as Arthur-Merlin games, which we will now describe informally.

Merlin and Arthur wish to jointly solve a problem L, i.e., they wish to jointly decide whether or not a given input x belongs to L. The mighty wizard Merlin is represented by an NP machine M, and the impatient King Arthur is represented by a randomised polynomial-time Turing machine A. To make their decision, they play the following game, where they are taking turns to make alternating moves. Merlin's intention is always to convince Arthur that x belongs to L (no matter whether or not that indeed is the case). Thus, each of Merlin's moves consists in presenting a proof for "$x \in L$", which he obtains by simulating $M(x, y)$, where x is the input and y describes all previous moves in this game. That is, the string y encodes all previous nondeterministic choices of M and all previous random choices of A.

King Arthur, however, does not trust Merlin. Of course, he cannot check the mighty wizard's proofs all alone; this task simply exceeds his computational power. But he knows Merlin well enough to express some doubts. So, he replies with a nifty challenge to Merlin by picking some details of his proofs at random and requiring certificates for them that he can verify. In order to satisfy Arthur, Merlin must convince him with overwhelming probability. Thus, each of Arthur's moves consists in the simulation of $A(x, y)$, where x again is the input and y describes the previous history of the game.

The idea of Arthur-Merlin games can be captured via alternating existential and probabilistic quantifiers, where the former formalise Merlin's NP computation and the latter formalise Arthur's randomised polynomial-time computation.[4] In this way, a hierarchy of complexity classes can be defined, the so-called Arthur-Merlin hierarchy. We here present only the class MA from this hierarchy, which corresponds to an Arthur-Merlin game with two moves, with Merlin moving first.

Definition 7.15 (MA in the Arthur-Merlin hierarchy). *The class* MA *contains exactly those problems L for which there exists an* NP *machine M and a randomised polynomial-time Turing machine A such that for each input x:*

- *If $x \in L$ then there exists a path y of $M(x)$ such that $A(x, y)$ accepts with probability at least $3/4$ (i.e., Arthur cannot refute Merlin's proof y for "$x \in L$", and Merlin thus wins).*
- *If $x \notin L$ then for each path y of $M(x)$, $A(x, y)$ rejects with probability at least $3/4$ (i.e., Arthur is not taken in by Merlin's wrong proofs for "$x \in L$" and thus wins).*

Analogously, the classes $\mathrm{AM}, \mathrm{MAM}, \mathrm{AMA}, \ldots$ *can be defined, see Exercise 7.5-1.*

[4] This is similar to the well-known characterisation of the levels of the polynomial hierarchy via alternating $\exists$ and $\forall$ quantifiers, see Section 8.4 and in particular item 3 Theorem 8.11.

In Definition 7.15, the probability threshold of 3/4 for Arthur to accept or to reject, respectively, is chosen at will and does not appear to be large enough at first glance. In fact, the probability of success can be amplified using standard techniques and can be made arbitrarily close to one. In other words, one might have chosen even a probability threshold as low as $1/2 + \varepsilon$, for an arbitrary fixed constant $\varepsilon > 0$, and would still have defined the same class. Furthermore, it is known that for a constant number of moves, the Arthur-Merlin hierarchy collapses down to AM:

$$\mathrm{NP} \subseteq \mathrm{MA} \subseteq \mathrm{AM} = \mathrm{AMA} = \mathrm{MAM} = \cdots .$$

It is an open question whether or not any of the inclusions $\mathrm{NP} \subseteq \mathrm{MA} \subseteq \mathrm{AM}$ is a strict one.

A similar model, which can be used as an alternative to the Arthur-Merlin games, are the *interactive proof systems* mentioned above. The two notions use different terminology: Merlin now is called the "prover" and Arthur the "verifier". Also, their communication is not interpreted as a game but rather as a protocol. Another difference between the two models, which appears to be crucial at first, is that Arthur's random bits are public (so, in particular, Merlin knows them), whereas the random bits of the verifier in an interactive proof system are private. However, Goldwasser and Sipser [89] proved that, in fact, it does not matter whether the random bits are private or public, so Arthur-Merlin games essentially are equivalent to interactive proof systems.

If one allows a polynomial number of moves (instead of a constant number), then one obtains the complexity class IP. Note that interactive proof systems are also called IP protocols. By definition, IP contains all of NP. In particular, the graph isomorphism problem is in IP. We will see later that IP also contains problems from $\mathrm{coNP} = \{\overline{L} \mid L \in \mathrm{NP}\}$ that are supposed to be not in NP. In particular, the proof of Theorem 8.16 shows that the complement of the graph isomorphism problem is in AM and thus in IP. A celebrated result by Shamir [219] says that IP equals PSPACE, the class of problems solvable in polynomial space.

Let us now turn back to the problem of authentication mentioned above, and to the related notion of zero-knowledge protocols. Here is the idea. Suppose Arthur and Merlin play one of their games. So, Merlin sends hard proofs to Arthur. Merlin alone knows how to get such proofs. Being a wise wizard, he keeps this knowledge secret. And he uses his secret to authenticate himself in the communication with Arthur.

Now suppose that malicious wizard Marvin wishes to fool Arthur by pretending to be Merlin. He disguises as Merlin and uses his magic to look just like him. However, he does not know Merlin's secret of how to produce hard proofs. His magic is no more powerful than that of an ordinary randomised polynomial-time Turing machine. Still, he seeks to simulate the communication between Merlin and Arthur. An interactive proof system has the ***zero-knowledge property*** if the information communicated between Marvin and Arthur cannot be distinguished from the information communicated between Merlin and Arthur. Note that Marvin, who does not know Merlin's secret, cannot introduce any information about this secret into the simulated IP protocol. Nonetheless, he is able to perfectly copy the original protocol, so no one can tell a difference. Hence, the (original) protocol itself must have the

property that it does not leak any information whatsoever about Merlin's secret. If there is nothing to put in, there can be nothing to take out.

Definition 7.16 (Zero-knowledge protocol). *Let L be any set in* IP*, and let (M, A) be an interactive proof system for L, where M is an NPTM and A is a randomised polynomial-time Turing machine. The* IP *protocol (M, A) is a* zero-knowledge protocol for L *if and only if there exists a randomised polynomial-time Turing machine such that $(\widehat{M}, A)$ simulates the original protocol (M, A) and, for each $x \in L$, the tuples $(m_1, m_2, \ldots, m_k)$ and $(\widehat{m}_1, \widehat{m}_2, \ldots, \widehat{m}_k)$ representing the information communicated in (M, A) and in $(\widehat{M}, A)$, respectively, are identically distributed over the random choices in (M, A) and in $(\widehat{M}, A)$, respectively.*

The notion defined above is called "*honest-verifier perfect zero-knowledge*" in the literature, since (a) it is assumed that the verifier is *honest* (which may not necessarily be true in cryptographic applications though), and (b) it is required that the information communicated in the simulated protocol *perfectly* coincides with the information communicated in the original protocol. Assumption (a) may be somewhat too idealistic, and assumption (b) may be somewhat too strict. That is why also other variants of zero-knowledge protocols are studied, see the notes at the end of this chapter.

7.5.2. Zero-knowledge protocol for graph isomorphism

Let us consider a concrete example now. As mentioned above, the graph isomorphism problem (GI, for short) is in NP, and the complementary problem $\overline{\mathtt{GI}}$ is in AM, see the proof of Theorem 8.16. Thus, both problems are contained in IP. We now describe a zero-knowledge protocol for GI that is due to Goldreich, Micali, and Wigderson [84]. Figure 7.12 shows this IP protocol between the prover Merlin and the verifier Arthur.

Although there is no efficient algorithm known for GI, Merlin can solve this problem, since GI is in NP. However, there is no need for him to do so in the protocol. He can simply generate a large graph G_0 with n vertices and a random permutation $\pi \in \mathfrak{S}_n$. Then, he computes the graph $G_1 = \pi(G_0)$ and makes the pair (G_0, G_1) public. The isomorphism π between G_0 and G_1 is kept secret as Merlin's private information.

Of course, Merlin cannot send π to Arthur, since he does not want to give his secret away. Rather, to prove that the two graphs, G_0 and G_1, indeed are isomorphic, Merlin randomly chooses an isomorphism ρ under the uniform distribution and a bit $a \in \{0, 1\}$ and computes the graph $H = \rho(G_a)$. He then sends H to Arthur whose response is a challenge for Merlin: Arthur sends a random bit $b \in \{0, 1\}$, chosen under the uniform distribution, to Merlin and requests to see an isomorphism σ between G_b and H. Arthur accepts if and only if σ indeed satisfies $\sigma(G_b) = H$.

The protocol works, since Merlin knows his secret isomorphism π and his random permutation ρ: It is no problem for Merlin to compute the isomorphism σ between G_b and H and thus to authenticate himself. The secret π is not given away. Since G_0 and G_1 are isomorphic, Arthur accepts with probability one. The case of two nonisomorphic graphs does not need to be considered here, since Merlin has chosen isomorphic graphs G_0 and G_1 in the protocol; see also the proof of Theorem 8.16.

Step	Merlin		Arthur
1	randomly chooses a permutation ρ on $V(G_0)$ and a bit a, computes $H = \rho(G_a)$		
2		$H \Rightarrow$	
3			chooses a random bit b and requests to see an isomorphism between G_b and H
4		$\Leftarrow b$	
5	computes the isomorphism σ satisfying $\sigma(G_b) = H$ as follows: if $b = a$ then $\sigma = \rho$; if $0 = b \neq a = 1$ then $\sigma = \pi\rho$; if $1 = b \neq a = 0$ then $\sigma = \pi^{-1}\rho$.		
6		$\sigma \Rightarrow$	
7			verifies that $\sigma(G_b) = H$ and accepts accordingly

Figure 7.12. Goldreich, Micali, and Wigderson's zero-knowledge protocol for `GI`.

Step	Marvin		Arthur
1	randomly chooses a permutation ρ on $V(G_0)$ and a bit a, computes $H = \rho(G_a)$		
2		$H \Rightarrow$	
3			chooses a random bit b and requests to see an isomorphism between G_b and H
4		$\Leftarrow b$	
5	if $b \neq a$ then $\widehat{M}$ deletes all information communicated in this round; if $b = a$ then $\widehat{M}$ sends $\sigma = \rho$		
6		$\sigma \Rightarrow$	
7			$b = a$ implies $\sigma(G_b) = H$, so Arthur accepts Marvin's faked identity

Figure 7.13. Simulation of the zero-knowledge protocol for `GI` without knowing π.

Now, suppose, Marvin wishes to pretend to be Merlin when communicating with Arthur. He does know the graphs G_0 and G_1, but he doesn't know the secret isomorphism π. Nonetheless, he tries to convince Arthur that he does know π. If Arthur's random bit b happens to be the same as his bit a, to which Marvin committed *before* he sees b, then Marvin wins. However, if $b \neq a$, then computing $\sigma = \pi\rho$ or $\sigma = \pi^{-1}\rho$ requires knowledge of π. Since `GI` is not efficiently solvable (and even too hard for a randomised polynomial-time Turing machine), Marvin cannot determine the isomorphism π for sufficiently large graphs G_0 and G_1. But without knowing π, all he can do is guess. His chances of hitting a bit b with $b = a$ are at most $1/2$. Of course, Marvin can always guess, so his success probability is exactly $1/2$. If Arthur challenges him in r independent rounds of this protocol again and again, the probability of Marvin's success will be only 2^{-r}. Already for $r = 20$, this probability is negligibly small: Marvin's probability of success is then less than one in one million.

It remains to show that the protocol from Figure 7.12 is a zero-knowledge protocol. Figure 7.13 shows a simulated protocol with Marvin who does not know Merlin's secret π but pretends to know it. The information communicated in one round of the protocol has the form of a triple: (H, b, σ). If Marvin is lucky enough to choose a random bit a with $a = b$, he can simply send $\sigma = \rho$ and wins: Arthur (or any third party watching the communication) will not notice the fraud. On the other hand, if $a \neq b$ then Marvin's attempt to betray will be uncovered. However, that is no problem for the malicious wizard: He simply deletes this round from the protocol and repeats. Thus, he can produce a sequence of triples of the form (H, b, σ) that is indistinguishable from the corresponding sequence of triples in the original protocol between Merlin and Arthur. It follows that Goldreich, Micali, and Wigderson's protocol for `GI` is a zero-knowledge protocol.

Exercises

7.5-1 Arthur-Merlin hierarchy:

a. Analogously to MA from Definition 7.15, define the other classes AM, MAM, AMA, ... of the Arthur-Merlin hierarchy.

b. What is the inclusion structure between the classes MA, coMA, AM, coAM, and the classes of the polynomial hierarchy defined in Definition 8.10 of Section 8.4.1.

7.5-2 Zero-knowledge protocol for graph isomorphism:

a. Consider the graphs G and H from Example 7.4 in Section 7.1.3. Execute the zero-knowledge protocol from Figure 7.12 with the graphs $G_0 = G$ and $G_1 = H$ and the isomorphism $\pi = \left(\begin{smallmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 1 & 5 & 2 \end{smallmatrix}\right)$. Use an isomorphism ρ of your choice, and try all possibilities for the random bits a and b. Repeat this Arthur-Merlin game with an unknown isomorphism ρ chosen by somebody else.

b. Modify the protocol from Figure 7.12 such that, with only one round of the protocol, Marvin's success probability is less than 2^{-25}.

Problems

7-1 Arithmetics in the ring $\mathbf{Z}_k$

Let $k \in \mathbf{N}$ and $x, y, z \in \mathbf{Z}$. We say x *is congruent to* y *modulo* k ($x \equiv y \bmod k$, for short) if and only if k divides the difference $y - x$. For example, $-10 \equiv 7 \bmod 17$ and $4 \equiv 0 \bmod 2$. The congruence $\equiv$ modulo k defines an *equivalence relation* on $\mathbf{Z}$, i.e., it is *reflexive* ($x \equiv x \bmod k$), *symmetric* (if $x \equiv y \bmod k$ then $y \equiv x \bmod k$), and *transitive* ($x \equiv y \bmod k$ and $y \equiv z \bmod k$ imply $x \equiv z \bmod k$).

The set $x + k\mathbf{Z} = \{y \in \mathbf{Z} \mid y \equiv x \bmod k\}$ is the *remainder class of* $x \bmod k$. For example, the remainder class of 3 mod 7 is

$$3 + 7\mathbf{Z} = \{3, 3 \pm 7, 3 \pm 2 \cdot 7, \ldots\} = \{3, 10, -4, 17, -11, \ldots\} .$$

We represent the remainder class of $x \bmod k$ by the smallest natural number in $x + k\mathbf{Z}$. For instance, 3 represents the remainder class of 3 mod 7. The set of all remainder classes $\bmod k$ is $\mathbf{Z}_k = \{0, 1, \ldots, k-1\}$. On $\mathbf{Z}_k$, *addition* is defined by

$(x+k\mathbf{Z})+(y+k\mathbf{Z}) = (x+y)+k\mathbf{Z}$, and *multiplication* is defined by $(x+k\mathbf{Z})\cdot(y+k\mathbf{Z}) = (x\cdot y)+k\mathbf{Z}$. For example, in the arithmetics modulo 7, we have $(4+7\mathbf{Z})+(5+7\mathbf{Z}) = (4+5)+7\mathbf{Z} = 2+7\mathbf{Z}$ and $(4+7\mathbf{Z})\cdot(5+7\mathbf{Z}) = (4\cdot 5)+7\mathbf{Z} = 6+7\mathbf{Z}$.

Prove that in the arithmetics modulo k:

a. $(\mathbf{Z}_k, +, \cdot)$ is a commutative ring with one;

b. $\mathbf{Z}_k^*$, which is defined in Example 7.3, is a multiplicative group;

c. $(\mathbf{Z}_p, +, \cdot)$ is a field for each prime number p.

d. Prove that the neutral element of a group and the inverse of each group element are unique.

e. Let $\mathfrak{R}$ be a commutative ring with one. Prove that the set of all invertible elements in $\mathfrak{R}$ forms a multiplicative group. Determine this group for the ring $\mathbf{Z}_k$.

7-2 Tree isomorphism

The graph isomorphism problem can be solved efficiently on special graph classes, such as on the class of trees. An (undirected) *tree* is a connected graph without cycles. (A *cycle* is a sequence of pairwise incident edges that returns to the point of origin.) The *leaves of a tree* are the vertices with degree one. The *tree isomorphism problem* is defined by

$$\texttt{TI} \quad = \quad \{(G,H) \mid G \text{ and } H \text{ are isomorphic trees}\} \,.$$

Design an efficient algorithm for this problem.

Hint. Label the vertices of the given pair of trees successively by suitable number sequences. Compare the resulting sequences of labels in the single loops of the algorithm. Starting from the leaves of the trees and then working step by step towards the center of the trees, the algorithm halts as soon as all vertices are labelled. This algorithm can be found, e.g., in [129].

7-3 Computing the determinant

Design an efficient algorithm in pseudocode for computing the determinant of a matrix. Implement your algorithm in a programming language of your choice. Can the inverse of a matrix be computed efficiently?

7-4 Low-exponent attack

a. Consider the RSA cryptosystem from Figure 7.8. For the sake of efficiency, the public exponent $e = 3$ has been popular. However, this choice is dangerous. Suppose Alice, Bob, and Chris encrypt the same message m with the same public exponent $e = 3$, but perhaps with distinct moduli, n_A, n_B, and n_C. Erich intercepts the resulting three ciphertexts: $c_i = m^3 \bmod n_i$ for $i \in \{A, B, C\}$. Then, Erich can easily decrypt the message m. How?

Hint. Erich knows the Chinese remainder theorem, which also was useful in Exercise 7.3-1.

A recommended value for the public exponent is $e = 2^{16} + 1$, since its binary expansion has only two ones. Thus, the square-and-multiply algorithm runs fast for this e.

b. The attack described above can be extended to k ciphertexts that are related with each other as follows. Let a_i and b_i be known, $1 \leq i \leq k$, and suppose that k messages $c_i = (a_i m + b_i)^e \bmod n_i$ are sent and are intercepted by Erich. Further, suppose that $k > e(e+1)/2$ and $\min(n_i) > 2^{e^2}$. How can attacker Erich now determine the original message m?

Hint. Apply so-called lattice reduction techniques (see, e.g., Micciancio and Goldwasser [165]). The attack mentioned here is due to Håstad [97] and has been strengthened later by Coppersmith [50].

c. How can the attacks described above be prevented?

Chapter notes

Singh's book [227] gives a nice introduction to the history of cryptology, from its ancient roots to modern cryptosystems. For example, you can find out there about recent discoveries related to the development of RSA and Diffie-Hellman in the nonpublic sector. Ellis, Cocks, and Williamson from the Communications Electronics Security Group (CESG) of the British Government Communications Head Quarters (GCHQ) proposed the RSA system from Figure 7.8 and the Diffie-Hellman protocol from Figure 7.7 even earlier than Rivest, Shamir, and Adleman and at about the same time as but independent of Diffie and Hellman, respectively. RSA and Diffie-Hellman are described in probably every book about cryptography written since their invention. A more comprehensive list of attacks against RSA than that of Section 7.3 can be found in, e.g., [27, 124, 173, 203, 205, 220].

Primality tests such as the Miller-Rabin test and factoring algorithms are also described in many books, e.g., in [86, 205, 209, 234].

The notion of strongly noninvertible associative one-way functions, on which the secret-key agreement protocol from Figure 7.11 is based, is due to Rivest and Sherman. The modification of this protocol to a digital signature scheme is due to Rabi and Sherman. In their paper [194], they also proved that commutative, associative one-way function exist if and only if P $\neq$ NP. However, the one-way functions they construct are neither total nor strongly noninvertible, even if P $\neq$ NP is assumed. Hemaspaandra and Rothe [108] proved that total, strongly noninvertible, commutative, associative one-way functions exist if and only if P $\neq$ NP. Further investigations on this topic can be found in [26, 105, 107, 110].

The notion of interactive proof systems and zero-knowledge protocols is due to Goldwasser, Micali, and Rackoff [88]. One of the best and most comprehensive sources on this field is Chapter 4 in Goldreich's book [86]; see also the books [131, 184, 205] and the surveys [87, 85, 203]. Arthur-Merlin games were introduced by Babai and Moran [18, 17] and have been investigated in many subsequent papers. Variants of the notion of zero-knowledge, which differ from the notion in Definition 7.16 in their technical details, are extensively discussed in, e.g., [86] and also in, e.g., [85, 87, 203].

8. Complexity Theory

In Chapter 7, efficient algorithms were introduced that are important for cryptographic protocols. Designing efficient algorithms of course is a central task in all areas of computer science. Unfortunately, many important problems have resisted all attempts in the past to devise efficient algorithms solving them. Well-known examples of such problems are the satisfiability problem for boolean formulas and the graph isomorphism problem.
One of the most important tasks in complexity theory is to classify such problems according to their computational complexity. Complexity theory and algorithmics are the two sides of the same medal; they complement each other in the following sense. While in algorithmics one seeks to find the best *upper bound* for some problem, an important goal of complexity theory is to obtain the best possible *lower bound* for the same problem. If the upper and the lower bound coincide, the problem has been classified.

The proof that some problem cannot be solved efficiently often appears to be "negative" and not desirable. After all, we wish to solve our problems and we wish to solve them fast. However, there is also some "positive" aspect of proving lower bounds that, in particular, is relevant in cryptography (see Chapter 7). Here, we are interested in the applications of inefficiency: A proof that certain problems (such as the factoring problem or the discrete logarithm) cannot be solved efficiently can support the security of some cryptosystems that are important in practical applications.

In Section 8.1, we provide the foundations of complexity theory. In particular, the central complexity classes P and NP are defined. The question of whether or not these two classes are equal is still open after more than three decades of intense research. Up to now, neither a proof of the inequality $\mathrm{P} \neq \mathrm{NP}$ (which is widely believed) could be achieved, nor were we able to prove the equality of P and NP. This question led to the development of the beautiful and useful theory of NP-completeness.

One of the best understood NP-complete problems is `SAT`, the satisfiability problem of propositional logic: Given a boolean formula φ, does there exist a satisfying assignment for φ, i.e., does there exist an assignment of truth values to φ's variables that makes φ true? Due to its NP-completeness, it is very unlikely that there exist efficient deterministic algorithms for `SAT`. In Section 8.3, we present a deterministic and a randomised algorithm for `SAT` that both run in exponential time. Even though

these algorithms are *asymptotically inefficient* (which is to say that they are useless in practice for *large* inputs), they are useful for sufficiently small inputs of sizes still relevant in practice. That is, they outperform the naive deterministic exponential-time algorithm for SAT in that they considerably increase the input size for which the algorithm's running time still is tolerable.

In Section 8.4, we come back to the graph isomorphism problem, which was introduced in Section 7.1.3 (see Definition 7.7) and which was useful in Section 7.5.2 with regard to the zero-knowledge protocols. This problem is one of the few natural problems in NP, which (under the plausible assumption that P $\neq$ NP) may be neither efficiently solvable nor be NP-complete. In this regard, this problem is special among the problems in NP. Evidence for the hypothesis that the graph isomorphism problem may be neither in P nor NP-complete comes from the theory of *lowness*, which is introduced in Section 8.4. In particular, we present Schöning's result that GI is contained in the low hierarchy within NP. This result provides strong evidence that GI is not NP-complete. We also show that GI is contained in the complexity class SPP and thus is low for certain probabilistic complexity classes. Informally put, a set is *low* for a complexity class $\mathcal{C}$ if it does not provide any useful information when used as an "oracle" in $\mathcal{C}$ computations. For proving the lowness of GI, certain group-theoretic algorithms are useful.

8.1. Foundations

As mentioned above, complexity theory is concerned with proving lower bounds. The difficulty in such proofs is that it is not enough to analyse the runtime of just *one* concrete algorithm for the problem considered. Rather, one needs to show that *every* algorithm solving the problem has a runtime no better than the lower bound to be proven. This includes also algorithms that have not been found as yet. Hence, it is necessary to give a formal and mathematically precise definition of the notion of algorithm.

Since the 1930s, a large variety of formal algorithm models has been proposed. All these models are equivalent in the sense that each such model can be transformed (via an algorithmic procedure) into any other such model. Loosely speaking, one might consider this transformation as some sort of compilation between distinct programming languages. The equivalence of all these algorithm models justifies Church's thesis, which says that each such model captures precisely the somewhat vague notion of "intuitive computability". The algorithm model that is most common in complexity theory is the Turing machine, which was introduced in 1936 by Alan Turing (1912 until 1954) in his pathbreaking work [244]. The Turing machine is a very simple abstract model of a computer. In what follows, we describe this model by defining its syntax and its semantics, and introduce at the same time two basic computation paradigms: determinism and nondeterminism. It makes sense to first define the more general model of nondeterministic Turing machines. Deterministic Turing machines then are easily seen to be a special case.

First, we give some technical details and describe how Turing machines work. A Turing machine has k infinite work tapes that are subdivided into cells. Every cell

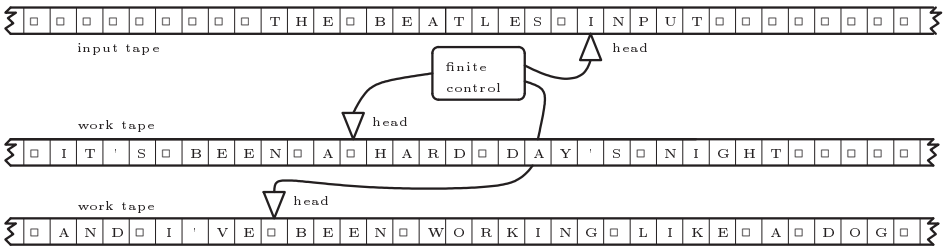


Figure 8.1. A Turing machine with one input and two work tapes.

may contain one letter of the work alphabet. If a cell does not contain a letter, we indicate this by a special blank symbol, denoted by $\square$. The computation is done on the work tapes. Initially, a designated input tape contains the input string, and all other cells contain the blank. If a computation halts, its result is the string contained in the designated output tape.[1] To each tape belongs a head that accesses exactly one cell of this tape. In each step of the computation, the head can change the symbol currently read and then moves to the left or to the right or does not move at all. At the same time, the current state of the machine, which is stored in its "finite control" can change. Figure 8.1 displays a Turing machine with one input and two work tapes.

Definition 8.1 (Syntax of Turing machines). *A* ***nondeterministic Turing machine with k tapes*** *(k-tape NTM, for short) is a 7-tuple $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$, where Σ is the input alphabet, Γ is the work alphabet, Z is a finite* set of states *disjoint with Γ, $\delta : Z \times \Gamma^k \to \mathfrak{P}(Z \times \Gamma^k \times \{L, R, N\}^k)$ is the* transition function*, $z_0 \in Z$ is the* initial state*, $\square \in \Gamma - \Sigma$ is the* blank symbol*, and $F \subseteq Z$ is the* set of final states*. Here, $\mathfrak{P}(S)$ denotes the* power set *of set S, i.e., the set of all subsets of S.*

For readability, we write $(z, a) \mapsto (z', b, x)$ instead of $(z', b, x) \in \delta(z, a)$ with $z, z' \in Z$, $x \in \{L, R, N\}$ and $a, b \in \Gamma$. This transition has the following meaning. If the current state is z and the head currently reads a symbol a, then:

- *a is replaced by b,*
- *z' is the new state, and*
- *the head moves according to $x \in \{L, R, N\}$, i.e., the head either moves one cell to the left (if $x = L$), or one cell to the right (if $x = R$), or it does not move at all (if $x = N$).*

The special case of a deterministic Turing machine with k tapes *(k-tape DTM, for short) is obtained by requiring that the transition function δ maps from $Z \times \Gamma^k$ to $Z \times \Gamma^k \times \{L, R, N\}^k$.*

For $k = 1$, we obtain the one-tape Turing machine, abbreviated simply by NTM and DTM, respectively. Every k-tape NTM or k-tape DTM can be simulated by

[1] One can require, for example, that the input tape is a read-only and the output tape is a write-only tape. Similarly, one can specify a variety of further variations of the technical details, but we do not pursue this any further here.

a Turing machine with only one tape, where the runtime at most doubles. Turing machines can be considered both as acceptors (which accept languages) and as transducers (which compute functions).

Definition 8.2 (Semantics of Turing machines). *Let $M = (\Sigma, \Gamma, Z, \delta, z_0, \square, F)$ be an NTM. A* configuration of M *is a string $k \in \Gamma^* Z \Gamma^*$, where $k = \alpha z \beta$ is interpreted as follows: $\alpha\beta$ is the current tape inscription, the head reads the first symbol of β, and z is the current state of M.*

On the set $\mathfrak{K}_M = \Gamma^* Z \Gamma^*$ of all configurations of M, define a binary relation $\vdash_M$, which describes the transition from a configuration $k \in \mathfrak{K}_M$ into a configuration $k' \in \mathfrak{K}_M$ according to δ. For any two strings $\alpha = a_1 a_2 \cdots a_m$ and $\beta = b_1 b_2 \cdots b_n$ in Γ^*, where $m \geq 0$ and $n \geq 1$, and for all $z \in Z$, define

$$\alpha z \beta \vdash_M \begin{cases} a_1 a_2 \cdots a_m z' c b_2 \cdots b_n & \text{if } (z, b_1) \mapsto (z', c, N) \text{ and } m \geq 0 \text{ and } n \geq 1 \\ a_1 a_2 \cdots a_m c z' b_2 \cdots b_n & \text{if } (z, b_1) \mapsto (z', c, R) \text{ and } m \geq 0 \text{ and } n \geq 2 \\ a_1 a_2 \cdots a_{m-1} z' a_m c b_2 \cdots b_n & \text{if } (z, b_1) \mapsto (z', c, L) \text{ and } m \geq 1 \text{ and } n \geq 1 \end{cases} .$$

Two special cases need be considered separately:

1. *If $n = 1$ and $(z, b_1) \mapsto (z', c, R)$ (i.e., M's head moves to the right and reads a $\square$ symbol), then $a_1 a_2 \cdots a_m z b_1 \vdash_M a_1 a_2 \cdots a_m c z' \square$.*
2. *If $m = 0$ and $(z, b_1) \mapsto (z', c, L)$ (i.e., M's head moves to the left and reads a $\square$ symbol), then $z b_1 b_2 \cdots b_n \vdash_M z' \square c b_2 \cdots b_n$.*

The initial configuration of M on input x *is always $z_0 x$. The* final configurations of M on input x *have the form $\alpha z \beta$ with $z \in F$ and $\alpha, \beta \in \Gamma^*$.*

Let $\vdash_M^*$ be the reflexive, transitive closure of $\vdash_M$: For $k, k' \in \mathfrak{K}_M$, we have $k \vdash_M^* k'$ if and only if there is a finite sequence $k_0, k_1, \ldots, k_t$ of configurations in $\mathfrak{K}_M$ such that

$$k = k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_t = k' ,$$

where possibly $k = k_0 = k_t = k'$. If $k_0 = z_0 x$ is the initial configuration of M on input x, then this sequence of configurations is a finite computation of $M(x)$, *and we say M* halts on input x. *The* language accepted by M *is defined by*

$$L(M) = \{x \in \Sigma^* \mid z_0 x \vdash_M^* \alpha z \beta \text{ with } z \in F \text{ and } \alpha, \beta \in \Gamma^*\} .$$

For NTMs, any configuration may be followed by more than one configuration. Thus, they have a *computation tree*, whose root is labelled by the initial configuration and whose leaves are labelled by the final configurations. Note that trees are special graphs (recall Definition 7.7 in Section 7.1.3 and Problem 7-2), so they have vertices and edges. The vertices of a computation tree $M(x)$ are the configurations of M on input x. For any two configurations k and k' from $\mathfrak{K}_M$, there is exactly one directed edge from k to k' if and only if $k \vdash_M k'$. A path in the computation tree of $M(x)$ is a sequence of configurations $k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_t \vdash_M \cdots$. The computation tree of an NTM can have infinite paths on which never a halting configuration is reached. For DTMs, each non-halting configuration has a unique successor configuration. Thus, the computation tree of a DTM degenerates to a linear chain.

$(s_0, a) \mapsto (s_1, \$, R)$	$(s_2, \$) \mapsto (s_2, \$, R)$	$(s_5, c) \mapsto (s_5, c, L)$
$(s_1, a) \mapsto (s_1, a, R)$	$(s_3, c) \mapsto (s_3, c, R)$	$(s_5, \$) \mapsto (s_5, \$, L)$
$(s_1, b) \mapsto (s_2, \$, R)$	$(s_3, \square) \mapsto (s_4, \square, L)$	$(s_5, b) \mapsto (s_5, b, L)$
$(s_1\$) \mapsto (s_1, \$, R)$	$(s_4, \$) \mapsto (s_4, \$, L)$	$(s_5, a) \mapsto (s_5, a, L)$
$(s_2, b) \mapsto (s_2, b, R)$	$(s_4, \square) \mapsto (s_6, \square, R)$	$(s_5, \square) \mapsto (s_0, \square, R)$
$(s_2, c) \mapsto (s_3, \$, R)$	$(s_4, c) \mapsto (s_5, c, L)$	$(s_0, \$) \mapsto (s_0, \$, R)$

Figure 8.2. Transition function δ of M for $L = \{a^n b^n c^n \mid n \geq 1\}$.

Z	Meaning	Intention
s_0	initial state	start next cycle
s_1	one a stored	look for next b
s_2	one a and one b stored	look for next c
s_3	one a, one b, and one c deleted	look for right boundary
s_4	right boundary reached	move back and test if all a, b, and c are deleted
s_5	test not successful	move back and start next cycle
s_6	test successful	accept

Figure 8.3. M's states, their meaning and their intention.

Example 8.1 Consider the language $L = \{a^n b^n c^n \mid n \geq 1\}$. A Turing machine accepting L is given by

$$M = (\{a, b, c\}, \{a, b, c, \$, \square\}, \{s_0, s_1, \ldots, s_6\}, \delta, s_0, \square, \{s_6\}) \ ,$$

where the transitions of δ are stated in Figure 8.2. Figure 8.3 provides the meaning of the single states of M and the intention corresponding to the each state. See also Exercise 8.1-2.

In order to classify problems according to their computational complexity, we need to define complexity classes. Each such class is defined by a given resource function and contains all problems that can be solved by a Turing machine that requires no more of a resource (e.g., computation time or memory space) than is specified by the resource function. We consider only the resource time here, i.e., the number of steps—as a function of the input size—needed to solve (or to accept) the problem. Further, we consider only the traditional *worst-case* complexity model. That is, among all inputs of size n, those that require the maximum resource are decisive; one thus assumes the worst case to occur. We now define deterministic and nondeterministic time complexity classes.

Definition 8.3 (Deterministic and nondeterministic time complexity).

- *Let M be a DTM with $L(M) \subseteq \Sigma^*$ and let $x \in \Sigma^*$ be an input. Define the* time function *of $M(x)$, which maps from Σ^* to $\mathbf{N}$, as follows:*

$$\mathrm{Time}_M(x) \quad = \quad \begin{cases} k & \text{if } M(x) \text{ has exactly } k+1 \text{ configurations} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Define the function $\mathrm{time}_M : \mathbf{N} \to \mathbf{N}$ *by:*

$$\mathrm{time}_M(n) \quad = \quad \begin{cases} \max_{x:|x|=n} \mathrm{Time}_M(x) & \textit{if } \mathrm{Time}_M(x) \textit{ is defined} \\ & \quad \textit{for all } x \textit{ with } |x| = n \\ \textit{undefined} & \textit{otherwise} \quad . \end{cases}$$

- *Let M be an NTM with $L(M) \subseteq \Sigma^*$ and let $x \in \Sigma^*$ be an input. Define the* time function of $M(x)$*, which maps from Σ^* to $\mathbf{N}$, as follows:*

$$\mathrm{NTime}_M(x) \quad = \quad \begin{cases} \min\{\mathrm{Time}_M(x,\alpha) \mid M(x) \textit{ accepts on path } \alpha\} & \textit{if } x \in L(M) \\ \textit{undefined} & \textit{otherwise} \; . \end{cases}$$

Define the function $\mathrm{ntime}_M : \mathbf{N} \to \mathbf{N}$ *by*

$$\mathrm{ntime}_M(n) \quad = \quad \begin{cases} \max_{x:|x|=n} \mathrm{NTime}_M(x) & \textit{if } \mathrm{NTime}_M(x) \textit{ is defined} \\ & \quad \textit{for all } x \textit{ with } |x| = n \\ \textit{undefined} & \textit{otherwise} \; . \end{cases}$$

- *Let t be a computable function that maps from $\mathbf{N}$ to $\mathbf{N}$. Define the* deterministic and nondeterministic complexity classes with time function t *by*

$$\mathrm{DTIME}(t) \quad = \quad \left\{ A \;\middle|\; \begin{array}{l} A = L(M) \textit{ for some DTM } M \textit{ and} \\ \textit{for all } n \in \mathbf{N},\ \mathrm{time}_M(n) \leq t(n) \end{array} \right\} \; ;$$

$$\mathrm{NTIME}(t) \quad = \quad \left\{ A \;\middle|\; \begin{array}{l} A = L(M) \textit{ for an NTM } M \textit{ and} \\ \textit{for all } n \in \mathbf{N} \textit{ is } \mathrm{ntime}_M(n) \leq t(n) \end{array} \right\} \; .$$

- *Let* $\mathbb{P}\mathrm{ol}$ *be the set of all polynomials. Define the complexity classes* P *and* NP *as follows:*

$$\mathrm{P} = \bigcup_{t \in \mathbb{P}\mathrm{ol}} \mathrm{DTIME}(t) \qquad \textit{and} \qquad \mathrm{NP} = \bigcup_{t \in \mathbb{P}\mathrm{ol}} \mathrm{NTIME}(t) \; .$$

Why are the classes P and NP so important? Obviously, exponential-time algorithms cannot be considered efficient in general. Garey and Johnson compare the rates of growth of some particular polynomial and exponential time functions $t(n)$ for certain input sizes relevant in practice, see Figure 8.4. They assume that a computer executes one million operations per second. Then all algorithms bounded by a polynomial run in a "reasonable" time for inputs of size up to $n = 60$, whereas for example an algorithm with time bound $t(n) = 3^n$ takes more than 6 years already for the modest input size of $n = 30$. For $n = 40$ it takes almost 4000 centuries, and for $n = 50$ a truly astronomic amount of time.

The last decades have seen an impressive development of computer and hardware technology. Figure 8.5 (taken from [75]) shows that this is not enough to provide an essentially better runtime behaviour for exponential-time algorithms, even assuming that the previous trend in hardware development continues. What would happen if one had a computer that is 100 times or even 1000 times as fast as current computers are? For functions $t_i(n)$, $1 \leq i \leq 6$, let N_i be the maximum size of inputs that can

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

Figure 8.4. Comparison of some polynomial and exponential time functions.

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

Figure 8.5. What happens when the computers get faster?

be solved by a $t_i(n)$ time-bounded algorithm within one hour. Figure 8.5 also taken from [75]) shows that a computer 1000 times faster than today's computers increases N_5 for $t_5(n) = 2^n$ by only an additive value close to 10. In contrast, using computers with the same increase in speed, an n^5 time-bounded algorithm can handle problem instances about four times as large as before.

Intuitively, the complexity class P contains the efficiently solvable problems. The complexity class NP contains many problems important in practice but currently not efficiently solvable. Examples are the satisfiability problem and the graph isomorphism problem that will be dealt with in more detail later in this chapter. The question of whether or not the classes P and NP are equal is still open. This famous P-versus-NP question gave rise to the theory of NP-completeness, which is briefly introduced in Section 8.2.

Exercises

8.1-1 Can Church's thesis ever be proven formally?

8.1-2 Consider the Turing machine M in Example 8.1.

a. What are the sequences of configurations of M for inputs $x = a^3b^3c^2$ and $y = a^3b^3c^3$, respectively?

b. Prove that M is correct, i.e., show that $L(M) = \{a^n b^n c^n \mid n \geq 1\}$.

c. Estimate the running time of M.

d. Show that the graph isomorphism problem and the graph automorphism problem introduced in Definition 7.7 are both in NP.

8.2. NP-completeness

The theory of NP-completeness provides methods to prove lower bounds for problems in NP. An NP problem is said to be complete in NP if it belongs to the hardest problems in this class, i.e., if it is at least as hard as any NP problem. The complexity of two given problems can be compared by polynomial-time reductions. Among the different types of reduction one can consider, we focus on the *polynomial-time many-one reducibility* in this section. In Section 8.4, more general reducibilities will be introduced, such as the *polynomial-time Turing reducibility* and the *polynomial-time (strong) nondeterministic Turing reducibility*.

Definition 8.4 (Reducibility, NP-Completeness). *A set A is* reducible *to a set B (in symbols, $A \leq_{\mathrm{m}}^{\mathrm{p}} B$) if and only if there exists a polynomial-time computable function r such that for all $x \in \Sigma^*$, $x \in A \iff r(x) \in B$. A set B is said to be* $\leq_{\mathrm{m}}^{\mathrm{p}}$-hard for NP *if and only if $A \leq_{\mathrm{m}}^{\mathrm{p}} B$ for each set $A \in \mathrm{NP}$. A set B is said to be* $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete in NP *(*NP-complete*, for short) if and only if B is $\leq_{\mathrm{m}}^{\mathrm{p}}$-hard for* NP *and $B \in \mathrm{NP}$.*

Reductions are efficient algorithms that can be used to show that problems are not efficiently solvable. That is, if one can efficiently transform a hard problem into another problem via a reduction, the hardness of the former problem is inherited by the latter problem. At first glance, it might seem that infinitely many efficient algorithms are required to prove some problem X NP-hard, namely one reduction from each of the infinitely many NP problems to X. However, an elementary result says that it is enough to find just one such reduction, from *some* NP-complete problem V. Since the $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducibility is transitive (see Exercise 8.2-2), the NP-hardness of V implies the NP-hardness of X via the reduction $A \leq_{\mathrm{m}}^{\mathrm{p}} V \leq_{\mathrm{m}}^{\mathrm{p}} X$ for each NP problem A.

In 1971, Stephen Cook found a first such NP-complete problem: the satisfiability problem of propositional logic, `SAT` for short. For many NP-completeness result, it is useful to start from the special problem `3-SAT`, the restriction of the satisfiability problem in which each given boolean formula is in conjunctive normal form and each clause contains exactly three literals. `3-SAT` is also NP-complete.

Definition 8.5 (Satisfiability problem). *The boolean constants* false *and* true *are represented by* 0 *and* 1*. Let $x_1, x_2, \ldots, x_m$ be boolean variables, i.e., $x_i \in \{0,1\}$ for each i. Variables and their negations are called* literals*. A boolean formula φ is* satisfiable *if and only if there is an assignment to the variables in φ that makes the formula true. A boolean formula φ is in* conjunctive normal form *(*CNF*, for short) if and only if φ is of the form $\varphi(x_1, x_2, \ldots, x_m) = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{k_i} \ell_{i,j} \right)$, where the $\ell_{i,j}$ are literals over $\{x_1, x_2, \ldots, x_m\}$. The disjunctions $\bigvee_{j=1}^{k_i} \ell_{i,j}$ of literals are called the* clauses *of φ. A boolean formula φ is in k-*CNF *if and only if φ is in CNF and each clause of φ contains exactly k literals. Define the following two problems:*

$$\begin{aligned} \texttt{SAT} &= \{\varphi \mid \varphi \text{ is a satisfiable boolean formula in CNF}\}\,; \\ \texttt{3-SAT} &= \{\varphi \mid \varphi \text{ is a satisfiable boolean formula in 3-CNF}\}\,. \end{aligned}$$

Example 8.2 [Boolean formulas] Consider the following two satisfiable boolean formulas (see also Exercise 8.2-1):

$$\begin{aligned}\varphi(w,x,y,z) &= (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (w \vee \neg y \vee z) \wedge (\neg w \vee \neg x \vee z);\\ \psi(w,x,y,z) &= (\neg w \vee x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z) \wedge (\neg w \vee y \vee z) \wedge (w \vee \neg x \vee \neg z)\ .\end{aligned}$$

Here, φ is in 3-CNF, so φ is in 3-SAT. However, ψ is not in 3-CNF, since the first clause contains four literals. Thus, ψ is in SAT but not in 3-SAT.

Theorem 8.6 states the above-mentioned result of Cook.

Theorem 8.6 (Cook). *The problems* SAT *and* 3-SAT *are* NP*-complete.*

The proof that SAT is NP-complete is omitted. The idea is to encode the computation of an arbitrary NP machine M running on input x into a boolean formula $\varphi_{M,x}$ such that $\varphi_{M,x}$ is satisfiable if and only if M accepts x.

SAT is a good starting point for many other NP-completeness results. In fact, in many cases it is very useful to start with its restriction 3-SAT. To give an idea of how such proofs work, we now show that SAT $\leq_{\mathrm{m}}^{\mathrm{p}}$ 3-SAT, which implies that 3-SAT is NP-complete. To this end, we need to find a reduction r that transforms any given boolean formula φ in CNF into another boolean formula $\widehat{\varphi}$ in 3-CNF (i.e., with exactly three literals per clause) such that

$$\varphi \text{ is satisfiable} \iff \widehat{\varphi} \text{ is satisfiable}\ . \tag{8.1}$$

Let $\varphi(x_1, x_2, \ldots, x_n)$ be the given formula with clauses $C_1, C_2, \ldots, C_m$. Construct the formula $\widehat{\varphi}$ from φ as follows. The variables of $\widehat{\varphi}$ are

- φ's variables $x_1, x_2, \ldots, x_n$ and
- for each clause C_j of φ, a number of additional variables $y_1^j, y_2^j, \ldots, y_{k_j}^j$ as needed, where k_j depends on the structure of C_j according to the case distinction below.

Now, define $\widehat{\varphi} = \widehat{C}_1 \wedge \widehat{C}_2 \wedge \cdots \wedge \widehat{C}_m$, where each clause $\widehat{C}_j$ of $\widehat{\varphi}$ is constructed from the clause C_j of φ as follows. Suppose that $C_j = (z_1 \vee z_2 \vee \cdots \vee z_k)$, where each z_i is a literal over $\{x_1, x_2, \ldots, x_n\}$. Distinguish the following four cases.

- If $k = 1$, define
$$\widehat{C}_j = (z_1 \vee y_1^j \vee y_2^j) \wedge (z_1 \vee \neg y_1^j \vee y_2^j) \wedge (z_1 \vee y_1^j \vee \neg y_2^j) \wedge (z_1 \vee \neg y_1^j \vee \neg y_2^j)\ .$$
- If $k = 2$, define $\widehat{C}_j = (z_1 \vee z_2 \vee y_1^j) \wedge (z_1 \vee z_2 \vee \neg y_1^j)$.
- If $k = 3$, define $\widehat{C}_j = C_j = (z_1 \vee z_2 \vee z_3)$, i.e., the jth clause remains unchanged.
- If $k \geq 4$, define
$$\begin{aligned}\widehat{C}_j &= (z_1 \vee z_2 \vee y_1^j) \wedge (\neg y_1^j \vee z_3 \vee y_2^j) \wedge (\neg y_2^j \vee z_4 \vee y_3^j) \wedge \cdots \wedge\\ &\quad (\neg y_{k-4}^j \vee z_{k-2} \vee y_{k-3}^j) \wedge (\neg y_{k-3}^j \vee z_{k-1} \vee z_k)\ .\end{aligned}$$

It remains to show that (a) the reduction r is polynomial-time computable, and (b) the equivalence (8.1) is true. Both claims are easy to see; the details are left to

the reader as Exercise 8.2-4.

Thousands of problems have been proven NP-complete by now. A comprehensive collection can be found in the work of Garey und Johnson [75].

Exercises

8.2-1 Find a satisfying assignment each for the boolean formulas φ and ψ from Example 8.2.

8.2-2 Show that the $\leq_{\mathrm{m}}^{\mathrm{p}}$-reducibility is transitive: $(A \leq_{\mathrm{m}}^{\mathrm{p}} B \wedge B \leq_{\mathrm{m}}^{\mathrm{p}} C) \Longrightarrow A \leq_{\mathrm{m}}^{\mathrm{p}} C$.

8.2-3 Prove that SAT is in NP.

8.2-4 Consider the reduction SAT $\leq_{\mathrm{m}}^{\mathrm{p}}$ 3-SAT. Prove the following:

a. the reduction r is polynomial-time computable, and

b. the equivalence (8.1) holds.

8.3. Algorithms for the satisfiability problem

By Theorem 8.6, SAT and 3-SAT are both NP-complete. Thus, if SAT were in P, it would immediately follow that P = NP, which is considered unlikely. Thus, it is very unlikely that there is a polynomial-time deterministic algorithm for SAT or 3-SAT. But what is the runtime of the best deterministic algorithms for them? And what about randomised algorithms? Let us focus on the problem 3-SAT in this section.

8.3.1. A deterministic algorithm

The "naive" deterministic algorithm for 3-SAT works as follows: Given a boolean formula φ with n variables, sequentially check the 2^n possible assignments to the variables of φ. Accept if the first satisfying assignment is found, otherwise reject. Obviously, this algorithm runs in time $O(2^n)$. Can this upper bound be improved?

Yes, it can. We will present an slightly better deterministic algorithm for 3-SAT that still runs in exponential time, namely in time $\tilde{O}(1.9129^n)$, where the $\tilde{\mathcal{O}}$ notation neglects polynomial factors as is common for exponential-time algorithms.[2] The point of such an improvement is that a $\tilde{\mathcal{O}}(c^n)$ algorithm, where $1 < c < 2$ is a constant, can deal with larger instances than the naive $\tilde{\mathcal{O}}(2^n)$ algorithm in the same amount of time before the exponential growth rate eventually hits and the running time becomes infeasible. For example, if $c = \sqrt{2} \approx 1.414$ then $\tilde{O}\left(\sqrt{2}^{2n}\right) = \tilde{O}(2^n)$. Thus, this algorithm can deal with inputs twice as large as the naive algorithm in the same amount of time. Doubling the size of inputs that can be handled by some algorithm can be quite important in practice.

The deterministic algorithm for 3-SAT is based on a simple "*backtracking*" idea. Backtracking is useful for problems whose solutions consist of n components each having more than one choice possibility. For example, a solution of 3-SAT is composed of the n truth values of a satisfying assignment, and each such truth value can be either true (represented by 1) or false (represented by 0).

[2] The result presented here is not the best result known, but see Figure 8.7 on page 392 for further improvements.

The idea is the following. Starting from the initially empty solution (i.e., the empty truth assignment), we seek to construct by recursive calls to our backtracking algorithm, step by step, a larger and larger partial solution until eventually a complete solution is found, if one exists. In the resulting recursion tree,[3] the root is marked by the empty solution, and the leaves are marked with complete solutions of the problem. If the current branch in the recursion tree is "dead" (which means that the subtree underneath it cannot lead to a correct solution), one can prune this subtree and "backtracks" to pursue another extention of the partial solution constructed so far. This pruning may save time in the end.

BACKTRACKING-SAT(φ, β)

```
1  if (β assigns truth values to all variables of φ)
2    then return φ(β)
3    else  if (β makes one of the clauses of φ false)
4              then return 0
5    ' ▷ "dead branch"
6                    else if BACKTRACKING-SAT(φ, β0))
7                         then return 1
8                         else return BACKTRACKING-SAT(φ, β1))
```

The input of algorithm BACKTRACKING-SAT are a boolean formula φ and a partial assignment β to some of the variables of φ. This algorithm returns a boolean value: 1, if the partial assignment β can be extended to a satisfying assignment to all variables of φ, and 0 otherwise. Partial assignments are here considered to be strings of length at most n over the alphabet $\{0, 1\}$.

The first call of the algorithm is BACKTRACKING-SAT(φ, λ), where λ denotes the empty assignment. If it turns out that the partial assignment constructed so far makes one of the clauses of φ false, it cannot be extended to a satisfying assignment of φ. Thus, the subtree underneath the corresponding vertex in the recursion tree can be pruned; see also Exercise 8.3-1.

To estimate the runtime of BACKTRACKING-SAT, note that this algorithm can be specified so as to select the variables in an "intelligent" order that minimises the number of steps needed to evaluate the variables in any clause. Consider an arbitrary, fixed clause C_j of the given formula φ. Each satisfying assignment β of φ assigns truth values to the three variables in C_j. There are $2^3 = 8$ possibilities to assign a truth value to these variables, and one of them can be excluded certainly: the assignment that makes C_j false. The corresponding vertex in the recursion tree of BACKTRACKING-SAT(φ, β) thus leads to a "dead" branch, so we prune the subtree underneath it.

Depending on the structure of φ, there may exist further "dead" branches whose subtrees can also be pruned. However, since we are trying to find an upper bound in the worst case, we do not consider these additional "dead" subtrees. It follows that

$$\tilde{O}\left(\left(2^3 - 1\right)^{\frac{n}{3}}\right) = \tilde{O}(\sqrt[3]{7}^{\,n}) \approx \tilde{O}(1.9129^n)$$

[3] The inner vertices of the recursion tree represent the recursive calls of the algorithm, its root is the first call, and the algorithm terminates at the leaves without any further recursive call.

is an upper bound for BACKTRACKING-SAT in the worst case. This bound slightly improves upon the trivial $\tilde{O}(2^n)$ upper bound of the "naive" algorithm for 3-SAT.

As mentioned above, the deterministic time complexity of 3-SAT can be improved even further. For example, Monien and Speckenmeyer [172] proposed a divide-and-conquer algorithm with runtime $\tilde{O}(1.618^n)$. Dantsin et al. [56] designed a deterministic "local search with restart" algorithm whose runtime is $\tilde{O}(1.481^n)$, which was further improved by Brueggemann and Kern [32] in 2004 to a $\tilde{O}(1.4726^n)$ bound.

There are also randomised algorithms that have an even better runtime. One will be presented now, a "random-walk" algorithm that is due to Schöning [214].

8.3.2. A randomised algorithm

A *random walk* can be done on a specific structure, such as in the Euclidean space, on an infinite grid or on a given graph. Here we are interested in random walks occurring on a graph that represents a certain stochastic automaton. To describe such automata, we first introduce the notion of a finite automaton.

A *finite automaton* can be represented by its transition graph, whose vertices are the states of the finite automaton, and the transitions between states are directed edges marked by the symbols of the alphabet Σ. One designated vertex is the *initial state* in which the computation of the automaton starts. In each step of the computation, the automaton reads one input symbol (proceeding from left to right) and moves to the next state along the edge marked by the symbol read. Some vertices represent *final states*. If such a vertex is reached after the entire input has been read, the automaton accepts the input, and otherwise it rejects. In this way, a finite automaton accepts a set of input strings, which is called its language.

A *stochastic automaton* $\mathcal{S}$ is a finite automaton whose edges are marked by probabilities in addition. If the edge from u to v in the transition graph of $\mathcal{S}$ is marked by $p_{u,v}$, where $0 \leq p_{u,v} \leq 1$, then $\mathcal{S}$ moves from state u to state v with probability $p_{u,v}$. The process of random transitions of a stochastic automaton is called a *Markov chain* in probability theory. Of course, the acceptance of strings by a stochastic automaton depends on the transition probabilities.

RANDOM-SAT(φ)

1 **for** $i \leftarrow 1$ **to** $\lceil (4/3)^n \rceil$)
  ▷ n is the number of variables in φ
2 **do** randomly choose an assignment $\beta \in \{0,1\}^n$
  under the uniform distribution

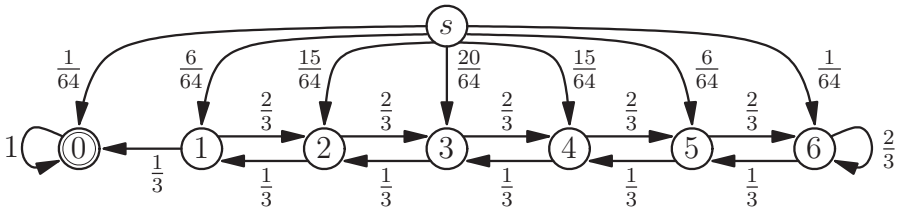


Figure 8.6. Transition graph of a stochastic automaton for describing RANDOM-SAT.

```
3     for j ← 1 to n
4         if (φ(β) = 1)
5             then return the satisfying assignment β to φ
6             else  choose a clause C = (x ∨ y ∨ z) with C(β) = 0
7                   randomly choose a literal ℓ ∈ {x, y, z}
                    under the uniform distribution
8                   determine the bit β_ℓ ∈ {0, 1} in β assigning ℓ
9                   swap β_ℓ to 1 − β_ℓ in β
10  return "φ is not satisfiable"
```

Here, we are not interested in recognising languages by stochastic automata, but rather we will use them to describe a random walk by the randomised algorithm RANDOM-SAT. Given a boolean formula φ with n variables, RANDOM-SAT tries to find a satisfying assignment to φ's variables, if one exists.

On input φ, RANDOM-SAT starts by guessing a random initial assignment β, where each bit of β takes on the value 0 and 1 with probability 1/2. Suppose φ is satisfiable. Let $\tilde{\beta}$ be an arbitrary fixed assignment of φ. Let X be a random variable that expresses the *Hamming distance between β and $\tilde{\beta}$*, i.e., X gives the number of bits in which β and $\tilde{\beta}$ differ. Clearly, X can take on values $j \in \{0, 1, \ldots, n\}$ and is distributed according to the binomial distribution with parameters n and 1/2. That is, the probability for $X = j$ is exactly $\binom{n}{j} 2^{-n}$.

RANDOM-SAT now checks whether the initial assignment β already satisfies φ, and if so, it accepts. Otherwise, if β does not satisfy φ, there must exist a clause in φ not satisfied by β. RANDOM-SAT now picks any such clause, randomly chooses under the uniform distribution some literal in this clause, and "flips" the corresponding bit in the current assignment β. This procedure is repeated n times. If the current assignment β still does not satisfy φ, RANDOM-SAT restarts with a new initial assignment, and repeats this entire procedure t times, where $t = \lceil (4/3)^n \rceil$.

Figure 8.6 shows a stochastic automaton $\mathcal{S}$, whose edges are not marked by symbols but only by transition probabilities. The computation of RANDOM-SAT on input φ can be viewed as a random walk on $\mathcal{S}$ as follows. Starting from the initial state s, which will never be reached again later, RANDOM-SAT(φ) first moves to one of the states $j \in \{0, 1, \ldots, n\}$ according to the binomial distribution with parameters n and 1/2. This is shown in the upper part of Figure 8.6 for a formula with $n = 6$ variables. Reaching such a state j means that the randomly chosen initial assignment β and the fixed satisfying assignment $\tilde{\beta}$ have Hamming distance j. As long as $j \neq 0$,

RANDOM-SAT(φ) changes one bit β_ℓ to $1-\beta_\ell$ in the current assignment β, searching for a satisfying assignment in each iteration of the inner `for` loop. In the random walk on $\mathcal{S}$, this corresponds to moving one step to the left to state $j-1$ or moving one step to the right to state $j+1$, where only states less than or equal to n can be reached.

The fixed assignment $\tilde{\beta}$ satisfies φ, so it sets at least one literal in each clause of φ to true. If we fix *exactly* one of the literals satisfied by $\tilde{\beta}$ in each clause, say ℓ, then RANDOM-SAT(φ) makes a step to the left if and only if ℓ was chosen by RANDOM-SAT(φ). Hence, the probability of moving from state $j > 0$ to state $j-1$ is $1/3$, and the probability of moving from state $j > 0$ to state $j+1$ is $2/3$.

If the state $j = 0$ is reached eventually after at most n iterations of this process, β and $\tilde{\beta}$ have Hamming distance 0, so β satisfies φ and RANDOM-SAT(φ) returns β and halts accepting. Of course, one might also hit a satisfying assignment (distinct from $\tilde{\beta}$) in some state $j \neq 0$. But since this would only increase the acceptance probability, we neglect this possibility here.

If this process is repeated n times unsuccessfully, then the initial assignment β was chosen so badly that RANDOM-SAT now dumps it, and restarts the above process from scratch with a new initial assignment. The entire procedure is repeated at most t times, where $t = \lceil (4/3)^n \rceil$. If it is still unsuccessful after t trials, RANDOM-SAT rejects its input.

Since the probability of moving away from state 0 to the right is larger than the probability of moving toward 0 to the left, one might be tempted to think that the success probability of RANDOM-SAT is very small. However, one should not underestimate the chance that one already after the initial step from s reaches a state close to 0. The closer to 0 the random walk starts, the higher is the probability of reaching 0 by random steps to the left or to the right.

We only give a rough sketch of estimating the success probability (assuming that φ is satisfiable) and the runtime of RANDOM-SAT(φ). For convenience, suppose that 3 divides n. Let p_i be the probability for the event that RANDOM-SAT(φ) reaches the state 0 within n steps after the initial step, under the condition that it reaches some state $i \leq n/3$ with the initial step from s. For example, if the state $n/3$ is reached with the initial step and if no more than $n/3$ steps are done to the right, then one can still reach the final state 0 by a total of at most n steps. If one does more than $n/3$ steps to the right starting from state $n/3$, then the final state cannot be reached within n steps. In general, starting from state i after the initial step, no more than $(n-i)/2$ steps to the right may be done. As noted above, a step to the right is done with probability $2/3$, and a step to the left is done with probability $1/3$. It follows that

$$p_i \quad = \quad \binom{n}{\frac{n-i}{2}} \left(\frac{2}{3}\right)^{\frac{n-i}{2}} \left(\frac{1}{3}\right)^{n-\frac{n-i}{2}} . \tag{8.2}$$

Now, let q_i be the probability for the event that RANDOM-SAT(φ) reaches some state $i \leq n/3$ with the initial step. Clearly, we have

$$q_i \quad = \quad \binom{n}{i} \cdot 2^{-n} . \tag{8.3}$$

Finally, let p be the probability for the event that RANDOM-SAT(φ) reaches the final state 0 within the inner `for` loop. Of course, this event can occur also when starting from a state $j > n/3$. Thus,

$$p \quad \geq \quad \sum_{i=0}^{n/3} p_i \cdot q_i \ .$$

Approximating this sum by the entropy function and estimating the binomial coefficients from (8.2) and (8.3) in the single terms by Stirling's formula, we obtain the lower bound $\Omega((3/4)^n)$ for p.

To reduce the error probability, RANDOM-SAT performs a total of t independent trials, each starting with a new initial assignment β. For each trial, the probability of success (i.e., the probability of finding a satisfying assignment of φ, if one exists) is at least $(3/4)^n$, so the error is bounded by $1-(3/4)^n$. Since the trials are independent, these error probabilities multiply, which gives an error of $(1-(3/4)^n)^t \leq \mathrm{e}^{-1}$. Thus, the total probabilitiy of success of RANDOM-SAT(φ) is at least $1-1/\mathrm{e} \approx 0.632$ if φ is satisfiable. On the other hand, RANDOM-SAT(φ) does not make any error at all if φ is unsatisfiable; in this case, the output is : "φ is not satisfiable".

The particular choice of this value of t can be explained as follows. The runtime of a randomised algorithm, which performs independent trials such as RANDOM-SAT(φ), is roughly reciprocal to the success probability of one trial, $p \approx (3/4)^n$. In particular, the error probability (i.e., the probability that that in none of the t trials a satisfying assignment of φ is found even though φ is satisfiable) can be estimated by $(1-p)^t \leq e^{-t \cdot p}$. If a fixed error of ε is to be not exceeded, it is enough to choose t such that $e^{-t \cdot p} \leq \varepsilon$; equivalently, such that $t \geq \ln(1/\varepsilon)/p$. Up to constant factors, this can be accomplished by choosing $t = \lceil (4/3)^n \rceil$. Hence, the runtime of the algorithm is in $\tilde{O}\left((4/3)^n\right)$.

Exercises

8.3-1 Start the algorithm BACKTRACKING-SAT for the boolean formula $\varphi = (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg u \vee y \vee z) \wedge (u \vee \neg y \vee z)$ and construct step by step a satisfying assignment of φ. How does the resulting recursion tree look like?

8.4. Graph isomorphism and lowness

In this section, we need some of the group-theoretic and graph-theoretic foundations presented in Section 7.1.3. In particular, recall the notion of permutation group from Definition 7.6 and the graph isomorphism problem (`GI`, for short) and the graph automorphism problem (`GA`, for short) from Definition 7.7; see also Example 7.4 in Chapter 7. We start by providing some more background from complexity theory.

8.4.1. Reducibilities and complexity hierarchies

In Section 8.2, we have seen that the problems `SAT` and `3-SAT` are NP-complete. Clearly, P = NP if and only if *every* NP problem (including the NP-complete problems)

is in P, which in turn is true if and only if *some* NP-complete problem is in P. So, no NP-complete problem can be in P if P $\neq$ NP. An interesting question is whether, under the plausible assumption that P $\neq$ NP, every NP problem is either in P or NP-complete. Or, assuming P $\neq$ NP, can there exist NP problems that are *neither* in P *nor* NP-complete? A result by Ladner [145] answers this question.

Theorem 8.7 (Ladner). *If* P $\neq$ NP *then there exist sets in* NP *that are neither in* P *nor* NP*-complete.*

The problems constructed in the proof of Theorem 8.7 are not overly natural problems. However, there are also good candidates of "natural" problems that are neither in P nor NP-complete. One such candidate is the graph isomorphism problem. To provide some evidence for this claim, we now define two hierarchies of complexity classes, the *low hierarchy* and the *high hierarchy*, both introduced by Schöning [212]. First, we need to define the *polynomial hierarchy*, which builds upon NP. And to this end, we need a more flexible reducibility than the (polynomial-time) many-one reducibility $\leq_{\mathrm{m}}^{\mathrm{p}}$ from Definition 8.4, namely the *Turing reducibility* $\leq_{\mathrm{T}}^{\mathrm{p}}$. We will also define the (polynomial-time) *nondeterministic Turing reducibility*, $\leq_{\mathrm{T}}^{\mathrm{NP}}$, and the (polynomial-time) *strong nondeterministic Turing reducibility*, $\leq_{\mathrm{sT}}^{\mathrm{NP}}$. These two reducibilities are important for the polynomial hierarchy and for the high hierarchy, respectively. Turing reducibilities are based on the notion of oracle Turing machines, which we now define.

Definition 8.8 (Oracle Turing machine). *An* ***oracle set*** *is a set of strings. An* oracle Turing machine *M, with oracle B, is a Turing machine that has a special worktape, which is called the* oracle tape *or* query tape*. In addition to other states, M contains a special* query state*, $s_?$, and the two* answer states *$s_{\mathtt{yes}}$ and $s_{\mathtt{no}}$. During a computation on some input, if M is not in the query state $s_?$, it works just like a regular Turing machine. However, when M enters the query state $s_?$, it interrupts its computation and queries its oracle about the string q currently written on the oracle tape. Imagine the oracle B as some kind of "black box": B answers the query of whether or not it contains q within one step of M's computation, regardless of how difficult it is to decide the set B. If $q \in B$, then M changes its current state into the new state $s_{\mathtt{yes}}$ and continues its computation. Otherwise, if $q \notin B$, M continues its computation in the new state $s_{\mathtt{no}}$. We say that the computation of M on input x* is performed relative to the oracle *B, and we write $M^B(x)$.*

The language accepted by M^B is denoted $L(M^B)$. We say a language L is represented by an oracle Turing machine *M if and only if $L = L(M^{\emptyset})$. We say a class $\mathcal{C}$ of languages is* relativizable *if and only if it can be represented by oracle Turing machines relative to the empty oracle. For any relativizable class $\mathcal{C}$ and for any oracle B, define the class $\mathcal{C}$* relative to *B by*

$$\mathcal{C}^B = \{L(M^B) \mid M \text{ is an oracle Turing machine representing some set in } \mathcal{C}\} .$$

For any class $\mathcal{B}$ of oracle sets, define $\mathcal{C}^{\mathcal{B}} = \bigcup_{B \in \mathcal{B}} \mathcal{C}^B$.

Let NPOTM (respectively, DPOTM) be a shorthand for *nondeterministic* (respectively, *deterministic*) *polynomial-time oracle Turing machine*. For example, the

following classes can be defined:

$$\mathrm{NP}^{\mathrm{NP}} = \bigcup_{B\in\mathrm{NP}} \mathrm{NP}^B = \{L(M^B) \mid M \text{ is an NPOTM and } B \text{ is in NP}\} ;$$
$$\mathrm{P}^{\mathrm{NP}} = \bigcup_{B\in\mathrm{NP}} \mathrm{P}^B = \{L(M^B) \mid M \text{ is a DPOTM and } B \text{ is in NP}\} .$$

For the empty oracle set $\emptyset$, we obtain the unrelativized classes $\mathrm{NP} = \mathrm{NP}^{\emptyset}$ and $\mathrm{P} = \mathrm{P}^{\emptyset}$, and we then write NPTM instead of NPOTM and DPTM instead of DPOTM.

In particular, oracle Turing machines can be used for prefix search. Let us consider an example.

Example 8.3 [Prefix search by an oracle Turing machine]

Suppose we wish to find the smallest solution of the graph isomorphism problem, which is in NP; see Definition 7.7 in Subsection 7.1.3. Let G and H be two given graphs with $n \geq 1$ vertices each. An isomorphism between G and H is called a *solution of* "$(G, H) \in \mathtt{GI}$". The set of isomorphisms between G and H, $\mathrm{Iso}(G, H)$, contains all solutions of "$(G, H) \in \mathtt{GI}$".

Our goal is to find the lexicographically smallest solution if $(G, H) \in \mathtt{GI}$; otherwise, we output the empty string λ to indicate that $(G, H) \notin \mathtt{GI}$. That is, we wish to compute the function f defined by $f(G, H) = \min\{\pi \mid \pi \in \mathrm{Iso}(G, H)\}$ if $(G, H) \in \mathtt{GI}$, and $f(G, H) = \lambda$ if $(G, H) \notin \mathtt{GI}$, where the minimum is to be taken according to the lexicographic order on $\mathfrak{S}_n$. More precisely, we view a permutation $\pi \in \mathfrak{S}_n$ as the string $\pi(1)\pi(2)\cdots\pi(n)$ of length n over the alphabet $[n] = \{1, 2, \ldots, n\}$, and we write $\pi < \sigma$ for $\pi, \sigma \in \mathfrak{S}_n$ if and only if there is a $j \in [n]$ such that $\pi(i) = \sigma(i)$ for all $i < j$ and $\pi(j) < \sigma(j)$.

From a permutation $\sigma \in \mathfrak{S}_n$, one obtains a *partial permutation* by cancelling some pairs $(i, \sigma(i))$. A partial permutation can be represented by a string over the alphabet $[n] \cup \{*\}$, where $*$ indicates an undefined position. Let $k \leq n$. A *prefix of length k of* $\sigma \in \mathfrak{S}_n$ is a partial permutation of σ containing each pair $(i, \sigma(i))$ with $i \leq k$, but none of the pairs $(i, \sigma(i))$ with $i > k$. In particular, for $k = 0$, the empty string λ is the (unique) length 0 prefix of σ. For $k = n$, the total permutation σ is the (unique) length n prefix of itself. Suppose that π is a prefix of length $k < n$ of $\sigma \in \mathfrak{S}_n$ and that $w = i_1 i_2 \cdots i_{|w|}$ is a string over $[n]$ of length $|w| \leq n - k$ with none of the i_j occurring in π. Then, πw denotes the partial permutation that extends π by the pairs $(k + 1, i_1), (k + 2, i_2), \ldots, (k + |w|, i_{|w|})$. If in addition $\sigma(k+j) = i_j$ for $1 \leq j \leq |w|$, then πw is also a prefix of σ. For our prefix search, given two graphs G and H, we define the set of prefixes of the isomorphisms in $\mathrm{Iso}(G, H)$ by

$$\text{Pre-Iso} = (G, H, \pi) \left| \begin{array}{l} G \text{ and } H \text{ are graphs with } n \text{ vertices each and} \\ (\exists w \in [n]^*)\ [w = i_1 i_2 \cdots i_{n-|\pi|} \text{ and } \pi w \in \mathrm{Iso}(G, H)] \end{array} \right. .$$

Note that, for $n \geq 1$, the empty string λ does not encode a permutation in $\mathfrak{S}_n$. Furthermore, $\mathrm{Iso}(G, H) = \emptyset$ if and only if $(G, H, \lambda) \notin$ Pre-Iso, which in turn is true if and only if $(G, H) \notin \mathtt{GI}$.

Starting from the empty string, we will construct, bit by bit, the smallest isomorphism between the two given graphs (if there exists any). We below present an DPOTM N that, using the NP set Pre-Iso as its oracle, computes the function f by prefix search; see also Exercise 8.4-2. Denoting the class of functions computable in polynomial time by FP, we thus have shown that f is in $\mathrm{FP}^{\text{Pre-Iso}}$. Since Pre-Iso is in NP (see Exercise 8.4-2), it follows that f is in $\mathrm{FP}^{\mathrm{NP}}$.

N-PRE-ISO(G, H)

```
1  if ((G, H, λ) ∉ Pre-Iso)
2     then return λ
3     else  π := λ
4           j ← 0
5           while j < n                          ▷ G and H have n vertices each
6                 do i ← 1
7                    while (G, H, πi) ∉ Pre-Iso
8                          do i ← i + 1
9                          π ← πi
10                         j ← j + 1
11          return π
```

Example 8.3 shows that also Turing machines computing functions can be equipped with an oracle, and that also function classes such as FP can be relativizable. We now return to oracle machines accepting languages and use them to define several reducibilities. All reducibilities considered here are polynomial-time computable.

Definition 8.9 (Turing reducibilities). *Let $\Sigma = \{0,1\}$ be a binary alphabet, let A and B be sets of strings over Σ, and let $\mathcal{C}$ be any complexity class. The set of complements of sets in $\mathcal{C}$ is defined by* $\mathrm{co}\mathcal{C} = \{\overline{L} \mid L \in \mathcal{C}\}$.

Define the following reducibilities:

- ***Turing reducibility:*** $A \leq_{\mathrm{T}}^{\mathrm{p}} B \iff A = L(M^B)$ *for some DPOTM M.*
- ***Nondeterministic Turing reducibility:*** $A \leq_{\mathrm{T}}^{\mathrm{NP}} B \iff A = L(M^B)$ *for some NPOTM M.*
- ***Strong nondeterministic Turing reducibility:*** $A \leq_{\mathrm{sT}}^{\mathrm{NP}} B \iff A \in \mathrm{NP}^B \cap \mathrm{coNP}^B$.
- *Let $\leq_r$ be one of the reducibilities defined above. We call a set B* $\leq_r$-hard for $\mathcal{C}$ *if and only if $A \leq_r B$ for each set $A \in \mathcal{C}$. A set B is said to be* ***$\leq_r$-complete in*** *$\mathcal{C}$ if and only if B is $\leq_r$-hard for $\mathcal{C}$ and $B \in \mathcal{C}$.*
- $\mathrm{P}^{\mathcal{C}} = \{A \mid (\exists B \in \mathcal{C})\,[A \leq_{\mathrm{T}}^{\mathrm{p}} B]\}$ *is the* ***closure of $\mathcal{C}$ under the $\leq_{\mathrm{T}}^{\mathrm{p}}$-reducibility.***
- $\mathrm{NP}^{\mathcal{C}} = \{A \mid (\exists B \in \mathcal{C})\,[A \leq_{\mathrm{T}}^{\mathrm{NP}} B]\}$ *is the* ***closure of $\mathcal{C}$ under the $\leq_{\mathrm{T}}^{\mathrm{NP}}$-reducibility.***

Using the $\leq_{\mathrm{T}}^{\mathrm{p}}$-reducibility and the $\leq_{\mathrm{T}}^{\mathrm{NP}}$-reducibility introduced in Definition 8.9, we now define the polynomial hierarchy, and the low and the high hierarchy within NP.

Definition 8.10 (Polynomial hierarchy). *Define the* ***polynomial hierarchy*** PH *inductively as follows:* $\Delta_0^p = \Sigma_0^p = \Pi_0^p = \mathrm{P}$, $\Delta_{i+1}^p = \mathrm{P}^{\Sigma_i^p}$, $\Sigma_{i+1}^p = \mathrm{NP}^{\Sigma_i^p}$ *and* $\Pi_{i+1}^p = \mathrm{co}\Sigma_{i+1}^p$ *for $i \geq 0$, and* $\mathrm{PH} = \bigcup_{k \geq 0} \Sigma_k^p$.

In particular, $\Delta_1^p = \mathrm{P}^{\Sigma_0^p} = \mathrm{P}^{\mathrm{P}} = \mathrm{P}$ and $\Sigma_1^p = \mathrm{NP}^{\Sigma_0^p} = \mathrm{NP}^{\mathrm{P}} = \mathrm{NP}$ and $\Pi_1^p = \mathrm{co}\Sigma_1^p = \mathrm{coNP}$. The following theorem, which is stated without proof, provides some properties of the polynomial hierarchy, see Exercise *8-2*.

Theorem 8.11 (Meyer and Stockmeyer). *For alle $i \geq 1$ holds:*

1. $\Sigma_{i-1}^p \cup \Pi_{i-1}^p \subseteq \Delta_i^p \subseteq \Sigma_i^p \cap \Pi_i^p$.
2. Σ_i^p, Π_i^p, Δ_i^p, *and* PH *are closed under* $\leq_{\mathrm{m}}^{\mathrm{p}}$*-reductions.* Δ_i^p *is even closed under* $\leq_{\mathrm{T}}^{\mathrm{p}}$*-reductions.*
3. Σ_i^p *contains exactly those sets* A *for which there exist a set* B *in* P *and a polynomial* p *such that for each* $x \in \Sigma^*$*:*

$$x \in A \iff (\exists^p w_1)\,(\forall^p w_2) \cdots (\mathfrak{Q}^p w_i)\,[(x, w_1, w_2, \ldots, w_i) \in B],$$

   where the quantifiers $\exists^p$ *and* $\forall^p$ *are polynomially length-bounded, and* $\mathfrak{Q}^p = \exists^p$ *if* i *is odd, and* $\mathfrak{Q}^p = \forall^p$ *if* i *is even.*
4. *If* $\Sigma_{i-1}^p = \Sigma_i^p$ *for some* i*, then* PH *collapses to* $\Sigma_{i-1}^p = \Pi_{i-1}^p = \Delta_i^p = \Sigma_i^p = \Pi_i^p = \cdots = \mathrm{PH}$.
5. *If* $\Sigma_i^p = \Pi_i^p$ *for some* i*, then* PH *collapses to* $\Sigma_i^p = \Pi_i^p = \Delta_{i+1}^p = \Sigma_{i+1}^p = \Pi_{i+1}^p = \cdots = \mathrm{PH}$.
6. *There are* $\leq_{\mathrm{m}}^{\mathrm{p}}$*-complete problems for each of the classes* Σ_i^p, Π_i^p, *and* Δ_i^p*. In contrast, if* PH *has a* $\leq_{\mathrm{m}}^{\mathrm{p}}$*-complete problem, then* PH *collapses to a finite level, i.e.,* $\mathrm{PH} = \Sigma_k^p = \Pi_k^p$ *for some* k.

Definition 8.12 (Low hierarchy and high hierarchy within NP). *For* $k \geq 0$*, define the* k*th level of the*

- low hierarchy $\mathrm{LH} = \bigcup_{k \geq 0} \mathrm{Low}_k$ in NP *by* $\mathrm{Low}_k = \{L \in \mathrm{NP} \mid \Sigma_k^{p,L} \subseteq \Sigma_k^p\}$;
- high hierarchy $\mathrm{HH} = \bigcup_{k \geq 0} \mathrm{High}_k$ in NP *by* $\mathrm{High}_k = \{H \in \mathrm{NP} \mid \Sigma_{k+1}^p \subseteq \Sigma_k^{p,H}\}$.

Informally put, a set L is in Low_k if and only if it is useless as an oracle for a Σ_k^p computation. All information contained in $L \in \mathrm{Low}_k$ can be computed by the Σ_k^p machine itself without the help of an oracle. On the other hand, a set H in High_k is so rich and provides so useful information that the computational power of a Σ_k^p machine is increased by the maximum amount an NP set can provide: it "jumps" to the next level of the polynomial hierarchy. That is, by using an oracle H from High_k, a Σ_k^p machine can simulate any Σ_{k+1}^p computation. Thus, H is as useful for a Σ_k^p machine as any NP-complete set.

For $k = 0$, the question of whether or not $\Sigma_k^p \neq \Sigma_{k+1}^p$ is nothing other than the P-versus-NP question. Theorem 8.13 lists some important properties of these hierarchies, omitting the proofs, see [212] and Exercise *8-2*. For the class coAM mentioned in the first item of Theorem 8.13, the reader is referred to the definition of the Arthur-Merlin hierarchy introduced in Subsection 7.5.1; cf. Definition 7.15. Ladner's theorem (Theorem 8.7) is a special case (for $n = 0$) of item 7 in Theorem 8.13.

Theorem 8.13 (Schöning).

1. $\mathrm{Low}_0 = \mathrm{P}$ *and* $\mathrm{Low}_1 = \mathrm{NP} \cap \mathrm{coNP}$ *and* $\mathrm{NP} \cap \mathrm{coAM} \subseteq \mathrm{Low}_2$.

2. $\mathrm{High}_0 = \{H \mid H \text{ is } \leq_{\mathrm{T}}^{\mathrm{p}}\text{-complete in NP}\}$.
3. $\mathrm{High}_1 = \{H \mid H \text{ is } \leq_{\mathrm{sT}}^{\mathrm{NP}}\text{-complete in NP}\}$.
4. $\mathrm{Low}_0 \subseteq \mathrm{Low}_1 \subseteq \cdots \subseteq \mathrm{Low}_k \subseteq \cdots \subseteq \mathrm{LH} \subseteq \mathrm{NP}$.
5. $\mathrm{High}_0 \subseteq \mathrm{High}_1 \subseteq \cdots \subseteq \mathrm{High}_k \subseteq \cdots \subseteq \mathrm{HH} \subseteq \mathrm{NP}$.
6. *For each* $n \geq 0$, $\mathrm{Low}_n \cap \mathrm{High}_n$ *is nonempty if and only if* $\Sigma_n^p = \Sigma_{n+1}^p = \cdots = \mathrm{PH}$.
7. *For each* $n \geq 0$, NP *contains sets that are neither in* Low_n *nor in* High_n *if and only if* $\Sigma_n^p \neq \Sigma_{n+1}^p$.
8. *There exist sets in* NP *that are neither in* LH *nor in* HH *if and only if* PH *is a strictly infinite hierarchy, i.e., if and only if* PH *does not collapse to a finite level.*

8.4.2. Graph isomorphism is in the low hierarchy

We now turn to the result that the graph isomorphism problem (GI) is in Low_2, the second level of the low hierarchy. This result provides strong evidence against the NP-completeness of GI, as can be seen as follows. If GI were NP-complete, then GI would be in $\mathrm{High}_0 \subseteq \mathrm{High}_2$, since by Theorem 8.13 High_0 contains exactly the $\leq_{\mathrm{T}}^{\mathrm{p}}$-complete NP sets and in particular the $\leq_{\mathrm{m}}^{\mathrm{p}}$-complete sets in NP. Again by Theorem 8.13, we have that $\mathrm{Low}_2 \cap \mathrm{High}_2$ is nonempty if and only if PH collapses to Σ_2^p, which is considered very unlikely.

To prove the lowness of the graph isomorphism problem, we first need a technical prerequisite, the so-called hashing lemma, stated here as Lemma 8.15. Hashing is method widely used in computer science for dynamic data management. The idea is the following. Assign to every data set some (short) key in a unique manner. The set of all potential keys, called the universe U, is usually very large. In contrast, the set $V \subseteq U$ of those keys actually used is often much smaller. A *hash function* $h : U \to T$ maps the elements of U to the *hash table* $T = \{0, 1, \ldots, k-1\}$. Hash functions are many-to-one mappings. That is, distinct keys from U can be mapped to the same address in T. However, if possible, one wishes to map two distinct keys from V to distinct addresses in T. That is, one seeks to avoid collisions on the set of keys actually used. If possible, a hash function thus should be injective on V.

Among the various known hashing techniques, we are here interested in *universal hashing*, which was introduced by Carter and Wegman [38] in 1979. The idea is to randomly choose a hash function from a suitable family of hash functions. This method is universal in the sense that it does no longer depend on a particular set V of keys actually used. Rather, it seeks to avoid collisions with high probability on *all* sufficiently small sets V. The probability here is with respect to the random choice of the hash function.

In what follows, we assume that keys are encoded as strings over the alphabet $\Sigma = \{0, 1\}$. The set of all length n strings in Σ^* is denoted by Σ^n.

Definition 8.14 (Hashing). *Let $\Sigma = \{0,1\}$, and let m and t be positive integers with $t > m$. A* hash function *$h : \Sigma^t \to \Sigma^m$ is a linear mapping determined by a boolean $t \times m$ matrix $B_h = (b_{i,j})_{i,j}$, where $b_{i,j} \in \{0,1\}$. For $x \in \Sigma^t$ and $1 \leq j \leq m$, the jth bit of $y = h(x)$ in Σ^m is given by $y_j = (b_{1,j} \wedge x_1) \oplus (b_{2,j} \wedge x_2) \oplus \cdots \oplus (b_{t,j} \wedge x_t)$, where $\oplus$ denotes the logical* exclusive-or *operation, i.e.,*

$$a_1 \oplus a_2 \oplus \cdots \oplus a_n = 1 \iff |\{i \mid a_i = 1\}| \equiv 1 \bmod 2 \; .$$

Let $\mathcal{H}_{t,m}$ be a family of hash functions for the parameters t and m:

$$\mathcal{H}_{t,m} = \{h : \Sigma^t \to \Sigma^m \mid B_h \text{ is a boolean } t \times m \text{ matrix}\} \; .$$

On $\mathcal{H}_{t,m}$, we assume the uniform distribution: A hash function h is chosen from $\mathcal{H}_{t,m}$ by picking the bits $b_{i,j}$ in B_h independently and uniformly distributed.

Let $V \subseteq \Sigma^t$. For any subfamily $\widehat{\mathcal{H}}$ of $\mathcal{H}_{t,m}$, we say there is a collision on V *if*

$$(\exists \vec{v} \in V)\,(\forall h \in \widehat{\mathcal{H}})\,(\exists \vec{x} \in V)\,[\vec{v} \neq \vec{x} \wedge h(\vec{v}) = h(\vec{x})] \; .$$

Otherwise, $\widehat{\mathcal{H}}$ is said to be collision-free on V.

A collision on V means that none of the hash functions in a subfamily $\widehat{\mathcal{H}}$ is injective on V. The following lemma says that, if V is sufficiently small, a randomly chosen subfamily of $\mathcal{H}_{t,m}$ must be collision-free. In contrast, if V is too large, collisions cannot be avoided. The proof of Lemma 8.15 is omitted.

Lemma 8.15 (Hashing lemma). *Let $t, m \in \mathbf{N}$ be fixed parameters, where $t > m$. Let $V \subseteq \Sigma^t$ and let $\widehat{\mathcal{H}} = (h_1, h_2, \ldots, h_{m+1})$ be a family of hash functions randomly chosen from $\mathcal{H}_{t,m}$ under the uniform distribution. Let*

$$C(V) = \{\widehat{\mathcal{H}} \mid (\exists v \in V)\,(\forall h \in \widehat{\mathcal{H}})\,(\exists x \in V)\,[v \neq x \wedge h(v) = h(x)]\}$$

be the event that for $\widehat{\mathcal{H}}$ a collision occurs on V. Then, the following two statements hold:

1. *If $|V| \leq 2^{m-1}$, then $C(V)$ occurs with probability at most $1/4$.*
2. *If $|V| > (m+1)2^m$, then $C(V)$ occurs with probability 1.*

In Section 7.5, the Arthur-Merlin hierarchy has been defined, and it was mentioned that this hierarchy collapses to its second level. Here, we are interested in the class coAM, cf. Definition 7.15 in Section 7.5.1.

Theorem 8.16 (Schöning). `GI` *is in* Low_2.

Proof By Theorem 8.13, every $\text{NP} \cap \text{coAM}$ set is in Low_2. Thus, to prove that `GI` in Low_2, it is enough to show that `GI` is in coAM. Let G and H be two graphs with n vertices each. We wish to apply Lemma 8.15. A first idea is to use

$$A(G,H) \quad = \quad \{(F,\varphi) \mid F \cong G \wedge \varphi \in \text{Aut}(F)\} \cup \{(F,\varphi) \mid F \cong H \wedge \varphi \in \text{Aut}(F)\}$$

as the set V from that lemma. By Lemma 7.10, we have $|A(G,H)| = n!$ if $G \cong H$, and $|A(G,H)| = 2n!$ if $G \ncong H$.

The coAM machine we wish to construct for GI is polynomial-time bounded. Thus, the parameters t and m from the hashing lemma must be polynomial in n. So, to apply Lemma 8.15, we would have to choose a polynomial $m = m(n)$ such that

$$n! \leq 2^{m-1} < (m+1)2^m < 2n! \; . \tag{8.4}$$

This would guarantee that the set $V = A(G,H)$ would be large enough to tell two isomorphic graphs G and H apart from two nonisomorphic graphs G and H. Unfortunately, it is not possible to find a polynomial m that satisfies (8.4). Thus, we define a different set V, which yields a gap large enough to distinguish isomorphic graphs from nonisomorphic graphs.

Define $V = A(G,H)^n = \underbrace{A(G,H) \times A(G,H) \times \cdots \times A(G,H)}_{n \text{ times}}$. Now, (8.4) changes to

$$(n!)^n \leq 2^{m-1} < (m+1)2^m < (2n!)^n \; , \tag{8.5}$$

and this inequality can be satisfied by choosing $m = m(n) = 1 + \lceil n \log n! \rceil$, which is polynomially in n as desired.

Construct a coAM machine M for GI as follows. Given the graphs G and H each having n vertices, M first computes the parameter m. The set $V = A(G,H)^n$ contains n-tuples of pairs each having the form (F,φ), where F is a graph with n vertices, and where φ is a permutation in the automorphism group $\mathrm{Aut}(F)$. The elements of V can be suitably encoded as strings over the alphabet $\Sigma = \{0,1\}$, for a suitable polynomial $t = t(n)$. All computations performed so far are deterministic.

Then, M performs Arthur's probabilistic move by randomly choosing a family $\widehat{\mathcal{H}} = (h_1, h_2, \ldots, h_{m+1})$ of hash functions from $\mathcal{H}_{t,m}$ under the uniform distribution. Each hash function $h_i \in \widehat{\mathcal{H}}$ is represented by a boolean $t \times m$ matrix. Thus, the $m+1$ hash functions h_i in $\widehat{\mathcal{H}}$ can be represented as a string $z_{\widehat{\mathcal{H}}} \in \Sigma^*$ of length $p(n)$ for a suitable polynomial p. Modify the collision predicate $C(V)$ defined in the hashing lemma as follows:

$$B = \{(G,H,z_{\widehat{\mathcal{H}}}) \mid (\exists v \in V)\,(\forall i : 1 \leq i \leq m+1)\,(\exists x \in V)\,[v \neq x \wedge h_i(v) = h_i(x)]\} \; .$$

Note that the $\forall$ quantifier in B ranges over only polynomially many i and can thus be evaluated in deterministic polynomial time. It follows that the two $\exists$ quantifiers in B can be merged into a *single* polynomially length-bounded $\exists$ quantifier. By Theorem 8.11, B is a set in $\Sigma_1^p = \mathrm{NP}$. Let N be an NPTM for B. For the string $z_{\widehat{\mathcal{H}}}$ that encodes $m+1$ randomly picked hash functions from $\mathcal{H}_{t,m}$, M now simulates the computation of $N(G,H,z_{\widehat{\mathcal{H}}})$. This corresponds to Merlin's move. Finally, M accepts its input (G,H) if and only if $N(G,H,z_{\widehat{\mathcal{H}}})$ accepts.

We now estimate the probability (taken over the random choice of the hash functions in $z_{\widehat{\mathcal{H}}}$ that M accepts its input (G,H). If G and H isomorphic, then $|A(G,H)| = n!$ by Lemma 7.10. Inequality (8.5) implies $|V| = (n!)^n \leq 2^{m-1}$. By Lemma 8.15, the probability that $(G,H,z_{\widehat{\mathcal{H}}})$ is in B (and that $M(G,H)$ thus accepts) is at most $1/4$. However, if G and H are nonisomorphic, Lemma 7.10 implies that $|A(G,H)| = 2n!$. Inequality (8.5) now gives $|V| = (2n!)^n > (m+1)2^m$. By

Lemma 8.15, the probability that $(G, H, z_{\widehat{\pi}})$ is in B and $M(G, H)$ thus accepts is 1. It follows that GI is in coAM as desired. ∎

8.4.3. Graph isomorphism is in SPP

The probabilistic complexity class RP was introduced in Definition 7.13 in Subsection 7.3.1. In this section, two other probabilistic complexity classes are important that we will now define: PP and SPP, which stand for *P*robabilistic *P*olynomial Time and *S*toic *P*robabilistic *P*olynomial Time, respectively.

Definition 8.17 (PP and SPP). *The class* PP *contains exactly those problems A for which there exists an NPTM M such that for each input x: If $x \in A$ then $M(x)$ accepts with probability at least $1/2$, and if $x \notin A$ then $M(x)$ accepts with probability less than $1/2$.*

For any NPTM M running on input x, let $\mathrm{acc}_M(x)$ denote the number of accepting computation paths of $M(x)$ and let $\mathrm{rej}_M(x)$ denote the number of rejecting computation paths of $M(x)$. Define $\mathrm{gap}_M(x) = \mathrm{acc}_M(x) - \mathrm{rej}_M(x)$.

The class SPP *contains exactly those problems A for which there exists an NPTM M such that for each input x: $(x \in A \implies \mathrm{gap}_M(x) = 1)$ and $(x \notin A \implies \mathrm{gap}_M(x) = 0)$.*

In other words, an SPP machine is "stoic" in the sense that its "gap" (i.e., the difference between its accepting and rejecting computation paths) can take on only two out of an exponential number of possible values, namely 1 and 0. Unlike PP, SPP is a so-called "*promise class*". since an SPP machine M "promises" that $\mathrm{gap}_M(x) \in \{0, 1\}$ for each x.

The notion of lowness can be defined for any relativizable complexity class $\mathcal{C}$: A set A is said to be *$\mathcal{C}$-low* if and only if $\mathcal{C}^A = \mathcal{C}$. In particular, for each k, the kth level Low_k of the low hierarchy within NP (see Definition 8.12) contains exactly the NP sets that are Σ_k^p-low. It is known that all sets in SPP are PP-low. This and other useful properties of SPP are listed in the following theorem without proof; see also [66, 129, 130].

Theorem 8.18

1. SPP *is* PP*-low, i.e.,* $\mathrm{PP}^{\mathrm{SPP}} = \mathrm{PP}$.
2. SPP *is self-low, i.e.,* $\mathrm{SPP}^{\mathrm{SPP}} = \mathrm{SPP}$.
3. *Let A be a set in* NP *via some NPTM N and let L be a set in* SPP^A *via some NPOTM M such that, for each input x, $M^A(x)$ asks only queries q satisfying* $\mathrm{acc}_N(q) \leq 1$. *Then, L is in* SPP.
4. *Let A be a set in* NP *via some NPTM N and let f be a function in* FP^A *via some DPOTM M such that, for each input x, $M^A(x)$ asks only queries q satisfying* $\mathrm{acc}_N(q) \leq 1$. *Then, f is in* $\mathrm{FP}^{\mathrm{SPP}}$.

The following theorem says that the lexicographically smallest permutation in a right coset (see Definition 7.6 in Section 7.1.3) can be determined efficiently. The lexicographic order on $\mathfrak{S}_n$ is defined in Example 8.3.

Theorem 8.19 *Let $\mathfrak{G} \leq \mathfrak{S}_n$ be a permutation group with $\mathfrak{G} = \langle G \rangle$ and let π be a permutation in $\mathfrak{S}_n$. There is a polynomial-time algorithm that, given (G, π), computes the lexicographically smallest permutation in the right coset $\mathfrak{G}\pi$ of $\mathfrak{G}$ in $\mathfrak{S}_n$.*

Proof We now state our algorithm LERC for computing the lexicographically smallest permutation in the right coset $\mathfrak{G}\pi$ of $\mathfrak{G}$ in $\mathfrak{S}_n$, where the permutation group $\mathfrak{G}$ is given by a generator G, see Definition 7.6 in Subsection 7.1.3.

LERC(G, π)

```
1  compute the tower 𝔊^(n) ≤ 𝔊^(n−1) ≤ ··· ≤ 𝔊^(1) ≤ 𝔊^(0) of stabilisers in 𝔊
2      φ_0 ← π
3      for i ← 0 to n − 1
4          do x ← i + 1
5              compute the element y in the orbit 𝔊^(i)(x) for which φ_i(y) is minimum
6              compute a permutation τ_i in 𝔊^(i) such that τ_i(x) = y
7              φ_{i+1} ← τ_i φ_i
8      return φ_n
```

By Theorem 7.1, the tower $\mathbf{id} = \mathfrak{G}^{(n)} \leq \mathfrak{G}^{(n-1)} \leq \cdots \leq \mathfrak{G}^{(1)} \leq \mathfrak{G}^{(0)} = \mathfrak{G}$ of stabilisers of $\mathfrak{G}$ can be computed in polynomial time. More precisely, for each i with $1 \leq i \leq n$, the complete right transversals T_i $\mathfrak{G}^{(i)}$ in $\mathfrak{G}^{(i-1)}$ are determined, and thus a strong generator $S = \bigcup_{i=1}^{n-1} T_i$ of $\mathfrak{G}$.

Note that $\varphi_0 = \pi$ and $\mathfrak{G}^{(n-1)} = \mathfrak{G}^{(n)} = \mathbf{id}$. Thus, to prove that the algorithm works correctly, it is enough to show that for each i with $0 \leq i \leq n-1$, the lexicographically smallest permutation of $\mathfrak{G}^{(i)}\varphi_i$ is contained in $\mathfrak{G}^{(i+1)}\varphi_{i+1}$. By induction, it follows that $\mathfrak{G}^{(n)}\varphi_n = \{\varphi_n\}$ also contains the lexicographically smallest permutation of $\mathfrak{G}\pi = \mathfrak{G}^{(0)}\varphi_0$. Thus, algorithm LERC indeed outputs the lexicographically smallest permutation of φ_n of $\mathfrak{G}\pi$.

To prove the above claim, let us denote the orbit of an element $x \in [n]$ in a permutation group $\mathfrak{H} \leq \mathfrak{S}_n$ by $\mathfrak{H}(x)$. Let τ_i be the permutation in $\mathfrak{G}^{(i)}$ that maps $i+1$ onto the element y in the orbit $\mathfrak{G}^{(i)}(i+1)$ for which $\varphi_i(y) = x$ is the minimal element in the set $\{\varphi_i(z) \mid z \in \mathfrak{G}^{(i)}(i+1)\}$.

By Theorem 7.1, the orbit $\mathfrak{G}^{(i)}(i+1)$ can be computed in polynomial time. Since $\mathfrak{G}^{(i)}(i+1)$ contains at most $n-i$ elements, y can be determined efficiently. Our algorithm ensures that $\varphi_{i+1} = \tau_i\varphi_i$. Since every permutation in $\mathfrak{G}^{(i)}$ maps each element of $[i]$ onto itself, and since $\tau_i \in \mathfrak{G}^{(i)}$, it follows that for each j with $1 \leq j \leq i$, for each $\tau \in \mathfrak{G}^{(i)}$, and for each $\sigma \in \mathfrak{G}^{(i+1)}$,

$$(\sigma\varphi_{i+1})(j) = \varphi_{i+1}(j) = (\tau_i\varphi_i)(j) = \varphi_i(j) = (\tau\varphi_i)(j) \ .$$

In particular, it follows for the lexicographically smallest permutation μ in $\mathfrak{G}^{(i)}\varphi_i$ that every permutation from $\mathfrak{G}^{(i+1)}\varphi_{i+1}$ must coincide with μ on the first i elements,

i.e. on $[i]$.

Moreover, for each $\sigma \in \mathfrak{G}^{(i+1)}$ and for the element $x = \varphi_i(y)$ defined above, we have

$$(\sigma\varphi_{i+1})(i+1) = \varphi_{i+1}(i+1) = (\tau_i\varphi_i)(i+1) = x\ .$$

Clearly, $\mathfrak{G}^{(i+1)}\varphi_{i+1} = \{\varphi \in \mathfrak{G}^{(i)}\varphi_i \mid \varphi(i+1) = x\}$. The claim now follows from the fact that $\mu(i+1) = x$ for the lexicographically smallest permutation μ of $\mathfrak{G}^{(i)}\varphi_i$.

Thus, LERC is a correct algorithm. It easy to see that it is also efficient. ∎

Theorem 8.19 can easily be extended to Corollary 8.20, see Exercise *8-3*.

Corollary 8.20 *Let $\mathfrak{G} \leq \mathfrak{S}_n$ be a permutation group with $\mathfrak{G} = \langle G \rangle$, and let π and ψ be two given permutations in $\mathfrak{S}_n$. There exists a polynomial-time algorithm that, given (G, π, ψ), computes the lexicographically smallest permutation of $\psi\mathfrak{G}\pi$.*

We now prove Theorem 8.21, the main result of this section.

Theorem 8.21 (Arvind and Kurur). `GI` *is in* SPP.

Proof Define the (functional) problem `AUTO` as follows: Given a graph G, compute a strong generator of the automorphism group $\mathrm{Aut}(G)$; see Definition 7.6 and the subsequent paragraph and Definition 7.7 for these notions. By Mathon's [162] result, the problems `AUTO` and `GI` are Turing-equivalent (see also [130]), i.e., `AUTO` is in $\mathrm{FP}^{\mathtt{GI}}$ and `GI` is in $\mathrm{P}^{\mathtt{AUTO}}$. Thus, it is enough to show that `AUTO` is in $\mathrm{FP}^{\mathrm{SPP}}$ because the self-lowness of SPP stated in Theorem 8.18 implies that `GI` is in $\mathrm{P}^{\mathtt{AUTO}} \subseteq \mathrm{SPP}^{\mathrm{SPP}} \subseteq \mathrm{SPP}$, which will complete the proof.

So, our goal is to find an $\mathrm{FP}^{\mathrm{SPP}}$ algorithm for `AUTO`. Given a graph G, this algorithm has to compute a strong generator $S = \bigcup_{i=1}^{n-1} T_i$ for $\mathrm{Aut}(G)$, where

$$\mathbf{id} = \mathrm{Aut}(G)^{(n)} \leq \mathrm{Aut}(G)^{(n-1)} \leq \cdots \leq \mathrm{Aut}(G)^{(1)} \leq \mathrm{Aut}(G)^{(0)} = \mathrm{Aut}(G)$$

is the tower of stabilisers of $\mathrm{Aut}(G)$ and T_i, $1 \leq i \leq n$, is a complete right transversal of $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$.

Starting with the trivial case, $\mathrm{Aut}(G)^{(n)} = \mathbf{id}$, we build step by step a strong generator for $\mathrm{Aut}(G)^{(i)}$, where i is decreasing. Eventually, we will thus obtain a strong generator for $\mathrm{Aut}(G)^{(0)} = \mathrm{Aut}(G)$. So suppose a strong generator $S_i = \bigcup_{j=i}^{n-1} T_j$ for $\mathrm{Aut}(G)^{(i)}$ has already been found. We now describe how to determine a complete right transversal T_{i-1} of $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$ by our $\mathrm{FP}^{\mathrm{SPP}}$ algorithm. Define the oracle set

$$A = \left\{ (G, S, i, j, \pi) \;\middle|\; \begin{array}{l} S \subseteq \mathrm{Aut}(G) \text{ and } \langle S \rangle \text{ is a pointwise stabilizer of } [i] \\ \text{in } \mathrm{Aut}(G),\ \pi \text{ is a partial permutation, which pointwise} \\ \text{stabilises } [i-1], \text{ and } \pi(i) = j, \text{ and there is a } \tau \text{ in} \\ \mathrm{Aut}(G)^{(i-1)} \text{ with } \tau(i) = j \text{ and } \mathrm{LERC}(S, \tau) \text{ extends } \pi \end{array} \right\}.$$

By Theorem 8.19, the lexicographically smallest permutation $\mathrm{LERC}(S, \tau)$ of the right coset $\langle S \rangle\tau$ can be determined in polynomial time by our algorithm. The partial permutation π belongs to the input (G, S, i, j, π), since we wish to use A as an oracle in order to find the lexicographically smallest permutation by prefix search; cf. Example 8.3.

Consider the following NPTM N for A:

N(G, S, i, j, π)

1 verify that $S \subseteq \mathrm{Aut}(G)^{(i)}$
2 nondeterministically guess a permutation $\tau \in \mathfrak{S}_n$; // G has n vertices
3 **if** $\tau \in \mathrm{Aut}(G)^{(i-1)}$ and $\tau(i) = j$ and τ extends π and $\tau = \mathrm{LERC}(S, \tau)$
4 **then** accept and halt
5 **else** reject and halt

Thus, A is in NP. Note that if $\tau(i) = j$ then $\sigma(i) = j$, for each permutation σ in the right coset $\langle S\rangle\tau$.

We now show that if $\langle S\rangle = \mathrm{Aut}(G)^{(i)}$ then the number of accepting computation paths of N on input (G, S, i, j, π) is either 0 or 1. In general, $\mathrm{acc}_N(G, S, i, j, \pi) \in \{0, |\mathrm{Aut}(G)^{(i)}|/|\langle S\rangle|\}$.

Suppose (G, S, i, j, π) is in A and $\langle S\rangle = \mathrm{Aut}(G)^{(i)}$. If $\tau(i) = j$ for some $\tau \in \mathrm{Aut}(G)^{(i-1)}$ and $j > i$, then the right coset $\langle S\rangle\tau$ contains exactly those permutations in $\mathrm{Aut}(G)^{(i-1)}$ that map i to j. Thus, the only accepting computation path of $N(G, S, i, j, \pi)$ corresponds to the unique lexicographically smallest permutation $\tau = \mathrm{LERC}(S, \tau)$. If, on the other hand, $\langle S\rangle$ is a strict subgroup of $\mathrm{Aut}(G)^{(i)}$, then $\mathrm{Aut}(G)^{(i)}\tau$ can be written as the disjoint union of $k = |\mathrm{Aut}(G)^{(i)}|/|\langle S\rangle|$ right cosets of $\langle S\rangle$. In general, $N(G, S, i, j, \pi)$ thus possesses k accepting computation paths if (G, S, i, j, π) is in A, and otherwise it has no accepting computation path.

M-A(G)

1 set $T_i := \{\mathrm{id}\}$ for each i, $0 \le i \le n-2$; // G has n vertices
$\triangleright$ T_i will be a complete right transversal of $\mathrm{Aut}(G)^{(i+1)}$ in $\mathrm{Aut}(G)^{(i)}$
3 set $S_i := \emptyset$ for each i, $0 \le i \le n-2$
4 set $S_{n-1} := \{\mathrm{id}\}$
$\triangleright$ S_i will be a strong generator for $\mathrm{Aut}(G)^{(i)}$
5 **for** $i \leftarrow n-1$ **downto** 1
$\triangleright$ S_i is already found at the start
$\triangleright$ of the ith iteration, and S_{i-1} will now be computed
6 **do** let $\pi : [i-1] \to [n]$ be the partial permutation
with $\pi(a) = a$ for each $a \in [i-1]$
7 $\triangleright$ for $i = 1$, π is the nowhere defined partial permutation

```
8         for j ← i + 1 to n
9             do π̂ := πj, i.e., π̂ extends π by the pair (i, j) with π̂(i) = j
10                if ((G, S_i, i, j, π̂) ∈ A)  {
11    then                                  ▷ construct the smallest permutation
                                              in Aut(G)^(i−1) mapping i to j by prefix search
12                for k ← i + 1 to n
13                      do find the element ℓ
                          not in the image of π̂ with (G, S_i, i, j, π̂ℓ) ∈ A
14                π̂ := π̂ℓ
15                                          ▷ now, π̂ is a total permutation in 𝔖_n
16                    T_{i−1} := T_{i−1} ∪ π̂
              now, T_{i−1} is a complete right transversal of Aut(G)^(i) in Aut(G)^(i−1)
17    S_{i−1} := S_i ∪ T_{i−1}
18  return S_0                      ▷ S_0 is a strong generator for Aut(G) = Aut(G)^(0)
```

The above algorithm is an FP^A algorithm M^A for AUTO. The DPOTM M makes only queries $q = (G, S_i, i, j, \pi)$ to its oracle A for which $\langle S_i \rangle = \mathrm{Aut}(G)^{(i)}$. Thus, $\mathrm{acc}_N(q) \leq 1$ for each query q actually asked. By item 4 of Theorem 8.18, it follows that AUTO is in $\mathrm{FP}^{\mathrm{SPP}}$.

The claim that the output S_0 of $M^A(G)$ indeed is a strong generator for $\mathrm{Aut}(G) = \mathrm{Aut}(G)^{(0)}$ can be shown by induction on n. The induction base is $n-1$, and $S_{n-1} = \{\mathrm{id}\}$ of course generates $\mathrm{Aut}(G)^{(n-1)} = \mathbf{id}$.

For the induction step, assume that prior to the ith iteration a strong generator S_i for $\mathrm{Aut}(G)^{(i)}$ has already been found. We now show that after the ith iteration the set $S_{i-1} = S_i \cup T_{i-1}$ is a strong generator for $\mathrm{Aut}(G)^{(i-1)}$. For each j with $i+1 \leq j \leq n$, the oracle query "$(G, S_i, i, j, \hat{\pi}) \in A$?" checks whether there is a permutation in $\mathrm{Aut}(G)^{(i-1)}$ mapping i to j. By prefix search, which is performed by making suitable oracle queries to A again, the lexicographically smallest permutation $\hat{\pi}$ in $\mathrm{Aut}(G)^{(i-1)}$ with $\hat{\pi}(i) = j$ is constructed. Note that, as claimed above, only queries q satisfying $\mathrm{acc}_N(q) \leq 1$ are made to A, since S_i is a strong generator for $\mathrm{Aut}(G)^{(i)}$, so $\langle S_i \rangle = \mathrm{Aut}(G)^{(i)}$. By construction, after the ith iteration, T_{i-1} is a complete right transversal of $\mathrm{Aut}(G)^{(i)}$ in $\mathrm{Aut}(G)^{(i-1)}$. It follows that $S_{i-1} = S_i \cup T_{i-1}$ is a strong generator for $\mathrm{Aut}(G)^{(i-1)}$. Eventually, after n iterations, a strong generator S_0 for $\mathrm{Aut}(G) = \mathrm{Aut}(G)^{(0)}$ is found. ■

From Theorem 8.21 and the first two items of Theorem 8.18, we obtain Corollary 8.22.

Corollary 8.22 GI *is low for both* SPP *and* PP, *i.e.,* $\mathrm{SPP}^{\mathrm{GI}} = \mathrm{SPP}$ *and* $\mathrm{PP}^{\mathrm{GI}} = \mathrm{PP}$.

Exercises

8.4-1 By Definition 8.9, $A \leq_{\mathrm{T}}^{\mathrm{p}} B$ if and only if $A \in \mathrm{P}^B$. Show that $A \leq_{\mathrm{T}}^{\mathrm{p}} B$ if and only if $\mathrm{P}^A \subseteq \mathrm{P}^B$.

8.4-2 Show that the set Pre-Iso defined in Example 8.3 is in NP. Moreover, prove

that the machine N defined in Example 8.3 runs in polynomial time, i.e., show that N is a DPOTM.

Problems

8-1 Strong NPOTM

A ***strong NPOTM*** is an NPOTM with three types of final states, i.e., the set F of final states of M is partitioned into F_a (accepting states), F_r (rejecting states), and $F_?$ ("don't know" states) such that the following holds: If $x \in A$ then $M^B(x)$ has at least one computation path halting in an accepting state from F_a but no computation path halting in a rejecting state from F_r. If $x \notin A$ then $M^B(x)$ has at least one computation path halting in a rejecting state from F_r but no computation path halting in an accepting state from F_a. In both cases $M^B(x)$ may have computation paths halting in "don't know" states from $F_?$. In other words, strong NPOTMs are machines that never lie. Prove the following two assertions:

(a) $A \leq^{\mathrm{NP}}_{\mathrm{sT}} B$ if and only if there exists a strong NPOTM M with $A = L(M^B)$.

(b) $A \leq^{\mathrm{NP}}_{\mathrm{sT}} B$ if and only if $\mathrm{NP}^A \subseteq \mathrm{NP}^B$.

Hint. Look at Exercise 8.4-1.

8-2 Proofs

Prove the assertions from Theorems 8.11 and 8.13. (Some are far from being trivial!)

8-3 Modification of the proofs

Modify the proof of Theorem 8.19 so as to obtain Corollary 8.20.

Chapter notes

Parts of the Chapters 7 and 8 are based on the book [205] that provides the proofs omitted here, such as those of Theorems 8.11, 8.13, and 8.18 and of Lemma 8.15, and many more details.

More background on complexity theory can be found in the books [106, 184, 249, 253]. A valuable source on the theory of NP-completeness is still the classic [75] by Garey and Johnson. The $\leq^{\mathrm{p}}_{\mathrm{T}}$-reducibility was introduced by Cook [48], and the $\leq^{\mathrm{p}}_{\mathrm{m}}$-reducibility by Karp [128]. A deep and profound study of polynomial-time reducibilities is due to Ladner, Lynch, and Selman [145].

Exercise *8-1* and Problem *8-1* are due to Selman [217].

Dantsin et al. [56] obtained an upper bound of $\tilde{O}(1.481^n)$ for the deterministic time complexity of 3-SAT, which was further improved by Brueggemann and Kern [32] to $\tilde{O}(1.4726^n)$. The randomised algorithm presented in Section 8.3.2 is due to Schöning [214]; it is based on a "limited local search with restart". For k-SAT with $k \geq 4$, the algorithm by Paturi et al. [186] is slightly better than Schöning's algorithm. Iwama and Tamaki [119] combined the ideas of Schöning [214] and Paturi et al. [186] to obtain a bound of $\tilde{O}(1.324^n)$ for k-SAT with $k \in \{3, 4\}$. For k-SAT with

Algorithm	Type	3-SAT	4-SAT	5-SAT	6-SAT
Backtracking	det.	$\tilde{O}(1.913^n)$	$\tilde{O}(1.968^n)$	$\tilde{O}(1.987^n)$	$\tilde{O}(1.995^n)$
Monien and Speckenmeyer [172]	det.	$\tilde{O}(1.618^n)$	$\tilde{O}(1.839^n)$	$\tilde{O}(1.928^n)$	$\tilde{O}(1.966^n)$
Dantsin et al. [56]	det.	$\tilde{O}(1.481^n)$	$\tilde{O}(1.6^n)$	$\tilde{O}(1.667^n)$	$\tilde{O}(1.75^n)$
Brueggemann and Kern [32]	det.	$\tilde{O}(1.4726^n)$	—	—	—
Paturi et al. [186]	prob.	$\tilde{O}(1.362^n)$	$\tilde{O}(1.476^n)$	$\tilde{O}(1.569^n)$	$\tilde{O}(1.637^n)$
Schöning [214]	prob.	$\tilde{O}(1.334^n)$	$\tilde{O}(1.5^n)$	$\tilde{O}(1.6^n)$	$\tilde{O}(1.667^n)$
Iwama and Tamaki [119]	prob.	$\tilde{O}(1.324^n)$	$\tilde{O}(1.474^n)$	—	—

Figure 8.7. Runtimes of some algorithms for the satisfiability problem.

$k \geq 5$, their algorithm is not better than that by Paturi et al. [186].

Figure 8.7 gives an overview over some algorithms for the satisfiability problem.

For a thorough, comprehensive treatment of the graph isomorphism problem the reader is referred to the book by Köbler, Schöning, and Torán [131], particularly under complexity-theoretic aspects. Hoffman [109] investigates group-theoretic algorithms for the graph isomorphism problem and related problems.

The polynomial hierarchy was introduced by Meyer and Stockmeyer [164, 235]. In particular, Theorem 8.11 is due to them. Schöning [212] introduced the low hierarchy and the high hierarchy within NP. The results stated in Theorem 8.13 are due to him [212]. He also proved that GI is in Low_2, see [213]. Köbler et al. [129, 130] obtained the first lowness results of GI for probabilistic classes such as PP. These results were improved by Arvind and Kurur [11] who proved that GI is even in SPP. The class SPP generalises Valiant's class UP, see [246]. So-called "promise classes" such as UP and SPP have been thoroughly studied in a number of papers; see, e.g., [11, 28, 66, 129, 130, 196, 204]. Lemma 8.15 is due to Carter and Wegman [38].

The author is grateful to Uwe Schöning for his helpful comments on an earlier version of this chapter and for sending the slides of one of this talks that helped simplifying the probability analysis for the algorithm RANDOM-SAT sketched in Subsection 8.3; a more comprehensive analysis can be found in Schöning's book [215]. Thanks are also due to Dietrich Stoyan, Robert Stoyan, Sigurd Assing, Gábor Erdélyi and Holger Spakowski for proofreading previous versions of Chapters 7 and 8. This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG) under grants RO 1202/9-1 and RO 1202/9-3 and by the Alexander von Humboldt Foundation in the TransCoop program.

III. NUMERICAL METHODS

9. Competitive Analysis

In on-line computation an algorithm must make its decisions based only on past events without secure information on future. Such methods are called ***on-line algorithms***. On-line algorithms have many applications in different areas such as computer science, economics and operations research.
The first results in this area appeared around 1970, and later since 1990 more and more researchers have started to work on problems related to on-line algorithms. Many subfields have been developed and investigated. Nowadays new results of the area have been presented on the most important conferences about algorithms. This chapter does not give a detailed overview about the results, because it is not possible in this framework. The goal of the chapter is to show some of the main methods of analysing and developing on-line algorithms by presenting some subareas in more details.

In the next section we define the basic notions used in the analysis of on-line algorithms. After giving the most important definitions we present one of the best-known on-line problems – the on-line k-server problem – and some of the related results. Then we deal with a new area by presenting on-line problems belonging to computer networks. In the next section the on-line bin packing problem and its multidimensional generalisations are presented. Finally in the last section of this chapter we show some basic results concerning the area of on-line scheduling.

9.1. Notions, definitions

Since an on-line algorithm makes its decisions based on partial information without knowing the whole instance in advance, we cannot expect it to give the optimal solution which can be given by an algorithm having full information. An algorithm which knows the whole input in advance is called ***off-line algorithm***.

There are two main methods to measure the performance of on-line algorithms. One possibility is to use ***average case analysis*** where we hypothesise some distribution on events and we study the expected total cost.

The disadvantage of this approach is that usually we do not have any information about the distribution of the possible inputs. In this chapter we do not use the average case analysis.

An another approach is a worst case analysis, which is called ***competitive analysis***. In this case we compare the objective function value of the solution produced by the on-line algorithm to the optimal off-line objective function value.

In case of on-line minimisation problems an on-line algorithm is called ***C-competitive***, if the cost of the solution produced by the on-line algorithm is at most C times more than the optimal off-line cost for each input. The ***competitive ratio of an algorithm*** is the smallest C for which the algorithm is C-competitive.

For an arbitrary on-line algorithm ALG we denote the objective function value achieved on input I by $\mathrm{ALG}(I)$. The optimal off-line objective function value on I is denoted by $\mathrm{OPT}(I)$. Using this notation we can define the competitiveness as follows.

Algorithm ALG is C-competitive, if $\mathrm{ALG}(I) \leq C \cdot \mathrm{OPT}(I)$ is valid for each input I.

There are two further versions of the competitiveness which are often used. For a minimisation problem an algorithm ALG is called ***weakly C-competitive***, if there exists such a constant B that $\mathrm{ALG}(I) \leq C \cdot \mathrm{OPT}(I) + B$ is valid for each input I.

The ***weak competitive ratio of an algorithm*** is the smallest C for which the algorithm is weakly C-competitive.

A further version of the competitive ratio is the asymptotic competitive ratio. For minimisation problems the ***asymptotic competitive ratio*** of algorithm ALG ($R^{\infty}_{\mathrm{ALG}}$) can be defined as follows:

$$R^{n}_{\mathrm{ALG}} = \sup \left\{ \frac{\mathrm{ALG}(I)}{\mathrm{OPT}(I)} \;\mid\; \mathrm{OPT}(I) = n \right\} ,$$

$$R^{\infty}_{\mathrm{ALG}} = \limsup_{n \to \infty} R^{n}_{\mathrm{ALG}} \cdot$$

An algorithm is called ***asymptotically C-competitive*** if its asymptotic competitive ratio is at most C.

The main property of the asymptotic ratio is that it considers the performance of the algorithm under the assumption that the size of the input tends to ∞. This means that this ratio is not effected by the behaviour of the algorithm on the small size inputs.

Similar definitions can be given for maximisation problems. In that case algorithm ALG is called C-competitive, if $\mathrm{ALG}(I) \geq C \cdot \mathrm{OPT}(I)$ is valid for each input I, and the algorithm is weakly C-competitive if there exists such a constant B that $\mathrm{ALG}(I) \geq C \cdot \mathrm{OPT}(I) + B$ is valid for each input I. The asymptotic ratio for maximisation problems can be given as follows:

$$R^{n}_{\mathrm{ALG}} = \inf \left\{ \frac{\mathrm{ALG}(I)}{\mathrm{OPT}(I)} \;\mid\; \mathrm{OPT}(I) = n \right\} ,$$

$$R^{\infty}_{\mathrm{ALG}} = \liminf_{n \to \infty} R^{n}_{\mathrm{ALG}} \cdot$$

The algorithm is called ***asymptotically C-competitive*** if its asymptotic ratio is at least C.

Many papers are devoted ***randomised on-line algorithms***, in which case the objective function value achieved by the algorithm is a random variable, and the

expected value of this variable is used in the definition of the competitive ratio. Since we consider only deterministic on-line algorithms in this chapter, we do not detail the notions related to randomised on-line algorithms.

9.2. The k-server problem

One of the best-known on-line problems is the on-line k-server problem. To give the definition of the general problem the notion of metric spaces is needed. A pair (M, d) (where M contains the points of the space and d is the distance function defined on the set $M \times M$) is called metric space if the following properties are valid:

- $d(x, y) \geq 0$ for all $x, y \in M$,
- $d(x, y) = d(y, x)$ for all $x, y \in M$,
- $d(x, y) + d(y, z) \geq d(x, z)$ for all $x, y, z \in M$,
- $d(x, y) = 0$ holds if and only if $x = y$.

In the k-server problem a metric space is given, and there are k servers which can move in the space. The decision maker has to satisfy a list of requests appearing at the points of the metric space by sending a server to the point where the request appears.

The problem is on-line which means that the requests arrive one by one, and we must satisfy each request without any information about the further requests. The goal is to minimise the total distance travelled by the servers. In the remaining parts of the section the multiset which contains the points where the servers are is called the ***configuration of the servers***. We use multisets, since different servers can be at the same points of the space.

The first important results for the k-server problem were achieved by Manasse, McGeoch and Sleator. They developed the following algorithm called BALANCE, which we denote by BAL. During the procedure the servers are in different points. The algorithm stores for each server the total distance travelled by the server. The servers and the points in the space where the servers are located are denoted by $s_1, \ldots, s_k$. Let the total distance travelled by the server s_i be D_i. After the arrival of a request at point P algorithm BAL uses server i for which the value $D_i + d(s_i, P)$ is minimal. This means that the algorithm tries to balance the distances travelled by the servers. Therefore the algorithm maintains server configuration $S = \{s_1, \ldots, s_k\}$ and the distances travelled by the servers which distances have starting values $D_1 = \cdots = D_k = 0$. The behaviour of the algorithm on input $I = P_1, \ldots, P_n$ can be given by the following pseudocode:

BAL(I)

```
1  for j ← 1 to n
2      do i ← argmin{D_i + d(s_i, P_j)}
3         serve the request with server i
4         D_i ← D_i + d(s_i, P_j)
5         s_i ← P_j
```

Example 9.1 Consider the two dimensional Euclidean space as the metric space. The points are two dimensional real vectors (x, y), and the distance between (a, b) and (c, d) is $\sqrt{(a-c)^2+(b-d)^2}$. Suppose that there are two servers which are located at points $(0, 0)$ and $(1, 1)$ at the beginning. Therefore at the beginning $D_1 = D_2 = 0$, $s_1 = (0, 0)$, $s_2 = (1, 1)$. Suppose that the first request appears at point $(1, 4)$. Then $D_1 + d((0, 0), (1, 4)) = \sqrt{17} > D_2 + d((1, 1), (1, 4)) = 3$, thus the second server is used to satisfy the request and after the action of the server $D_1 = 0, D_2 = 3$, $s_1 = (0, 0)$, $s_2 = (1, 4)$. Suppose that the second request appears at point $(2, 4)$, so $D_1+d((0, 0), (2, 4)) = \sqrt{20} > D_2+d((1, 4), (2, 4)) = 3+1 = 4$, thus again the second server is used, and after serving the request $D_1 = 0, D_2 = 4$, $s_1 = (0, 0)$, $s_2 = (2, 4)$. Suppose that the third request appears at point $(1, 4)$, so $D_1+d((0, 0), (1, 4)) = \sqrt{17} < D_2 + d((2, 4), (1, 4)) = 4 + 1 = 5$, thus the first server is used, and after serving the request $D_1 = \sqrt{17}, D_2 = 4$, $s_1 = (1, 4)$, $s_2 = (2, 4)$.

The algorithm is efficient in the cases of some particular metric spaces as it is shown by the following statement. The references where the proof of the following theorem can be found are in the chapter notes at the end of the chapter.

Theorem 9.1 *Algorithm* BALANCE *is weakly k-competitive for the metric spaces containing $k + 1$ points.*

The following statement shows that there is no on-line algorithm which is better than k-competitive for the general k-server problem.

Theorem 9.2 *There is no metric space containing at least $k + 1$ points where an on-line algorithm exists with smaller competitive ratio than k.*

Proof Consider an arbitrary metric space containing at least $k + 1$ points and an arbitrary on-line algorithm say ONL. Denote the points of the starting configuration of ONL by $P_1, P_2, \ldots, P_k$, and let P_{k+1} be another point of the metric space. Consider the following long list of requests $I = Q_1, \ldots, Q_n$. The next request appears at the point among $P_1, P_2, \ldots, P_{k+1}$ where ONL has no server.

First calculate the value ONL(I). The algorithm does not have any servers at point Q_{j+1} after serving Q_j, thus the request appeared at Q_j is served by the server located at point Q_{j+1}. Therefore the cost of serving Q_j is $d(Q_j, Q_{j+1})$, which yields

$$\mathrm{ONL}(I) = \sum_{j=1}^{n} d(Q_j, Q_{j+1}) \ ,$$

where Q_{n+1} denotes the point from which the server was sent to serve Q_n. (This is the point where the $(n + 1)$-th request would appear.) Now consider the cost

$\mathrm{OPT}(I)$. Instead of calculating the optimal off-line cost we define k different off-line algorithms, and we use the mean of the costs resulting from these algorithms. Since the cost of each off-line algorithm is at least as much as the optimal off-line cost, the calculated mean is an upper bound for the optimal off-line cost.

We define the following k off-line algorithms, denoted by $\mathrm{OFF}_1, \ldots, \mathrm{OFF}_k$. Suppose that the servers are at points $P_1, P_2, \ldots, P_{j-1}, P_{j+1}, \ldots P_{k+1}$ in the starting configuration of OFF_j. We can move the servers into this starting configuration using an extra constant cost C_j.

The algorithms satisfy the requests as follows. If an algorithm OFF_j has a server at point Q_i, then none of the servers moves. Otherwise the request is served by the server located at point Q_{i-1}. The algorithms are well-defined, if Q_i does not contain a server, then each of the other points $P_1, P_2, \ldots, P_{k+1}$ contains a server, thus there is a server located at Q_{i-1}. Moreover $Q_1 = P_{k+1}$, thus at the beginning each algorithm has a server at the requested point.

We show that the servers of algorithms $\mathrm{OFF}_1, \ldots, \mathrm{OFF}_k$ are always in different configurations. At the beginning this property is valid because of the definition of the algorithms. Now consider the step where a request is served. Call the algorithms which do not move a server for serving the request stable, and the other algorithms unstable. The server configurations of the stable algorithms remain unchanged, so these configurations remain different from each other. Each unstable algorithm moves a server from point Q_{i-1}. This point is the place of the last request, thus the stable algorithms have server at it. Therefore, an unstable algorithm and a stable algorithm cannot have the same configuration after serving the request. Furthermore, each unstable algorithms moves a server from Q_{i-1} to Q_i, thus the server configurations of the unstable algorithms remain different from each other.

So at the arrival of the request at point Q_i the servers of the algorithms are in different configurations. On the other hand, each configuration has a server at point Q_{i-1}, therefore there is only one configuration where there is no server located at point Q_i. Consequently, the cost of serving Q_i is $d(Q_{i-1}, Q_i)$ for one of the algorithms and 0 for the other algorithms.

Therefore

$$\sum_{j=1}^{k} \mathrm{OFF}_j(I) = C + \sum_{i=2}^{n} d(Q_i, Q_{i-1}) \ ,$$

where $C = \sum_{j=1}^{k} C_j$ is an absolute constant which is independent of the input (this is the cost of moving the servers to the starting configuration of the defined algorithms).

On the other hand, the optimal off-line cost cannot be larger than the cost of any of the above defined algorithms, thus $k \cdot \mathrm{OPT}(I) \leq \sum_{j=1}^{k} \mathrm{OFF}_j(I)$. This yields

$$k \cdot \mathrm{OPT}(I) \leq C + \sum_{i=2}^{n} d(Q_i, Q_{i-1}) \leq C + \mathrm{ONL}(I) \ ,$$

which inequality shows that the weak competitive ratio of ONL cannot be smaller than k, since the value $\mathrm{OPT}(I)$ can be arbitrarily large as the length of the input is increasing. ■

There are many interesting results in connection with this problem.have appeared during the next few years. For the general case the first constant-competitive algorithm ($O(2^k)$-competitive) was developed by Fiat, Rabani and Ravid. Later Koutsoupias and Papadimitriou could analyse an algorithm based on the work function technique and they could prove that it is $(2k-1)$-competitive. They could not determine the competitive ratio of the algorithm, but it is a widely believed hypothesis that the algorithm is k-competitive. Determining the competitive ratio of the algorithm, or developing a k-competitive algorithm is still among the most important open problems in the area of on-line algorithms. We present the work function algorithm below.

Denote the starting configuration of the on-line servers by A_0. Then after the t-th request the ***work function*** value belonging to multiset X is the minimal cost needed to serve the first t requests starting at configuration A_0 and ending at configuration X. This value is denoted by $w_t(X)$. The WORK-FUNCTION algorithm is based on the above defined work function. Suppose that A_{t-1} is the server configuration before the arrival of the t-th request, and denote the place of the t-th request by R_t. The WORK-FUNCTION algorithm uses server s to serve the request for which the value $w_{t-1}(A_{t-1} \setminus \{P\} \cup \{R_t\}) + d(P, R_t)$ is minimal, where P denotes the point where the server is actually located.

Example 9.2 Consider the metric space containing three points A, B and C with the distances $d(A,B) = 1$, $d(B,C) = 2$, $d(A,C) = 3$. Suppose that we have two servers and the starting configuration is $\{A, B\}$. In this case the starting work function values are $w_0(\{A,A\}) = 1$, $w_0(\{A,B\}) = 0$, $w_0(\{A,C\}) = 2$, $w_0(\{B,B\}) = 1$, $w_0(\{B,C\}) = 3$, $w_0(\{C,C\}) = 5$. Suppose that the first request appears at point C. Then $w_0(\{A,B\} \setminus \{A\} \cup \{C\}) + d(A,C) = 3+3 = 6$ and $w_0(\{A,B\} \setminus \{B\} \cup \{C\}) + d(B,C) = 2+2 = 4$, thus algorithm WORK FUNCTION uses the server from point B to serve the request.

The following statement is valid for the algorithm.

Theorem 9.3 *The* WORK-FUNCTION *algorithm is weakly* $2k-1$*-competitive.*

Besides the general problem many particular cases have been investigated. If the distance of any pair of points is 1, then we obtain the on-line paging problem as a special case. Another well investigated metric space is the line. The points of the line are considered as real numbers, and the distance of points a and b is $|a-b|$. In this special case a k-competitive algorithm was developed by Chrobak and Larmore, which algorithm is called DOUBLE-COVERAGE. A request at point P is served by server s which is the closest to P. Moreover, if there are servers also on the opposite side of P, then the closest server among them moves distance $d(s,P)$ into the direction of P. Hereafter we denote the DOUBLE-COVERAGE algorithm by DC. The input of the algorithm is the list of requests which is a list of points (real numbers) denoted by $I = P_1, \ldots, P_n$ and the starting configuration of the servers is denoted by $S = (s_1, \ldots, s_k)$ which contains points (real numbers) too. The algorithm can be defined by the following pseudocode:

DC(I, S)

```
1  for j ← 1 to n
2      do i ← argmin_l d(P_j, s_l)
3  if s_i = min_l s_l or s_i = max_l s_l
4     then the request is served by the i-th server
5          s_i ← P_j
6     else if s_i ≤ P_j
7             then m ← argmin_{l:s_l>P_j} d(s_l, P_j)
8                  the request is served by the i-th server
9                  s_m ← s_m − d(s_i, P_j)
10                 s_i ← P_j
11            else if s_i ≥ P_j
12                    then r ← argmin_{l:s_l<P_j} d(s_l, P_j)
13                    the request is served by the i-th server
14                    s_r ← s_r + d(s_i, P_j)
15                    s_i ← P_j
```

Example 9.3 Suppose that there are three servers s_1, s_2, s_3 located at points $0, 1, 2$. If the next request appears at point 4, then DC uses the closest server s_3 to serve the request. The locations of the other servers remain unchanged, the cost of serving the request is 2 and the servers are at points $0, 1, 4$. If the next request appears at point 2, then DC uses the closest server s_2 to serve the request, but there are servers on the opposite side of the request, thus s_3 also travels distance 1 into the direction of 2. Therefore the cost of serving the request is 2 and the servers will be at points $0, 2, 3$.

The following statement, which can be proved by the potential function technique, is valid for algorithm DC. This technique is often used in the analysis of on-line algorithms.

Theorem 9.4 *Algorithm* DC *is weakly k-competitive on the line.*

Proof Consider an arbitrary sequence of requests and denote this input by I. During the analysis of the procedure we suppose that one off-line optimal algorithm and DC are running parallel on the input. We also suppose that each request is served first by the off-line algorithm and then by the on-line algorithm. The servers of the on-line algorithm and also the positions of the servers (which are real numbers) are denoted by $s_1, \ldots, s_k$, and the servers of the optimal off-line algorithm and also the positions of the servers are denoted by $x_1, \ldots, x_k$. We suppose that for the positions $s_1 \leq s_2 \leq \cdots \leq s_k$ and $x_1 \leq x_2 \leq \cdots \leq x_k$ are always valid, this can be achieved by swapping the notations of the servers.

We prove the theorem by the potential function technique. The potential function assigns a value to the actual positions of the servers, so the on-line and off-line costs are compared using the changes of the potential function. Let us define the following potential function:

$$\Phi = k \sum_{i=1}^{k} |x_i - s_i| + \sum_{i<j} (s_j - s_i) \ .$$

The following statements are valid for the potential function.

- While OPT is serving a request the increase of the potential function is not more than k times the distance travelled by the servers of OPT.
- While DC is serving a request, the decrease of Φ is at least as much as the cost of serving the request.

If the above properties are valid, then one can prove the theorem easily. In this case $\Phi_f - \Phi_0 \leq k \cdot \mathrm{OPT}(I) - \mathrm{DC}(I)$, where Φ_f and Φ_0 are the final and the starting values of the potential function. Furthermore, Φ is nonnegative, so we obtain that $\mathrm{DC}(I) \leq k\mathrm{OPT}(I) + \Phi_0$, which yields that the algorithms is weakly k-competitive (Φ_0 does not depend on the input sequence only on the starting position of the servers).

Now we prove the properties of the potential function.

First consider the case when one of the off-line servers travels distance d. The first part of the potential function increases at most by kd. The second part does not change, thus we proved the first property of the potential function.

Consider the servers of DC. Suppose that the request appears at point P. Since the request is first served by OPT, $x_j = P$ for some j. The following two cases are distinguished depending on the positions of the on-line servers.

First suppose that the on-line servers are on the same side of P. We can assume that the positions of the servers are not smaller than P, since the other case is completely similar. In this case s_1 is the closest server and DC sends s_1 to P and the other on-line servers do not move. Therefore the cost of DC is $d(s_1, P)$. In the first sum of the potential function only $|x_1 - s_1|$ changes; it decreases by $d(s_1, P)$, thus the first part decreases by $kd(s_1, P)$. The second sum is increasing; the increase is $(k-1)d(s_1, P)$, thus the value of Φ decreases by $d(s_1, P)$.

Assume that there are servers on both sides of P; suppose that the closest servers are s_i and s_{i+1}. We assume that s_i is closer to P, the other case is completely similar. In this case the cost of DC is $2d(s_i, P)$. Consider the first sum of the potential function. The i-th and the $i+1$-th part are changing. Since $x_j = P$ for some j, thus one of the i-th and the $i+1$-th parts decreases by $d(s_i, P)$ and the increase of the other one is at most $d(s_i, P)$, thus the first sum does not increase. The change of the second sum of Φ is

$$d(s_i, P)\big(-(k-i) + (i-1) - (i) + (k-(i+1))\big) = -2d(s_i, P) \ .$$

Thus we proved that the second property of the potential function is also valid in this case. ■

Exercises

9.2-1 Suppose that (M, d) is a metric space. Prove that (M, q) is also a metric space where $q(x, y) = \min\{1, d(x, y)\}$.

9.2-2 Consider the greedy algorithm which serves each request by the server which is closest to the place of the request. Prove that the algorithm is not constant competitive for the line.

9.2-3 Prove that for arbitrary k-element multisets X and Z and for arbitrary t the inequality $w_t(Z) \leq w_t(X) + d(X, Z)$ is valid, where $d(X, Z)$ is the cost of the

minimal matching of X and Z, (the minimal cost needed to move the servers from configuration X to configuration Z).

9.2-4 Consider the line as a metric space. Suppose that the servers of the on-line algorithm are at points $2, 4, 5, 7$, and the servers of the off-line algorithm are at points $1, 3, 6, 9$. Calculate the value of the potential function used in the proof of Theorem 9.4. How does this potential function change, if the on-line server moves from point 7 to point 8?

9.3. Models related to computer networks

The theory of computer networks has become one of the most significant areas of computer science. In the planning of computer networks many optimisation problems arise and most of these problems are actually on-line, since neither the traffic nor the changes in the topology of a computer network cannot be precisely predicted. Recently some researchers working at the area of on-line algorithms have defined some on-line mathematical models for problems related to computer networks. In this section we consider this area; we present three problems and show the basic results. First the data acknowledgement problem is considered, then we present the web caching problem, and the section is closed by the on-line routing problem.

9.3.1. The data acknowledgement problem

In the communication of a computer network the information is sent by packets. If the communication channel is not completely safe, then the arrival of the packets are acknowledged. The data acknowledgement problem is to determine the time of sending acknowledgements. An acknowledgement can acknowledge many packets but waiting for long time can cause the resending of the packets and that results in the congestion of the network. On the other hand, sending an acknowledgement about the arrival of each packet immediately would cause again the congestion of the network. The first optimisation model for determining the sending times of the acknowledgements was developed by Dooly, Goldman and Scott in 1998. We present the developed model and some of the basic results.

In the mathematical model of the data acknowledgement problem the input is the list of the arrival times $a_1, \ldots, a_n$ of the packets. The decision maker has to determine when to send acknowledgements; these times are denoted by $t_1, \ldots, t_k$. In the optimisation model the cost function is:

$$k + \sum_{j=1}^{k} \nu_j \ ,$$

where k is the number of the sent acknowledgements and $\nu_j = \sum_{t_{j-1} < a_i \leq t_j} (t_j - a_i)$ is the total latency collected by the j-th acknowledgement. We consider the on-line problem which means that at time t the decision maker only knows the arrival times of the packets already arrived and has no information about the further packets. We denote the set of the unacknowledged packets at the arrival time a_i by σ_i.

For the solution of the problem the class of the alarming algorithms has been developed. An ***alarming algorithm*** works as follows. At the arrival time a_j an alarm is set for time $a_j + e_j$. If no packet arrives before time $a_j + e_j$, then an acknowledgement is sent at time $a_j + e_j$ which acknowledges all of the unacknowledged packets. Otherwise at the arrival of the next packet at time a_{j+1} the alarm is reset for time $a_{j+1} + e_{j+1}$. Below we analyse an algorithm from this class in details. This algorithm sets the alarm to collect total latency 1 by the acknowledgement. The algorithm is called ALARM. We obtain the above defined rule from the general definition using the solution of the following equation as value e_j:

$$1 = |\sigma_j| e_j + \sum_{a_i \in \sigma_j} (a_j - a_i) \ .$$

Example 9.4 Consider the following example. The first packet arrives at time 0 ($a_1 = 0$), so ALARM sets an alarm with value $e_1 = 1$ for time 1. Suppose that the next arrival time is $a_2 = 1/2$. This arrival is before the alarm time, thus the first packet has not been acknowledged yet and we reset the alarm with value $e_2 = (1 - 1/2)/2 = 1/4$ for time $1/2 + 1/4$. Suppose that the next arrival time is $a_3 = 5/8$. This arrival is before the alarm time, thus the first two packets have not been acknowledged yet and we reset the alarm with value $e_3 = (1 - 5/8 - 1/8)/3 = 1/12$ for time $5/8 + 1/12$. Suppose that the next arrival time is $a_4 = 1$. No packet arrived before the alarm time $5/8 + 1/12$, thus at that time the first three packets were acknowledged and the alarm is set for the new packet with value $e_4 = 1$ for time 2.

Theorem 9.5 *Algorithm* ALARM *is 2-competitive.*

Proof Suppose that algorithm ALARM sends k acknowledgements. These acknowledgements divide the time into k time intervals. The cost of the algorithm is $2k$, since k is the cost of the acknowledgements, and the alarm is set to have total latency 1 for each acknowledgement.

Suppose that the optimal off-line algorithm sends k^* acknowledgements. If $k^* \geq k$, then $\mathrm{OPT}(I) \geq k = \mathrm{ALARM}(I)/2$ is obviously valid, thus we obtain that the algorithm is 2-competitive. If $k^* < k$, then at least $k - k^*$ time intervals among the ones defined by the acknowledgements of algorithm ALARM do not contain any of the off-line acknowledgements. This yields that the off-line total latency is at most $k - k^*$, thus we obtain that $\mathrm{OPT}(I) \geq k$ which inequality proves that ALARM is 2-competitive. ■

As the following theorem shows, algorithm ALARM has the smallest possible competitive ratio.

Theorem 9.6 *There is no on-line algorithm for the data acknowledgement problem which has smaller competitive ratio than* 2.

Proof Consider an arbitrary on-line algorithm and denote it by ONL. Analyse the following input. Consider a long sequence of packets where the packets always arrive immediately after the time when ONL sends an acknowledgement. The on-line cost of a sequence containing $2n$ packets is $\mathrm{ONL}(I_{2n}) = 2n + t_{2n}$, since the cost resulted

from the acknowledgements is $2n$, and the latency of the i-th acknowledgement is $t_i - t_{i-1}$, where the value $t_0 = 0$ is used.

Consider the following two on-line algorithms. ODD sends the acknowledgements after the odd numbered packets and EVEN sends the acknowledgements after the even numbered packets.

The costs achieved by these algorithms are

$$\mathrm{EVEN}(I_{2n}) = n + \sum_{i=0}^{n-1} (t_{2i+1} - t_{2i}) \ ,$$

and

$$\mathrm{ODD} = n + 1 + \sum_{i=1}^{n} (t_{2i} - t_{2i-1}) \ .$$

Therefore $\mathrm{EVEN}(I_{2n}) + \mathrm{ODD}(I_{2n}) = \mathrm{ONL}(I_{2n}) + 1$. On the other hand, none of the costs achieved by ODD and EVEN is greater than the optimal off-line cost, thus $\mathrm{OPT}(I_{2n}) \leq \min\{\mathrm{EVEN}(I_{2n}), \mathrm{ODD}(I_{2n})\}$, which yields that $\mathrm{ONL}(I_{2n})/\mathrm{OPT}(I_{2n}) \geq 2 - 1/\mathrm{OPT}(I_{2n})$. From this inequality it follows that the competitive ratio of ONL is not smaller than 2, because using a sufficiently long sequence of packets the value $\mathrm{OPT}(I_{2n})$ can be arbitrarily large. ■

9.3.2. The file caching problem

The file caching problem is a generalisation of the caching problem presented in the chapter on memory management. World-wide-web browsers use caches to store some files. This makes it possible to use the stored files if a user wants to see some web-page many times during a short time interval. If the cache becomes full, then some files must be eliminated to make space for the new file. The file caching problem models this scenario; the goal is to find good strategies for determining which files should be eliminated. It differs from the standard paging problem in the fact that the files have size and retrieval cost (the problem is reduced to the paging if each size and each retrieval cost are 1). So the following mathematical model describes the problem.

There is a given cache of size k and the input is a sequence of pages. Each page p has a ***size*** denoted by $s(p)$ and a ***retrieval cost*** denoted by $c(p)$. The pages arrive from a list one by one and after the arrival of a page the algorithm has to place it into the cache. If the page is not contained in the cache and there is not enough space to put it into the cache, then the algorithm has to delete some pages from the cache to make enough space for the requested page. If the required page is in the cache, then the cost of serving the request is 0, otherwise the cost is $c(p)$. The aim is to minimise the total cost. The problem is on-line which means that for the decisions (which pages should be deleted from the cache) only the earlier pages and decisions can be used, the algorithm has no information about the further pages. We assume that the size of the cache and also the sizes of the pages are positive integers.

For the solution of the problem and for its special cases many algorithms have been developed. Here we present algorithm LANDLORD which was developed by Young.

The algorithm stores a credit value $0 \leq cr(f) \leq c(f)$ for each page f which is contained in the current cache. In the rest of the section the set of the pages in the current cache of LANDLORD is denoted by LA. If LANDLORD has to retrieve a page g then the following steps are performed.

LANDLORD(LA, g)

```
1  if g is not contained in LA
2     then while there is not enough space for g
3                Δ ← min_{f∈LA} cr(f)/s(f)
4                for each f ∈ LA let cr(f) ← cr(f) − Δ · s(f)
5                evict some pages with cr(f) = 0
6                place g into cache LA and let cr(g) ← c(g)
7     else  reset cr(g) to any value between cr(g) and c(g)
```

Example 9.5 Suppose that $k = 10$ and LA contains the following three pages: g_1 with $s(g_1) = 2, cr(g_1) = 1$, g_2 with $s(g_2) = 4, cr(g_2) = 3$ and g_3 with $s(g_3) = 3, cr(g_3) = 3$. Suppose that the next requested page is g_4, with parameters $s(g_4) = 4$ and $c(g_4) = 4$. Therefore, there is not enough space for it in the cache, so some pages must be evicted. LANDLORD determines the value $\Delta = 1/2$ and changes the credits as follows: $cr(g_1) = 0, cr(g_2) = 1$ and $cr(g_3) = 3/2$, thus g_1 is evicted from cache LA. There is still not enough space for g_4 in the cache. The new Δ value is $\Delta = 1/4$ and the new credits are: $cr(g_2) = 0, cr(g_3) = 3/4$, thus g_2 is evicted from the cache. Then there is enough space for g_4, thus it is placed into cache LA with the credit value $cr(g_4) = 4$.

LANDLORD is weakly k-competitive, but a stronger statement is also true. For the web caching problem an on-line algorithm ALG is called (C, k, h)-competitive, if there exists such a constant B, that $\mathrm{ALG}_k(I) \leq C \cdot \mathrm{OPT}_h(I) + B$ is valid for each input, where $\mathrm{ALG}_k(I)$ is the cost of ALG using a cache of size k and $\mathrm{OPT}_h(I)$ is the optimal off-line cost using a cache of size h. The following statement holds for algorithm LANDLORD.

Theorem 9.7 *If $h \leq k$, then algorithm* LANDLORD *is $(k/(k-h+1), k, h)$-competitive.*

Proof Consider an arbitrary input sequence of pages and denote the input by I. We use the potential function technique. During the analysis of the procedure we suppose that an off-line optimal algorithm with cache size h and LANDLORD with cache size k are running parallel on the input. We also suppose that each page is placed first into the off-line cache by the off-line algorithm and then it is placed into LA by the on-line algorithm. We denote the set of the pages contained in the actual cache of the optimal off-line algorithm by OPT. Consider the following potential function:

$$\Phi = (h-1) \sum_{f \in LA} cr(f) + k \sum_{f \in OPT} (c(f) - cr(f)) \ .$$

The changes of the potential function during the different steps are as follows.

- OPT places g into its cache.

  In this case OPT has cost $c(g)$. In the potential function only the second part may change. On the other hand, $cr(g) \geq 0$, thus the increase of the potential function is at most $k \cdot c(g)$.

- LANDLORD decreases the credit value for each $f \in LA$.

  In this case for each $f \in LA$ the decrease of $cr(f)$ is $\Delta \cdot s(f)$, thus the decrease of Φ is

  $$\Delta((h-1)s(LA) - ks(OPT \cap LA)) \ ,$$

  where $s(LA)$ and $s(OPT \cap LA)$ denote the total size of the pages contained in sets LA and $OPT \cap LA$, respectively. At the time when this step is performed, OPT have already placed page g into its cache OPT, but the page is not contained in cache LA. Therefore $s(OPT \cap LA) \leq h - s(g)$. On the other hand, this step is performed if there is not enough space for the page in LA thus $s(LA) > k - s(g)$, which yields $s(LA) \geq k - s(g) + 1$, because the sizes are positive integers. Therefore we obtain that the decrease of Φ is at least

  $$\Delta\big((h-1)(k - s(g) + 1) - k(h - s(g))\big) \ .$$

  Since $s(g) \geq 1$ and $k \geq h$, this decrease is at least $\Delta((h-1)(k-1+1)-k(h-1)) = 0$.

- LANDLORD evicts a page f from cache LA.

  Since LANDLORD only evicts pages having credit 0, during this step Φ remains unchanged.

- LANDLORD places page g into cache LA and sets the value $cr(g) = c(g)$.

  The cost of LANDLORD is $c(g)$. On the other hand, g was not contained in cache LA before the performance of this step, thus $cr(g) = 0$ was valid. Furthermore, first OPT places the page into its cache, thus $g \in OPT$ is also valid. Therefore the decrease of Φ is $-(h-1)c(g) + kc(g) = (k - h + 1)c(g)$.

- LANDLORD resets the credit of a page $g \in LA$ to a value between $cr(g)$ and $c(g)$.

  In this case $g \in OPT$ is valid, since OPT places page g into its cache first. Value $cr(g)$ is not decreased and $k > h - 1$, thus Φ can not increase during this step.

Hence the potential function has the following properties..

- If OPT places a page into its cache, then the increase of the potential function is at most k times more than the cost of OPT.
- If LANDLORD places a page into its cache, then the decrease of Φ is $(k - h + 1)$ times more than the cost of LANDLORD.
- During the other steps Φ does not increase.

By the above properties we obtain that $\Phi_f - \Phi_0 \leq k \cdot \mathrm{OPT}_h(I) - (k - h + 1) \cdot \mathrm{LANDLORD}_k(I)$, where Φ_0 and Φ_f are the starting and final values of the potential

function. The potential function is nonnegative, thus we obtain that $(k-h+1)\cdot \textsc{Landlord}_k(I) \leq k \cdot \mathrm{OPT}_h(I) + \Phi_0$, which proves that Landlord is $(k/(k-h+1), k, h)$-competitive. ■

9.3.3. On-line routing

In computer networks the congestion of the communication channels decreases the speed of the communication and may cause loss of information. Thus congestion control is one of the most important problems in the area of computer networks. A related important problem is the routing of the communication, where we have to determine the path of the messages in the network. Since we have no information about the further traffic of the network, thus routing is an on-line problem. Here we present two on-line optimisation models for the routing problem.

The mathematical model

The network is given by a graph, each edge e has a maximal available bandwidth denoted by $u(e)$ and the number of edges is denoted by m. The input is a sequence of requests, where the j-th request is given by a vector $(s_j, t_j, r_j, d_j, b_j)$ which means that to satisfy the request bandwidth r_j must be reserved on a path from s_j to t_j for time duration d_j and the benefit of serving a request is b_j. Hereafter, we assume that $d_j = \infty$, and we omit the value of d_j from the requests. The problem is on-line which means that after the arrival of a request the algorithm has to make the decisions without any information about the further requests. We consider the following two models.

Load balancing model: In this model all requests must be satisfied. Our aim is to minimise the maximum of the overload of the edges. The overload is the ratio of the total bandwidth assigned to the edge and the available bandwidth. Since each request is served, thus the benefit is not significant in this model.

Throughput model: In this model the decision maker is allowed to reject some requests. The sum of the bandwidths reserved on an edge can not be more than the available bandwidth. The goal is to maximise the sum of the benefits of the accepted requests. We investigate this model in details. It is important to note that this is a maximisation problem thus the notion of competitiveness is used in the form defined for maximisation problems.

Below we define the exponential algorithm. We need the following notations to define and analyse the algorithm. Let P_i denote the path which is assigned to the accepted request i. Let A denote the set of requests accepted by the on-line algorithm. In this case $l_e(j) = \sum_{i \in A, i<j, e \in P_i} r_i/u(e)$ is the ratio of the total reserved bandwidth and the available bandwidth on e before the arrival of request j.

The basic idea of the exponential algorithm is the following. The algorithm assigns a cost to each e, which is exponential in $l_e(j)$ and chooses the path which has the minimal cost. Below we define and analyse the exponential algorithm for the throughput model. Let μ be a constant which depends on the parameters of the problem; its value will be given later. Let $c_e(j) = \mu^{l_e(j)}$, for each request j and edge e. The exponential algorithm performs the following steps after the arrival of a request (s_j, t_j, r_j, b_j).

EXP(s_j, t_j, r_j, b_j)

1 let U_j be the set of the paths (s_j, t_j)
2 $P_j \leftarrow \text{argmin}_{P \in U_j}\{\sum_{e \in P} \frac{r_j}{u(e)} c_e(j)\}$
3 **if** $C(P_j) = \sum_{e \in P_j} \frac{r_j}{u(e)} c_e(j) \leq 2mb_j$
4 **then** reserve bandwidth r_j on path P_j
5 **else** reject the request

Note. If we modify this algorithm to accept each request, then we obtain an exponential algorithm for the load balancing model.

Example 9.6 Consider the network which contains vertices A, B, C, D and edges (A, B), (B, D), (A, C), (C, D), where the available bandwidths of the edges are $u(A, B) = 1$, $u(B, D) = 3/2$, $u(A, C) = 2$, $u(C, D) = 3/2$. Suppose that $\mu = 10$ and that the reserved bandwidths are: 3/4 on path (A, B, D), 5/4 on path (A, C, D), 1/2 on path (B, D), 1/2 on path (A, C). The next request j is to reserve bandwidth 1/8 on some path between A and D. Therefore values $l_e(j)$ are: $l_{(A,B)}(j) = (3/4) : 1 = 3/4$, $l_{(B,D)}(j) = (3/4+1/2) : (3/2) = 5/6$, $l_{(A,C)}(j) = (5/4 + 1/2) : 2 = 7/8$, $l_{(C,D)}(j) = (5/4) : (3/2) = 5/6$. There are two paths between A and D and the costs are:

$$C(A, B, D) = 1/8 \cdot 10^{3/4} + 1/12 \cdot 10^{5/6} = 1.269 \ ,$$

$$C(A, C, D) = 1/16 \cdot 10^{7/8} + 1/12 \cdot 10^{5/6} = 1.035 \ .$$

The minimal cost belongs to path (A, C, D). Therefore, if $2mb_j = 8b_j \geq 1,035$, then the request is accepted and the bandwidth is reserved on path (A, C, D). Otherwise the request is rejected.

To analyse the algorithm consider an arbitrary input sequence I. Let A denote the set of the requests accepted by EXP, and A^* the set of the requests which are accepted by OPT and rejected by EXP. Furthermore let $P_j{}^*$ denote the path reserved by OPT for each request j accepted by OPT. Define the value $l_e(v) = \sum_{i \in A, e \in P_i} r_i/u(e)$ for each e, which value gives the ratio of the reserved bandwidth and the available bandwidth for e at the end of the on-line algorithm. Furthermore, let $c_e(v) = \mu^{l_e(v)}$ for each e.

Let $\mu = 4mPB$, where B is an upper bound on the benefits and for each request and each edge the inequality

$$\frac{1}{P} \leq \frac{r(j)}{u(e)} \leq \frac{1}{\lg \mu}$$

is valid. In this case the following statements hold.

Lemma 9.8 *The solution given by algorithm* EXP *is feasible, i.e. the sum of the reserved bandwidths is not more than the available bandwidth for each edge.*

Proof We prove the statement by contradiction. Suppose that there is an edge f where the available bandwidth is violated. Let j be the first accepted request which violates the available bandwidth on f.

The inequality $r_j/u(f) \leq 1/\lg \mu$ is valid for j and f (it is valid for all edges and

requests). Furthermore, after the acceptance of request j the sum of the bandwidths is greater than the available bandwidth on edge f, thus we obtain that $l_f(j) > 1 - 1/\lg\mu$. On the other hand, this yields that the inequality

$$C(P_j) = \sum_{e\in P_j} \frac{r_j}{u(e)} c_e(j) \geq \frac{r_j}{u(f)} c_f(j) > \frac{r_j}{u(f)} \mu^{1-1/\lg\mu}$$

holds for value $C(P_j)$ used in algorithm EXP. Using the assumption on P we obtain that $\frac{r_j}{u(e)} \geq \frac{1}{P}$, and $\mu^{1-1/\lg\mu} = \mu/2$, thus from the above inequality we obtain that

$$C(P) > \frac{1}{P}\frac{\mu}{2} = 2mB \ .$$

On the other hand, this inequality is a contradiction, since EXP would reject the request. Therefore we obtained a contradiction thus we proved the statement of the lemma. ∎

Lemma 9.9 *For the solution given by* OPT *the following inequality holds:*

$$\sum_{j\in A^*} b_j \leq \frac{1}{2m} \sum_{e\in E} c_e(v) \ .$$

Proof Since EXP rejected each $j \in A^*$, thus $b_j < \frac{1}{2m} \sum_{e\in P_j{}^*} \frac{r_j}{u(e)} c_e(j)$ for each $j \in A^*$, and this inequality is valid for all paths between s_j and t_j. Therefore

$$\sum_{j\in A^*} b_j < \frac{1}{2m} \sum_{j\in A^*} \sum_{e\in P_j{}^*} \frac{r_j}{u(e)} c_e(j) \ .$$

On the other hand, $c_e(j) \leq c_e(v)$ holds for each e, thus we obtain that

$$\sum_{j\in A^*} b_j < \frac{1}{2m} \sum_{e\in E} c_e(v) \Big(\sum_{j\in A^*: e\in P_j{}^*} \frac{r_j}{u(e)} \Big) \ .$$

The sum of the bandwidths reserved by OPT is at most the available bandwidth $u(e)$ for each e, thus $\sum_{j\in A^*: e\in P^*(j)} \frac{r_j}{u(e)} \leq 1$.

Consequently,

$$\sum_{j\in A^*} b_j \leq \frac{1}{2m} \sum_{e\in E} c_e(v) \ ,$$

which inequality is the one which we wanted to prove. ∎

Lemma 9.10 *For the solution given by algorithm* EXP *the following inequality holds:*

$$\frac{1}{2m} \sum_{e\in E} c_e(v) \leq (1 + \lg\mu) \sum_{j\in A} b_j \ .$$

Proof It is enough to show that the inequality $\sum_{e\in P_j} (c_e(j+1) - c_e(j)) \leq 2mb_j \log_2\mu$ is valid for each request $j \in A$. On the other hand,

$$c_e(j+1) - c_e(j) = \mu^{l_e(j) + \frac{r_j}{u(e)}} - \mu^{l_e(j)} = \mu^{l_e(j)} (2^{\log_2\mu \frac{r_j}{u(e)}} - 1) \ .$$

Since $2^x - 1 < x$, if $0 \leq x \leq 1$, and because of the assumptions $0 \leq \log_2 \mu \frac{r_j}{u(e)} \leq 1$, we obtain that

$$c_e(j+1) - c_e(j) \leq \mu^{l_e(j)} \log_2 \mu \frac{r_j}{u(e)}.$$

Summarising the bounds given above we obtain that

$$\sum_{e \in P_j} (c_e(j+1) - c_e(j)) \leq \log_2 \mu \sum_{e \in P_j} \mu^{l_e(j)} \frac{r_j}{u(e)} = \log_2 \mu \cdot C(P_j) \ .$$

Since EXP accepts the requests with the property $C(P_j) \leq 2mb_j$, the above inequality proves the required statement. ■

With the help of the above lemmas we can prove the following theorem.

Theorem 9.11 *Algorithm* EXP *is* $\Omega(1/\lg \mu)$*-competitive, if* $\mu = 4mPB$*, where* B *is an upper bound on the benefits, and for all edges and requests*

$$\frac{1}{P} \leq \frac{r(j)}{u(e)} \leq \frac{1}{\lg \mu} \ .$$

Proof From Lemma 9.8 it follows that the algorithm results in a feasible solution where the available bandwidths are not violated. Using the notations defined above we obtain that the benefit of algorithm EXP on the input I is $\mathrm{EXP}(I) = \sum_{j \in A} b_j$, and the benefit of OPT is at most $\sum_{j \in A \cup A^*} b_j$. Therefore by Lemma 9.9 and Lemma 9.10 it follows that

$$\mathrm{OPT}(I) \leq \sum_{j \in A \cup A^*} b_j \leq (2 + \log_2 \mu) \sum_{j \in A} b_j \leq (2 + \log_2 \mu) \mathrm{EXP}(I) \ ,$$

which inequality proves the theorem. ■

Exercises

9.3-1 Consider the modified version of the data acknowledgement problem with the objective function $k + \sum_{j=1}^{k} \mu_j$, where k is the number of acknowledgements and $\mu_j = \max_{t_{j-1} < a_i \leq t_j} \{t_j - a_i\}$ is the maximal latency of the j-th acknowledgement. Prove that algorithm ALARM is also 2-competitive in this modified model.

9.3-2 Represent the special case of the web caching problem, where $s(g) = c(g) = 1$ for each page g as a special case of the k-server problem. Define the metric space which can be used.

9.3-3 In the web caching problem cache LA of size 8 contains three pages a, b, c with the following sizes and credits: $s(a) = 3, s(b) = 2, s(c) = 3, cr(a) = 2, cr(b) = 1/2, cr(c) = 2$. We want to retrieve a page d of size 3 and retrieval cost 4. The optimal off-line algorithm OPT with cache of size 6 already placed the page into its cache, so its cache contains the pages d and c. Which pages are evicted by LANDLORD to place d? In what way does the potential function defined in the proof of Theorem 9.7 change?

9.3-4 Prove that if in the throughput model no bounds are given for the ratios $r(j)/u(e)$, then there is no constant-competitive on-line algorithm.

9.4. On-line bin packing models

In this section we consider the on-line bin packing problem and its multidimensional generalisations. First we present some fundamental results of the area. Then we define the multidimensional generalisations and present some details from the area of on-line strip packing.

9.4.1. On-line bin packing

In the bin packing problem the input is a list of items, where the i-th item is given by its size $a_i \in (0, 1]$. The goal is to pack the items into unit size bins and minimise the number of the bins used. In a more formal way we can say that we have to divide the items into groups where each group has the property that the total size of its items is at most 1, and the goal is to minimise the number of groups. This problem appears also in the area of memory management.

In this section we investigate the on-line problem which means that the decision maker has to make decisions about the packing of the i-th item based on values $a_1, \ldots, a_i$ without any information about the further items.

Algorithm Next-Fit, bounded space algorithms

First we consider the model where the number of the open bins is limited. The k-bounded space model means that if the number of open bins reaches bound k, then the algorithm can open a new bin only after closing some of the bins, and the closed bins cannot be used for packing further items into them. If only one bin can be open, then the evident algorithm packs the item into the open bin if it fits, otherwise it closes the bin, opens a new one and puts the item into it. This algorithm is called Next-Fit (NF) algorithm. We do not present the pseudocode of the algorithm, since it can be found in this book in the chapter about memory management. The asymptotic competitive ratio of algorithm NF is determined by the following theorem.

Theorem 9.12 *The asymptotic competitive ratio of* NF *is 2.*

Proof Consider an arbitrary sequence of items, denote it by σ. Let n denote the number of bins used by OPT and m the number of bins used by NF. Furthermore, let S_i, $i = 1, \ldots, m$ denote the total size of the items packed into the i-th bin by NF.

Then $S_i + S_{i+1} > 1$, since in the opposite case the first item of the $(i+1)$-th bin fits into the i-th bin which contradicts to the definition of the algorithm. Therefore the total size of the items is more than $\lfloor m/2 \rfloor$.

On the other hand the optimal off-line algorithm cannot put items with total size more than 1 into the same bin, thus we obtain that $n > \lfloor m/2 \rfloor$. This yields that $m \leq 2n - 1$, thus

$$\frac{\mathrm{NF}(\sigma)}{\mathrm{OPT}(\sigma)} \leq \frac{2n-1}{n} = 2 - 1/n \; .$$

Consequently, we proved that the algorithm is asymptotically 2-competitive.

Now we prove that the bound is tight. Consider the following sequence for each n denoted by σ_n. The sequence contains $4n-2$ items, the size of the $2i-1$-th item is $1/2$, the size of the $2i$-th item is $1/(4n-2)$, $i = 1, \ldots, 2n-1$. Algorithm NF puts the $(2i-1)$-th and the $2i$-th items into the i-th bin for each bin, thus $\mathrm{NF}(\sigma_n) = 2n-1$. The optimal off-line algorithm puts pairs of $1/2$ size items into the first $n-1$ bins and it puts one $1/2$ size item and the small items into the n-th bin, thus $\mathrm{OPT}(\sigma_n) = n$. Since $\mathrm{NF}(\sigma_n)/\mathrm{OPT}(\sigma_n) = 2 - 1/n$ and this function tends to 2 as n tends to ∞, we proved that the asymptotic competitive ratio of the algorithm is at least 2. ■

If $k > 1$, then there are better algorithms than NF for the k-bounded space model. The best known bounded space on-line algorithms belong to the family of ***harmonic algorithms***, where the basic idea is that the interval $(0, 1]$ is divided into subintervals and each item has a type which is the subinterval of its size. The items of the different types are packed into different bins. The algorithm runs several NF algorithms simultaneously; each for the items of a certain type.

Algorithm First-Fit and the weight function technique

In this section we present the weight function technique which is often used in the analysis of the bin packing algorithms. We show this method by analysing algorithm First-Fit (FF).

Algorithm FF can be used when the number of open bins is not bounded. The algorithm puts the item into the first opened bin where it fits. If the item does not fit into any of the bins, then a new bin is opened and the algorithm puts the item into it. The pseudocode of the algorithm is also presented in the chapter of memory management. The asymptotic competitive ratio of the algorithm is bounded above by the following theorem.

Theorem 9.13 FF *is asymptotically 1.7-competitive.*

Proof In the proof we use the weight function technique whose idea is that a weight is assigned to each item to measure in some way how difficult it can be to pack the certain item. The weight function and the total size of the items are used to bound the off-line and on-line objective function values. We use the following weight function:

$$w(x) = \begin{cases} 6x/5, & \text{if } 0 \le x \le 1/6 \\ 9x/5 - 1/10, & \text{if } 1/6 \le x \le 1/3 \\ 6x/5 + 1/10, & \text{if } 1/3 \le x \le 1/2 \\ 6x/5 + 2/5, & \text{if } 1/2 < x\ . \end{cases}$$

Let $w(H) = \sum_{i \in H} w(a_i)$ for any set H of items. The properties of the weight function are summarised in the following two lemmas. Both lemmas can be proven by case disjunction based on the sizes of the possible items. The proofs are long and contain many technical details, therefore we omit them.

Lemma 9.14 *If* $\sum_{i \in H} a_i \le 1$ *is valid for a set* H *of items, then* $w(H) \le 17/10$ *also holds.*

Lemma 9.15 *For an arbitrary list* L *of items* $w(L) \ge \mathrm{FF}(L) - 2$.

Using these lemmas we can prove that the algorithm is asymptotically 1.7-competitive. Consider an arbitrary list L of items. The optimal off-line algorithm can pack the items of the list into $\mathrm{OPT}(L)$ bins. The algorithm packs items with total size at most 1 into each bin, thus from Lemma 9.14 it follows that $w(L) \leq 1.7\mathrm{OPT}(L)$. On the other hand considering Lemma 9.15 we obtain that $\mathrm{FF}(L)-2 \leq w(L)$, which yields that $FF(L) \leq 1.7\mathrm{OPT}(L)+2$, and this inequality proves that the algorithm is asymptotically 1.7-competitive. ■

It is important to note that the bound is tight, i.e. it is also true that the asymptotic competitive ratio of FF is 1.7. Many algorithms have been developed with smaller asymptotic competitive ratio than 17/10, the best algorithm known at present time is asymptotically 1.5888-competitive.

Lower bounds

In this part we consider the techniques for proving lower bounds on the possible asymptotic competitive ratio. First we present a simple lower bound and then we show how the idea of the proof can be extended into a general method.

Theorem 9.16 *No on-line algorithm for the bin packing problem can have smaller asymptotic competitive ratio than 4/3.*

Proof Let A be an arbitrary on-line algorithm. Consider the following sequence of items. Let $\varepsilon < 1/12$ and L_1 be a list of n items of size $1/3+\varepsilon$, and L_2 be a list of n items of size $1/2+\varepsilon$. The input is started by L_1. Then A packs two items or one item into the bins. Denote the number of bins containing two items by k. In this case the number of the used bins is $A(L_1) = k+n-2k = n-k$. On the other hand, the optimal off-line algorithm can pack pairs of items into the bins, thus $\mathrm{OPT}(L_1) = \lceil n/2 \rceil$.

Now suppose that the input is the combined list L_1L_2. The algorithm is an on-line algorithm, therefore it does not know whether the input is L_1 or L_1L_2 at the beginning, thus it also uses k bins for packing two items from the part L_1. Therefore among the items of size $1/2+\varepsilon$ only $n-2k$ can be paired with earlier items and the other ones need separate bin. Thus $\mathrm{A}(L_1L_2) \geq n-k+(n-(n-2k)) = n+k$. On the other hand, the optimal off-line algorithm can pack a smaller (size $1/3+\varepsilon$) item and a larger (size $1/2+\varepsilon$) item into each bin, thus $\mathrm{OPT}(L_1L_2) = n$.

So we obtained that there is a list for algorithm A where

$$\mathrm{A}(L)/\mathrm{OPT}(L) \geq \max\left\{\frac{n-k}{n/2}, \frac{n+k}{n}\right\} \geq 4/3\ .$$

Moreover for the above constructed lists $\mathrm{OPT}(L)$ is at least $\lceil n/2 \rceil$, which can be arbitrarily great. This yields that the above inequality proves that the asymptotic competitive ratio of A is at least 4/3, and this is what we wanted to prove. ■

The fundamental idea of the above proof is that a long sequence (in this proof L_1L_2) is considered, and depending on the behaviour of the algorithm a prefix of the sequence is selected as input for which the ratio of the costs is maximal. It is an evident extension to consider more difficult sequences. Many lower bounds have been proven based on different sequences. On the other hand, the computations which are necessary to analyse the sequence have become more and more difficult. Below we show how the analysis of such sequences can be interpreted as a mixed

integer programming problem, which makes it possible to use computers to develop lower bounds.

Consider the following sequence of items. Let $L = L_1 L_2 \dots L_k$, where L_i contains $n_i = \alpha_i n$ identical items of size a_i. If algorithm A is asymptotically C-competitive, then the inequality

$$C \geq \limsup_{n\to\infty} \frac{\mathrm{A}(L_1 \dots L_j)}{\mathrm{OPT}(L_1 \dots L_j)}$$

is valid for each j. It is enough to consider an algorithm for which the technique can achieve the minimal lower bound, thus our aim is to determine the value

$$R = \min_{\mathrm{A}} \max_{j=1,\dots,k} \limsup_{n\to\infty} \frac{\mathrm{A}(L_1 \dots L_j)}{\mathrm{OPT}(L_1 \dots L_j)} \ ,$$

which value gives a lower bound on the possible asymptotic competitive ratio. We can determine this value as an optimal solution of a mixed integer programming problem. To define this problem we need the following definitions.

The contain of a bin can be described by the packing pattern of the bin, which gives that how many elements are contained in the bin from each subsequence. Formally, a ***packing pattern*** is a k-dimensional vector $(p_1, \dots, p_k)$, where coordinate p_j is the number of elements contained in the bin from subsequence L_j. For the packing patterns the constraint $\sum_{j=1}^{k} a_j p_j \leq 1$ must hold. (This constraint ensures that the items described by the packing pattern fit into the bin.)

Classify the set T of the possible packing patterns. For each j let T_j be the set of the patterns for which the first positive coordinate is the j-th one. (Pattern p belongs to class T_j if $p_i = 0$ for each $i < j$ and $p_j > 0$.)

Consider the packing produced by A. Each bin is packed by some packing pattern, therefore the packing can be described by the packing patterns. For each $p \in T$ denote the number of bins which are packed by the pattern p by $n(p)$. The packing produced by the algorithm is given by variables $n(p)$.

Observe that the bins which are packed by a pattern from class T_j receive their first element from subsequence L_j. Therefore we obtain that the number of bins opened by A to pack the elements of subsequence $L_1 \dots L_j$ can be given by variables $n(p)$ as follows:

$$\mathrm{A}(L_1 \dots L_j) = \sum_{i=1}^{j} \sum_{p\in T_i} n(p) \ .$$

Consequently, for a given n the required value R can be determined by the solution of the following mixed integer programming problem.

Min R

$$\begin{aligned}
&\textstyle\sum_{p\in T} p_j n(p) = n_j, && 1 \leq j \leq k,\\
&\textstyle\sum_{i=1}^{j} \sum_{p\in T_i} n(p) \leq R \cdot \mathrm{OPT}(L_1 \dots L_j), && 1 \leq j \leq k,\\
&n(p) \in \{0, 1, \dots\}, && p \in T.
\end{aligned}$$

The first k constraints describe that the algorithm has to pack all items. The

second k constraints describe that R is at least as large as the ratio of the on-line and off-line costs for the subsequences considered.

The set T of the possible packing patterns and also the optimal solutions $OPT(L_1 \dots L_j)$ can be determined by the list $L_1 L_2 \dots L_k$.

In this problem the number and the value of the variables can be large, thus instead of the problem its linear programming relaxation is considered. Moreover, we are interested in the solution under the assumption that n tends to ∞ and it can be proven that the integer programming and the linear programming relaxation give the same bound in this case.

The best currently known bound was proven by this method and it states that no on-line algorithm can have smaller asymptotic competitive ratio than 1.5401.

9.4.2. Multidimensional models

The bin packing problem has three different multidimensional generalisations: the vector packing, the box packing and the strip packing models. We consider only the strip packing problem in details. For the other generalisations we give only the model. In the ***vector packing problem*** the input is a list of d-dimensional vectors, and the algorithm has to pack these vectors into the minimal number of bins. A packing is legal for a bin if for each coordinate the sum of the values of the elements packed into the bin is at most 1. In the on-line version the vectors are coming one by one and the algorithm has to assign the vectors to the bins without any information about the further vectors. In the ***box packing problem*** the input is a list of d-dimensional boxes and the goal is to pack the items into the minimal number of d-dimensional unit cube without overlapping. In the on-line version the items are coming one by one and the algorithm has to pack them into the cubes without any information about the further items.

On-line strip packing

In the ***strip packing problem*** there is a set of two dimensional rectangles, defined by their widths and heights, and the task is to pack them into a vertical strip of width w without rotation minimising the total height of the strip. We assume that the widths of the rectangles is at most w and the heights of the rectangles is at most 1. This problem appears in many situations. Usually, scheduling of tasks with shared resource involves two dimensions, namely the resource and the time. We can consider the widths as the resources and the heights as the times. Our goal is to minimise the total amount of time used. Some applications can be found in computer scheduling problems. We consider the on-line version where the rectangles arrive from a list one by one and we have to pack each rectangle into the vertical strip without any information about the further items. Most of the algorithms developed for the strip packing problem belong to the class of shelf algorithms. We consider this family of algorithms below.

Shelf algorithms

A basic way of packing into the strip is to define shelves and pack the rectangles onto the shelves. By ***shelf*** we mean a rectangular part of the strip. Shelf packing

algorithms place each rectangle onto one of the shelves. If the algorithm decides which shelf will contain the rectangle, then the rectangle is placed onto the shelf as much to the left as it is possible without overlapping the other rectangles placed onto the shelf earlier. Therefore, after the arrival of a rectangle, the algorithm has to make two decisions. The first decision is whether to create a new shelf or not. If the algorithm creates a new shelf, it also has to decide the height of the new shelf. The created shelves always start from the top of the previous shelf. The first shelf is placed to the bottom of the strip. The algorithm also has to choose which shelf to put the rectangle onto. Hereafter we will say that it is possible to pack a rectangle onto a shelf, if there is enough room for the rectangle on the shelf. It is obvious that if a rectangle is higher than a shelf, we cannot place it onto the shelf.

We consider only one algorithm in details. This algorithm was developed and analysed by Baker and Schwarz in 1983 and it is called NEXT-FIT-SHELF (NFS_r) algorithm. The algorithm depends on a parameter $r < 1$. For each j there is at most one active shelf with height r^j. We define how the algorithm works below.

After the arrival of a rectangle $p_i = (w_i, h_i)$ choose a value for k which satisfies $r^{k+1} < h_i \leq r^k$. If there is an active shelf with height r^k and it is possible to pack the rectangle onto it, then pack it there. If there is no active shelf with height r^k, or it is not possible to pack the rectangle onto the active shelf with height r^k, then create a new shelf with height r^k, put the rectangle onto it, and let this new shelf be the active shelf with height r^k (if we had an active shelf with height r^k earlier, then we close it).

Example 9.7 Let $r = 1/2$. Suppose that the size of the first item is $(w/2, 3/4)$. Therefore, it is assigned to a shelf of height 1. We define a shelf of height 1 at the bottom of the strip; this will be the active shelf with height 1 and we place the item into the left corner of this shelf. Suppose that the size of the next item is $(3w/4, 1/4)$. In this case it is placed onto a shelf of height $1/4$. There is no active shelf with this height so we define a new shelf of height $1/4$ on the top of the previous shelf. This will be the active shelf of height $1/4$ and the item is placed onto its left corner. Suppose that the size of the next item is $(3w/4, 5/8)$. This item is placed onto a shelf of height 1. It is not possible to pack it onto the active shelf, thus we close the active shelf and we define a new shelf of height 1 on the top of the previous shelf. This will be the active shelf of height 1 and the item is placed into its left corner. Suppose that the size of the next item is $(w/8, 3/16)$. This item is placed onto a shelf of height $1/4$. We can pack it onto the active shelf of height $1/4$, thus we pack it onto that shelf as left as it is possible.

For the competitive ratio of NFS_r the following statements are valid.

Theorem 9.17 *Algorithm* NFS_r *is* $\left(\frac{2}{r} + \frac{1}{r(1-r)}\right)$*-competitive. Algorithm* NFS_r *is asymptotically* $2/r$*-competitive.*

Proof First we prove that the algorithm is $\left(\frac{2}{r} + \frac{1}{r(1-r)}\right)$-competitive. Consider an arbitrary list of rectangles and denote it by L. Let H_A denote the sum of the heights of the shelves which are active at the end of the packing, and let H_C be the sum of the heights of the other shelves. Let h be the height of the highest active shelf ($h = r^j$ for some j), and let H be the height of the highest rectangle. Since the algorithm created a shelf with height h, we have $H > rh$. As there is at most 1

active shelf for each height,

$$H_A \leq h \sum_{i=0}^{\infty} r^i = \frac{h}{1-r} \leq \frac{H}{r(1-r)} \ .$$

Consider the shelves which are not active at the end. Consider the shelves of height hr^i for each i, and denote the number of the closed shelves by n_i. Let S be one of these shelves with height hr^i. The next shelf S' with height hr^i contains one rectangle which would not fit onto S. Therefore, the total width of the rectangles is at least w. Furthermore, the height of these rectangles is at least hr^{i+1}, thus the total area of the rectangles packed onto S and S' is at least hwr^{i+1}. If we pair the shelves of height hr^i for each i in this way, using the active shelf if the number of the shelves of the considered height is odd, we obtain that the total area of the rectangles assigned to the shelves of height hr^i is at least $wn_ihr^{i+1}/2$. Thus the total area of the rectangles is at least $\sum_{i=0}^{\infty} wn_ihr^{i+1}/2$, and this yields that $\mathrm{OPT}(L) \geq \sum_{i=0}^{\infty} n_ihr^{i+1}/2$. On the other hand, the total height of the closed shelves is $H_Z = \sum_{i=0}^{\infty} n_ihr^i$, and we obtain that $H_Z \leq 2\mathrm{OPT}(L)/r$.

Since $\mathrm{OPT}(L) \geq H$ is valid we proved the required inequality

$$\mathrm{NFS}_r(L) \leq \mathrm{OPT}(L)(2/r + 1/r(1-r)) \ .$$

Since the heights of the rectangles are bounded by 1, H and H_A are bounded by a constant, so we obtain the result about the asymptotic competitive ratio immediately. ■

Besides this algorithm some other shelf algorithms have been investigated for the solution of the problem. We can interpret the basic idea of NFS_r as follows. We define classes of items belonging to types of shelves, and the rectangles assigned to the classes are packed by the classical bin packing algorithm NF. It is an evident idea to use other bin packing algorithms. The best shelf algorithm known at present time was developed by Csirik and Woeginger in 1997. That algorithm uses the harmonic bin packing algorithm to pack the rectangles assigned to the classes.

Exercises

9.4-1 Suppose that the size of the items is bounded above by 1/3. Prove that under this assumption the asymptotic competitive ratio of NF is 3/2.

9.4-2 Suppose that the size of the items is bounded above by 1/3. Prove Lemma 9.15 under this assumption.

9.4-3 Suppose that the sequence of items is given by a list $L_1L_2L_3$, where L_1 contains n items of size 1/2, L_2 contains n items of size 1/3, L_3 contains n items of size 1/4. Which packing patterns can be used? Which patterns belong to class T_2?

9.4-4 Consider the version of the strip packing problem where one can lengthen the rectangles keeping the area fixed. Consider the extension of NFS_r which lengthen the rectangles before the packing to the same height as the shelf which is chosen to pack them onto. Prove that this algorithm is $2 + \frac{1}{r(1-r)}$-competitive.

9.5. On-line scheduling

The area of scheduling theory has a huge literature. The first result in on-line scheduling belongs to Graham, who analysed the List scheduling algorithm in 1966. We can say that despite of the fact that Graham did not use the terminology which was developed in the area of the on-line algorithms, and he did not consider the algorithm as an on-line algorithm, he analysed it as an approximation algorithm.

From the area of scheduling we only recall the definitions which are used in this chapter.

In a scheduling problem we have to find an optimal schedule of jobs. We consider the parallel machines case, where m machines are given, and we can use them to schedule the jobs. In the most fundamental model each job has a known processing time and to schedule the job we have to assign it to a machine, and we have to give its starting time and a completion time, where the difference between the completion time and the starting time is the processing time. No machine may simultaneously run two jobs.

Concerning the machine environment three different models are considered. If the processing time of a job is the same for each machine, then we call the machines identical machines. If each machine has a speed s_i, the jobs have a processing weight p_j and the processing time of job j on the i-th machine is p_j/s_i, then we call the machines related machines. If the processing time of job j is given by an arbitrary positive vector $P_j = (p_j(1), \ldots, p_j(m))$, where the processing time of the job on the i-th machine is $p_j(i)$, then we call the machines unrelated machines.

Many objective functions are considered for scheduling problems, but here we consider only such models where the goal is the minimisation of the makespan (the maximal completion time).

In the next subsection we define the two most fundamental on-line scheduling models, and in the following two subsections we consider these models in details.

9.5.1. On-line scheduling models

Probably the following models are the most fundamental examples of on-line machine scheduling problems.

LIST model

In this model we have a fixed number of machines denoted by $M_1, M_2, \ldots, M_m$, and the jobs arrive from a list. This means that the jobs and their processing times are revealed to the on-line algorithm one by one. When a job is revealed, the on-line algorithm has to assign the job to a machine with a starting time and a completion time irrevocably.

By the ***load*** of a machine, we mean the sum of the processing times of all jobs assigned to the machine. Since the objective function is to minimise the maximal completion time, it is enough to consider the schedules where the jobs are scheduled on the machines without idle time. For these schedules the maximal completion time is the load for each machine. Therefore this scheduling problem is reduced to a load balancing problem, i.e. the algorithm has to assign the jobs to the machines

minimising the maximum load, which is the makespan in this case.

Example 9.8 Consider the LIST model and two identical machines. Consider the following sequence of jobs where the jobs are given by their processing time: $I = \{j_1 = 4, j_2 = 3, j_3 = 2, j_4 = 5\}$. The on-line algorithm first receives job j_1 from the list, and the algorithm has to assign this job to one of the machines. Suppose that the job is assigned to machine M_1. After that the on-line algorithm receives job j_2 from the list, and the algorithm has to assign this job to one of the machines. Suppose that the job is assigned to machine M_2. After that the on-line algorithm receives job j_3 from the list, and the algorithm has to assign this job to one of the machines. Suppose that the job is assigned to machine M_2. Finally, the on-line algorithm receives job j_4 from the list, and the algorithm has to assign this job to one of the machines. Suppose that the job is assigned to machine M_1. Then the loads on the machines are $4+5$ and $3+2$, and we can give a schedule where the maximal completion times on the machines are the loads: we can schedule the jobs on the first machine in time intervals $(0, 4)$ and $(4, 9)$, and we can schedule the jobs on the second machine in time intervals $(0, 3)$ and $(3, 5)$.

TIME model

In this model there are a fixed number of machines again. Each job has a processing time and a ***release time***. A job is revealed to the on-line algorithm at its release time. For each job the on-line algorithm has to choose which machine it will run on and assign a start time. No machine may simultaneously run two jobs. Note that the algorithm is not required to assign a job immediately at its release time. However, if the on-line algorithm assigns a job at time t then it cannot use information about jobs released after time t and it cannot start the job before time t. Our aim is to minimise the makespan.

Example 9.9 Consider the TIME model with two related machines. Let M_1 be the first machine with speed 1, and M_2 be the second machine with speed 2. Consider the following input $I = \{j_1 = (1, 0), j_2 = (1, 1), j_3 = (1, 1), j_4 = (1, 1)\}$, where the jobs are given by the (processing time, release time) pairs. Thus a job arrives at time 0 with processing time 1, and the algorithm can either start to process it on one of the machines or wait for jobs with larger processing time. Suppose that the algorithm waits till time $1/2$ and then it starts to process the job on machine M_1. The completion time of the job is $3/2$. At time 1 three further jobs arrive, and at that time only M_2 can be used. Suppose that the algorithm starts to process job j_2 on this machine. At time $3/2$ both jobs are completed. Suppose that the remaining jobs are started on machines M_1 and M_2, and the completion times are $5/2$ and 2, thus the makespan achieved by the algorithm is $5/2$. Observe that an algorithm which starts the first job immediately at time 0 can make a better schedule with makespan 2. But it is important to note that in some cases it can be useful to wait for larger jobs before starting a job.

9.5.2. LIST model

The first algorithm in this model has been developed by Graham. Algorithm LIST assigns each job to the machine where the actual load is minimal. If there are more machines with this property, it uses the machine with the smallest index. This means

that the algorithm tries to balance the loads on the machines. The competitive ratio of this algorithm is determined by the following theorem.

Theorem 9.18 *The competitive ratio of algorithm* LIST *is* $2 - 1/m$ *in the case of identical machines.*

Proof First we prove that the algorithm is $2-1/m$-competitive. Consider an arbitrary input sequence denoted by $\sigma = \{j_1, \ldots, j_n\}$, and denote the processing times by $p_1, \ldots, p_n$. Consider the schedule produced by LIST. Let j_l be a job with maximal completion time. Investigate the starting time S_l of this job. Since LIST chooses the machine with minimal load, thus the load was at least S_l on each of the machines when j_l was scheduled. Therefore we obtain that

$$S_l \leq \frac{1}{m} \sum_{\substack{j=1 \\ j \neq l}}^{n} p_j = \frac{1}{m}(\sum_{j=1}^{n} p_j - p_l) = \frac{1}{m}(\sum_{j=1}^{n} p_j) - \frac{1}{m} p_l \ .$$

This yields that

$$\mathrm{LIST}(\sigma) = S_l + p_l \leq \frac{1}{m}(\sum_{j=1}^{n} p_j) + \frac{m-1}{m} p_l \ .$$

On the other hand, OPT also processes all of the jobs, thus we obtain that $\mathrm{OPT}(\sigma) \geq \frac{1}{m}(\sum_{j=1}^{n} p_j)$. Furthermore, p_l is scheduled on one of the machines of OPT, thus $\mathrm{OPT}(\sigma) \geq p_l$. Due to these bounds we obtain that

$$LIST(\sigma) \leq (1 + \frac{m-1}{m})\mathrm{OPT}(\sigma) \ ,$$

which inequality proves that LIST is $2 - 1/m$-competitive.

Now we prove that the bound is tight. Consider the following input. It contains $m(m-1)$ jobs with processing time $1/m$ and one job with processing time 1. LIST assigns $m-1$ small jobs to each machine and the last large job is assigned to M_1. Therefore its makespan is $1 + (m-1)/m$. On the other hand, the optimal algorithm schedules the large job on M_1 and m small jobs on the other machines, and its makespan is 1. Thus the ratio of the makespans is $2 - 1/m$ which shows that the competitive ratio of LIST is at least $2 - 1/m$. ■

Although it is hard to imagine any other algorithm for the on-line case, many other algorithms have been developed. The competitive ratios of the better algorithms tend to smaller numbers than 2 as the number of machines tends to ∞. Most of these algorithms are based on the following idea. The jobs are scheduled keeping the load uniformly on most of the machines, but in contrast to LIST, the loads are kept low on some of the machines, keeping the possibility of using these machines for scheduling large jobs which may arrive later.

Below we consider the more general cases where the machines are not identical. LIST may perform very badly, and the processing time of a job can be very large on the machine where the actual load is minimal. However, we can easily change the greedy idea of LIST as follows. The extended algorithm is called GREEDY and it assigns the job to the machine where the load with the processing time of the job is

minimal. If there are several machines which have minimal value, then the algorithm chooses the machine where the processing time of the job is minimal from them, if there are several machines with this property, the algorithm chooses the one with the smallest index from them.

Example 9.10 Consider the case of related machines where there are 3 machines M_1, M_2, M_3 and the speeds are $s_1 = s_2 = 1$, $s_2 = 3$. Suppose that the input is $I = \{j_1 = 2, j_2 = 1, j_3 = 1, j_4 = 3, j_5 = 2\}$, where the jobs are defined by their processing weight. The load after the first job is 2/3 on machine M_3 and 2 on the other machines, thus j_1 is assigned to M_3. The load after job j_2 is 1 on all of the machines, and its processing time is minimal on machine M_3, thus GREEDY assigns it to M_3. The load after job j_3 is 1 on M_1 and M_2, and 4/3 on M_3, thus the job is assigned to M_1. The load after job j_4 is 4 on M_1, 3 on M_2, and 2 on M_3, thus the job is assigned to M_3. Finally, the load after job j_5 is 3 on M_1, 2 on and M_2, and 8/3 on M_3, thus the job is assigned to M_2.

Example 9.11 Consider the case of unrelated machines with two machines and the following input: $I = \{j_1 = (1, 2), j_2 = (1, 2), j_3 = (1, 3), j_4 = (1, 3)\}$, where the jobs are defined by the vectors of processing times. The load after job j_1 is 1 on M_1 and 2 on M_2, thus the job is assigned to M_1. The load after job j_2 is 2 on M_1 and also on M_2, thus the job is assigned to M_1, because it has smaller processing time. The load after job j_3 is 3 on M_1 and M_2, thus the job is assigned to M_1 because it has smaller processing time. Finally, the load after job j_4 is 4 on M_1 and 3 on M_2, thus the job is assigned to M_2.

The competitive ratio of the algorithm is determined by the following theorems.

Theorem 9.19 *The competitive ratio of algorithm* GREEDY *is m in the case of unrelated machines.*

Proof First we prove that the competitive ratio of the algorithm is at least m. Consider the following input sequence. Let $\varepsilon > 0$ be an arbitrarily small number. The sequence contains m jobs. The processing time of job j_1 is 1 on machine M_1, $1 + \varepsilon$ on machine M_m, and ∞ on the other machines, $(p_1(1) = 1, p_1(i) = \infty, i = 2, \ldots, m-1, p_1(m) = 1 + \varepsilon)$. For job j_i, $i = 2, \ldots, m$ the processing time is i on machine M_i, $1+\varepsilon$ on machine M_{i-1} and ∞ on the other machines $(p_i(i-1) = 1+\varepsilon$, $p_i(i) = i$, $p_i(j) = \infty$, if $j \neq i-1$ and $j \neq i)$.

In this case job j_i is scheduled on M_i by GREEDY and the makespan is m. On the other hand, the optimal off-line algorithm schedules j_1 on M_m and j_i is scheduled on M_{i-1} for the other jobs, thus the optimal makespan is $1 + \varepsilon$. The ratio of the makespans is $m/(1+\varepsilon)$. This ratio tends to m, as ε tends to 0, and this proves that the competitive ratio of the algorithm is at least m.

Now we prove that the algorithm is m-competitive. Consider an arbitrary input sequence, denote the makespan in the optimal off-line schedule by L^* and let $L(k)$ denote the maximal load in the schedule produced by GREEDY after scheduling the first k jobs. Since the processing time of the i-th job is at least $\min_j p_i(j)$, and the load is at most L^* on the machines in the off-line optimal schedule, we obtain that $mL^* \geq \sum_{i=1}^{n} \min_j p_i(j)$.

We prove by induction that the inequality $L(k) \leq \sum_{i=1}^{k} \min_j p_i(j)$ is valid. Since the first job is assigned to the machine where its processing time is minimal, the

statement is obviously true for $k = 1$. Let $1 \leq k < n$ be an arbitrary number and suppose that the statement is true for k. Consider the $k+1$-th job. Let M_l be the machine where the processing time of this job is minimal. If we assign the job to M_l, then we obtain that the load on this machines is at most $L(k) + p_{k+1}(l) \leq \sum_{i=1}^{k+1} \min_j p_i(j)$ from the induction hypothesis.

On the other hand, the maximal load in the schedule produced by GREEDY can not be more than the maximal load in the case when the job is assigned to M_l, thus $L(k+1) \leq \sum_{i=1}^{k+1} \min_j p_i(j)$, which means that we proved the inequality for $k+1$.

Therefore we obtained that $mL^* \geq \sum_{i=1}^{n} \min_j p_i(j) \geq L(n)$, which yields that the algorithm is m-competitive. ■

To investigate the case of the related machines consider an arbitrary input. Let L and L^* denote the makespans achieved by GREEDY and OPT respectively. The analysis of the algorithm is based on the following lemmas which give bounds on the loads of the machines.

Lemma 9.20 *The load on the fastest machine is at least $L - L^*$.*

Proof Consider the schedule produced by GREEDY. Consider a job J which causes the makespan (its completion time is maximal). If this job is scheduled on the fastest machine, then the lemma follows immediately, i.e. the load on the fastest machine is L. Suppose that J is not scheduled on the fastest machine. The optimal maximal load is L^*, thus the processing time of J on the fastest machine is at most L^*. On the other hand, the completion time of J is L, thus at the time when the job was scheduled the load was at least $(L - L^*)$ on the fastest machine, otherwise GREEDY would assign J to the fastest machine. ■

Lemma 9.21 *If the loads are at least l on all machines having a speed of at least v then the loads are at least $l - 4L^*$ on all machines having a speed of at least $v/2$.*

Proof If $l < 4L^*$, then the statement is obviously valid. Suppose that $l \geq 4L^*$. Consider the jobs which are scheduled by GREEDY on the machines having a speed of at least v in the time interval $[l - 2L^*, l]$. The total processing weight of these jobs is at least $2L^*$ times the total speed of the machines having a speed of at least v. This yields that there exists a job among them which is assigned by OPT to a machine having a speed of smaller than v (otherwise the optimal off-line makespan would be larger than L^*). Let J be such a job.

Since OPT schedules J on a machine having a speed of smaller than v, thus the processing weight of J is at most vL^*. This yields that the processing time of J is at most $2L^*$ on the machines having a speed of at least $v/2$. On the other hand, GREEDY produces a schedule where the completion time of J is at least $l - 2L^*$, thus at the time when the job was scheduled the loads were at least $l - 4L^*$ on the machines having a speed of at most $v/2$ (otherwise GREEDY would assign J to one of these machines). ■

Now we can prove the following statement.

Theorem 9.22 *The competitive ratio of algorithm* GREEDY *is $\Theta(\lg m)$ in the case of the related machines.*

Proof First we prove that GREEDY is $O(\lg m)$-competitive. Consider an arbitrary input. Let L and L^* denote the makespans achieved by GREEDY and OPT, respectively.

Let $v_{\max}$ be the speed of the fastest machine. Then by Lemma 9.20 the load on this machine is at least $L - L^*$. Then using Lemma 9.21 we obtain that the loads are at least $L - L^* - 4iL^*$ on the machines having a speed of at least $v_{\max}2^{-i}$. Therefore the loads are at least $L - (1 + 4\lceil \lg m \rceil)L^*$ on the machines having a speed of at least $v_{\max}/m$. Denote the set of the machines having a speed of at most $v_{\max}/m$ by I.

Denote the sum of the processing weights of the jobs by W. OPT can find a schedule of the jobs which has maximal load L^*, and there are at most m machines having smaller speed than $v_{\max}/m$, thus

$$W \leq L^* \sum_{i=1}^{m} v_i \leq mL^* v_{\max}/m + L^* \sum_{i \notin I} v_i \leq 2L^* \sum_{i \notin I} v_i \ .$$

On the other hand, GREEDY schedules the same jobs, thus the load on some machine not included in I is smaller than $2L^*$ in the schedule produced by GREEDY (otherwise we would obtain that the sum of the processing weights is greater than W).

Therefore we obtain that

$$L - (1 + 4\lceil \lg m \rceil)L^* \leq 2L^* \ ,$$

which yields that $L \leq 3 + 4\lceil \lg m \rceil)L^*$, which proves that GREEDY is $O(\lg m)$-competitive.

Now we prove that the competitive ratio of the algorithm is at least $\Omega(\lg m)$. Consider the following set of machines: G_0 contains one machine with speed 1 and G_1 contains 2 machines with speed $1/2$. For each $i = 1, 2, \ldots, k$, G_i contains machines with speed 2^{-i}, and G_i contains $|G_i| = \sum_{j=0}^{i-1} |G_j| 2^{i-j}$ machines. Observe that the number of jobs of processing weight 2^{-i} which can be scheduled during 1 time unit is the same on the machines of G_i and on the machines of $G_0 \cup G_1 \ldots, \cup G_{i-1}$. It is easy to calculate that $|G_i| = 2^{2i-1}$, if $i \geq 1$, thus the number of machines is $1 + \frac{2}{3}(4^k - 1)$.

Consider the following input sequence. In the first phase $|G_k|$ jobs arrive having processing weight 2^{-k}, in the second phase $|G_{k-1}|$ jobs arrive having processing weight $2^{-(k-1)}$, in the i-th phase $|G_i|$ jobs arrive with processing weight 2^{-i}, and the sequence ends with the $k+1$-th phase, which contains one job with processing weight 1. An off-line algorithm can schedule the jobs of the i-th phase on the machines of set G_{k+1-i} achieving maximal load 1, thus the optimal off-line cost is at most 1.

Investigate the behaviour of algorithm GREEDY on this input. The jobs of the first phase can be scheduled on the machines of $G_0, \ldots, G_{k-1}$ during 1 time unit, and it takes also 1 time unit to process these jobs on the machines of G_k. Thus GREEDY schedules these jobs on the machines of $G_0, \ldots, G_{k-1}$, and each load is 1 on these machines after the first phase. Then the jobs of the second phase are scheduled on the machines of $G_0, \ldots, G_{k-2}$, the jobs of the third phase are scheduled on the machines of $G_0, \ldots, G_{k-3}$ and so on. Finally, the jobs of the k-th and $k+1$-th phase are scheduled on the machine of set G_0. Thus the cost of GREEDY is $k+1$, (this is the load on the machine of set G_0). Since $k = \Omega(\lg m)$, we proved the required

statement. ■

9.5.3. TIME model

In this model we investigate only one algorithm. The basic idea is to divide the jobs into groups by the release time and to use an optimal off-line algorithm to schedule the jobs of the groups. This algorithm is called ***interval scheduling algorithm*** and we denote it by INTV. Let t_0 be the release time of the first job, and $i = 0$. The algorithm is defined by the following pseudocode:

INTV(I)

```
1  while not end of sequence
2        let H_i be the set of the unscheduled jobs released till t_i
3        let OFF_i be an optimal off-line schedule of the jobs of H_i
4        schedule the jobs as it is determined by OFF_i starting the schedule at t_i
5        let q_i be the maximal completion time
6        if a new job is released in time interval (t_i, q_i] or the sequence is ended
7           then t_{i+1} ← q_i
7           else  let t_{i+1} be the release time of the next job
8        i ← i + 1
```

Example 9.12 Consider two identical machines. Suppose that the sequence of jobs is $I = \{j_1 = (1,0), j_2 = (1/2,0), j_3 = (1/2,0), j_4 = (1,3/2), j_5 = (1,3/2), j_6 = (2,2)\}$, where the jobs are defined by the (processing time, release time) pairs. In the first iteration j_1, j_2, j_3 are scheduled: an optimal off-line algorithm schedules j_1 on machine M_1 and j_2, j_3 on machine M_2, so the jobs are completed at time 1. Since no new job have been released in the time interval $(0,1]$, the algorithm waits for a new job until time $3/2$. Then the second iteration starts: j_4 and j_5 are scheduled on M_1 and M_2 respectively in the time interval $[3/2, 5/2)$. During this time interval j_6 has been released thus at time $5/2$ the next iteration starts and INTV schedules j_6 on M_1 in the time interval $[5/2, 9/2]$.

The following statement holds for the competitive ratio of algorithm INTV.

Theorem 9.23 *In the TIME model algorithm* INTV *is 2-competitive.*

Proof Consider an arbitrary input and the schedule produced by INTV. Denote the number of iterations by i. Let $T_3 = t_{i+1} - t_i$, $T_2 = t_i - t_{i-1}$, $T_1 = t_{i-1}$ and let T_{OPT} denote the optimal off-line cost. In this case $T_2 \leq T_{\mathrm{OPT}}$. This inequality is obvious if $t_{i+1} \neq q_i$. If $t_{i+1} = q_i$, then the inequality holds, because also the optimal off-line algorithm has to schedule the jobs which are scheduled in the i-th iteration by INTV, and INTV uses an optimal off-line schedule in each iteration. On the other hand, $T_1 + T_3 \leq T_{\mathrm{OPT}}$. To prove this inequality first observe that the release time is at least $T_1 = t_{i-1}$ for the jobs scheduled in the i-th iteration (otherwise the algorithm would schedule them in the $i-1$-th iteration).

Therefore also the optimal algorithm has to schedule these jobs after time T_1. On the the other hand, it takes at least T_3 time units to process these jobs, because INTV uses optimal off-line algorithm in the iterations. The makespan of the schedule

produced by INTV is $T_1+T_2+T_3$, and we have shown that $T_1+T_2+T_3 \leq 2T_{\mathrm{OPT}}$, thus we proved that the algorithm is 2-competitive. ■

Some other algorithms have also been developed in the TIME model. Vestjens proved that the ***on-line* LPT** algorithm is 3/2-competitive. This algorithm schedules the longest unscheduled, released job at each time when some machine is available. The following lower bound for the possible competitive ratios of the on-line algorithms is also given by Vestjens.

Theorem 9.24 *The competitive ratio of any on-line algorithm is at least* 1.3473 *in the TIME model for minimising the makespan.*

Proof Let $\alpha \approx 0.3473$ be the solution of the equation $\alpha^3 - 3\alpha + 1 = 0$ which belongs to the interval $[1/3, 1/2]$. We prove that no on-line algorithm can have smaller competitive ratio than $1+\alpha$. Consider an arbitrary on-line algorithm, denote it by ALG. Investigate the following input sequence.

At time 0 one job arrives with processing time 1. Let S_1 be the time when the algorithm starts to process the job on one of the machines. If $S_1 > \alpha$, then the sequence is finished and $\mathrm{ALG}(I)/\mathrm{OPT}(I) > 1+\alpha$, which proves the statement. So we can suppose that $S_1 \leq \alpha$.

The release time of the next job is S_1 and its processing time is $\alpha/(1-\alpha)$. Denote its starting time by S_2. If $S_2 \leq S_1 + 1 - \alpha/(1-\alpha)$, then we end the sequence with $m-1$ jobs having release time S_2, and processing time $1+\alpha/(1-\alpha)-S_2$. In this case an optimal off-line algorithm schedules the first two jobs on the same machine and the last $m-1$ jobs on the other machines starting them at time S_2, thus its cost is $1+\alpha/(1-\alpha)$. On the other hand, the on-line algorithm must schedule one of the last $m-1$ jobs after the completion of the first or the second job, thus $\mathrm{ALG}(I) \geq 1+2\alpha/(1-\alpha)$ in this case, which yields that the competitive ratio of the algorithm is at least $1+\alpha$. Therefore we can suppose that $S_2 > S_1 + 1 - \alpha/(1-\alpha)$.

At time $S_1 + 1 - \alpha/(1-\alpha)$ further $m-2$ jobs arrive with processing times $\alpha/(1-\alpha)$ and one job with processing time $1-\alpha/(1-\alpha)$. The optimal off-line algorithm schedules the second and the last jobs on the same machine, and the other jobs are scheduled one by one on the other machines and the makespan of the schedule is $1+S_1$. Since before time $S_1 + 1 - \alpha/(1-\alpha)$ none of the last m jobs is started by ALG, after this time ALG must schedule at least two jobs on one of the machines and the maximal completion time is at least $S_1 + 2 - \alpha/(1-\alpha)$. Since $S_1 \leq \alpha$, the ratio $\mathrm{OPT}(I)/\mathrm{ALG}(I)$ is minimal if $S_1 = \alpha$, and in this case the ratio is $1+\alpha$, which proves the theorem. ■

Exercises

9.5-1 Prove that the competitive ratio is at least 3/2 for any on-line algorithm in the case of two identical machines.

9.5-2 Prove that LIST is not constant competitive in the case of unrelated machines.

9.5-3 Prove that the modification of INTV which uses a c-approximation schedule (a schedule with at most c times more cost than the optimal cost) instead of the optimal off-line schedule in each step is $2c$-competitive.

Problems

9-1 Paging problem
Consider the special case of the k-server problem, where the distance between each pair of points is 1. (This problem is equivalent with the on-line paging problem.) Analyse the algorithm which serves the requests not having server on their place by the server which was used least recently. (This algorithm is equivalent with the LRU paging algorithm.) Prove that the algorithm is k-competitive.

9-2* ALARM2 *algorithm
Consider the following alarming algorithm for the data acknowledgement problem. ALARM2 is obtained from the general definition with the values $e_j = 1/|\sigma_j|$. Prove that the algorithm is not constant-competitive.

9-3 Bin packing lower bound
Prove, that no on-line algorithm can have smaller competitive ratio than 3/2 using a sequence which contains items of size $1/7 + \varepsilon$, $1/3 + \varepsilon$, $1/2 + \varepsilon$, where ε is a small positive number.

9-4 Strip packing with modifiable rectangles
Consider the following version of the strip packing problem. In the new model the algorithms are allowed to lengthen the rectangles keeping the area fixed. Develop a 4-competitive algorithm for the solution of the problem.

9-5 On-line* LPT *algorithm
Consider the algorithm in the TIME model which starts the longest released job to schedule at each time when a machine is available. This algorithm is called on-line LPT. Prove that the algorithm is 3/2-competitive.

Chapter notes

More details about the results on on-line algorithms can be found in the books [29, 68].

The first results about the k-server problem (Theorems 9.1 and 9.2) are published by Manasse, McGeoch and Sleator in [158]. The presented algorithm for the line (Theorem 9.3) was developed by Chrobak, Karloff, Payne and Viswanathan (see [44]). Later Chrobak and Larmore extended the algorithm for trees in [42]. The first constant-competitive algorithm for the general problem was developed by Fiat, Rabani and Ravid (see [67]). The best known algorithm is based on the work function technique. The first work function algorithm for the problem was developed by Chrobak and Larmore in [43]. Koutsoupias and Papadimitriou have proven that the work function algorithm is $2k - 1$-competitive in [137].

The first mathematical model for the data acknowledgement problem and the first results (Theorems 9.5 and 9.6) are presented by Dooly, Goldman, and Scott in [61]. Albers and Bals considered a different objective function in [10]. Karlin Kenyon and Randall investigated randomised algorithms for the data acknowledgement problem in [127]. The LANDLORD algorithm was developed by Young in [261]. The detailed description of the results in the area of on-line routing can be found in the

survey [150] written by Leonardi. The exponential algorithm for the load balancing model is investigated by Aspnes, Azar, Fiat, Plotkin and Waarts in [12]. The exponential algorithm for the throughput objective function is applied by Awerbuch, Azar and Plotkin in [15].

A detailed survey about the theory of on-line bin packing is written by Csirik and Woeginger (see [53]). The algorithms NF and FF are analysed with competitive analysis by Johnson, Demers, Ullman, Garey and Graham in [121, 122], further results can be found in the PhD thesis of Johnson ([120]). Our Theorem 9.12 is a special case of Theorem 1 in [118] and Theorem 9.13 is a special case of Theorems 5.8 and 5.9 of the book [45] and Corollary 20.13 in the twentieth chapter of this book [?]. Van Vliet applied the packing patterns to prove lower bounds for the possible competitive ratios in [247, 266]. For the on-line strip packing problem algorithm NFS_r was developed and analysed by Baker and Schwarz in [19]. Later further shelf packing algorithms were developed, the best shelf packing algorithm for the strip packing problem was developed by Csirik and Woeginger in [54].

A detailed survey about the results in the area of on-line scheduling was written by Sgall ([218]). The first on-line result is the analysis of algorithm LIST, it was published by Grahamin [91]. Many further algorithms were developed and analysed for the case of identical machines, the algorithm with smallest competitive ratio (tends to 1.9201 as the number of machines tends to ∞) was developed by Fleischer and Wahl in [70]. The lower bound for the competitive ratio of GREEDY in the related machines model was proved by Cho and Sahni in [40]. The upper bound, the related machines case and a more sophisticated exponential function based algorithm were presented by Aspnes, Azar, Fiat, Plotkin and Waarts in [12]. A summary of further results about the applications of the exponential function technique in the area of on-line scheduling can be found in the paper of Azar ([16]). The interval algorithm presented in the TIME model and Theorem 9.23 are based on the results of Shmoys, Wein and Williamson (see [223]). A detailed description of further results (on-line LPT, lower bounds) in the area TIME model can be found in the PhD thesis of Vestjens [267]. We presented only the most fundamental on-line scheduling models in the chapter, although an interesting model has been developed recently, where the number of the machines is not fixed, and the algorithm is allowed to purchase machines. The model is investigated in papers [117] and [62].

Problem *9-1* is based on [229], Problem *9-2* is based on [61], Problem *9-3* is based on [260], Problem *9-4* is based on [116] and Problem *9-5* is based on [267].

10. Game Theory

In many situations in engineering and economy there are cases when the conflicting interests of several decision makers have to be taken into account simultaneously, and the outcome of the situation depends on the actions of these decision makers. One of the most popular methodology and modeling is based on game theory.

Let N denote the number of decision makers (who will be called ***players***), and for each $k = 1, 2, \ldots, N$ let S_k be the set of all feasible actions of player $\mathcal{P}_k$. The elements $s_k \in S_k$ are called ***strategies*** of player $\mathcal{P}_k$, S_k is the ***strategy set*** of this player. In any realization of the ***game*** each player selects a strategy, then the vector $\mathbf{s} = (s_1, s_2, \ldots, s_N)$ ($s_k \in S_k$, $k = 1, 2, \ldots, N$) is called a ***simultaneous strategy vector*** of the players. For each $\mathbf{s} \in S = S_1 \times S_2 \times \cdots \times S_N$ each player has an outcome which is assumed to be a real value. This value can be imagined as the utility function value of the particular outcome, in which this function represents how player $\mathcal{P}_k$ evaluates the outcomes of the game. If $f_k(s_1, \ldots, s_N)$ denotes this value, then $f_k : S \to \mathbb{R}$ is called the ***payoff function*** of player $\mathcal{P}_k$. The value $f_k(\mathbf{s})$ is called the ***payoff*** of player $\mathcal{P}_k$ and $(f_1(\mathbf{s}), \ldots, f_N(\mathbf{s}))$ is called the ***payoff vector***. The number N of players, the sets S_k of strategies and the payoff functions f_k ($k = 1, 2, \ldots, N$) completely determine and define the N-person game. We will also use the notation $G = \{N; S_1, S_2, \ldots, S_N; f_1, f_2, \ldots, f_N\}$ for this game.

The solution of game G is the ***Nash-equilibrium***, which is a simultaneous strategy vector $\mathbf{s}^\star = (s_1^\star, \ldots, s_N^\star)$ such that for all k,

1. $s_k^\star \in S_k$;
2. for all $s_k \in S_k$,

$$f_k(s_1^\star, s_2^\star, \ldots, s_{k-1}^\star, s_k, s_{k+1}^\star, \ldots, s_N^\star) \le f_k(s_1^\star, s_2^\star, \ldots, s_{k-1}^\star, s_k^\star, s_{k+1}^\star, \ldots, s_N^\star)\ . \tag{10.1}$$

Condition 1 means that the k-th component of the equilibrium is a feasible strategy of player $\mathcal{P}_k$, and condition 2 shows that none of the players can increase its payoff by unilaterally changing its strategy. In other words, it is the interest of all players to keep the equilibrium since if any player departs from the equilibrium, its payoff does not increase.

		Player $\mathcal{P}_2$	
		N	C
Player $\mathcal{P}_1$	N	$(-2,-2)$	$(-10,-1)$
	C	$(-1,-10)$	$(-5,-5)$

Figure 10.1. Prisoner's dilemma.

10.1. Finite games

Game G is called finite if the number of players is finite and all strategy sets S_k contain finitely many strategies. The most famous two-person finite game is the ***prisoner's dilemma,*** which is the following.

Example 10.1 The players are two prisoners who committed a serious crime, but the prosecutor has only insufficient evidence to prosecute them. The prisoners are held in separate cells and cannot communicate, and the prosecutor wants them to cooperate with the authorities in order to get the needed additional information. So $N = 2$, and the strategy sets for both players have two elements: cooperating (C), or not cooperating (N). It is told to both prisoners privately that if he is the only one to confess, then he will get only a light sentence of 1 year, while the other will go to prison for a period of 10 years. If both confess, then their reward will be a 5 year prison sentence each, and if none of them confesses, then they will be convicted to a less severe crime with sentence of 2 years each. The objective of both players are to minimize the time spent in prison, or equivalently to maximize its negative. Figure 10.1 shows the payoff values, where the rows correspond to the strategies of player $\mathcal{P}_1$, the columns show the strategies of player $\mathcal{P}_2$, and for each strategy pair the first number is the payoff of player $\mathcal{P}_1$, and the second number is the payoff of player $\mathcal{P}_2$. Comparing the payoff values, it is clear that only (C, C) can be equilibrium, since

$$\begin{aligned} f_2(\mathrm{N,N}) = -2 &< f_2(\mathrm{N,C}) = -1\ , \\ f_1(\mathrm{N,C}) = -10 &< f_1(\mathrm{C,C}) = -5\ , \\ f_2(\mathrm{C,N}) = -10 &< f_2(C,C) = -5\ . \end{aligned}$$

The strategy pair (C, C) is really an equilibrium, since

$$\begin{aligned} f_1(\mathrm{C,C}) = -5 > f_1(\mathrm{N,C}) = -10\ , \\ f_2(\mathrm{C,C}) = -5 > f_2(\mathrm{C,N}) = -10\ . \end{aligned}$$

In this case we have a unique equilibrium.

The existence of an equilibrium is not guaranteed in general, and if equilibrium exists, it might not be unique.

Example 10.2 Modify the payoff values of Figure 10.1 as shown in Figure 10.2. It is easy to see that no equilibrium exists:

$$\begin{aligned} f_1(\mathrm{N,N}) = 1 &< f_1(\mathrm{C,N}) = 2\ , \\ f_2(\mathrm{C,N}) = 4 &< f_2(\mathrm{C,C}) = 5\ , \\ f_1(\mathrm{C,C}) = 0 &< f_1(\mathrm{N,C}) = 2\ , \\ f_2(\mathrm{N,C}) = 1 &< f_2(\mathrm{N,N}) = 2\ . \end{aligned}$$

		Player $\mathcal{P}_2$	
		N	C
Player $\mathcal{P}_1$	N	$(1,2)$	$(2,1)$
	C	$(2,4)$	$(0,5)$

Figure 10.2. Game with no equilibrium.

If all payoff values are identical, then we have multiple equilibria: any strategy pair is an equilibrium.

10.1.1. Enumeration

Let N denote the number of players, and for the sake of notational convenience let $s_k^{(1)}, \ldots, s_k^{(n_k)}$ denote the feasible strategies of player $\mathcal{P}_k$. That is, $S_k = \{s_k^{(1)}, \ldots, s_k^{(n_k)}\}$. A strategy vector $\mathbf{s}^\star = (s_1^{(i_1)}, \ldots, s_N^{(i_N)})$ is an equilibrium if and only if for all $k = 1, 2, \ldots, N$ and $j \in \{1, 2, \ldots, n_k\} \setminus i_k$,

$$f_k(s_1^{(i_1)}, \ldots, s_{k-1}^{(i_{k-1})}, s_k^{(j)}, s_{k+1}^{(i_{k+1})}, \ldots, s_N^{(i_N)}) \leq f_k(s_1^{(i_1)}, \ldots, s_{k-1}^{(i_{k-1})}, s_k^{(i_k)}, s_{k+1}^{(i_{k+1})}, \ldots, s_N^{(i_N)}). \tag{10.2}$$

Notice that in the case of finite games inequality (10.1) reduces to (10.2).

In applying the enumeration method, inequality (10.2) is checked for all possible strategy N-tuples $\mathbf{s}^\star = (s_1^{(i_1)}, \ldots, s_N^{(i_N)})$ to see if (10.2) holds for all k and j. If it does, then $\mathbf{s}^\star$ is an equilibrium, otherwise not. If during the process of checking for a particular $\mathbf{s}^\star$ we find a k and j such that (10.2) is violated, then $\mathbf{s}^\star$ is not an equilibrium and we can omit checking further values of k and j. This algorithm is very simple, it consists of $N + 2$ imbedded loops with variables $i_1, i_2, \ldots, i_N, k$ and j.

The maximum number of comparisons needed equals

$$\left(\prod_{k=1}^{N} n_k\right)\left(\sum_{k=1}^{N}(n_k - 1)\right),$$

however in practical cases it might be much lower, since if (10.2) is violated with some j, then the comparison must stop for the same strategy vector.

The algorithm can formally be given as follows:

PRISONER-ENUMERATION(S_k)

```
1  for i_1 ← 1 to n_1
2      do for i_2 ← 1 to n_2
3             ...
4             do for i_N ← 1 to n_N
5                do key ← 0
6                for k ← 1 to N
7                     do for j ← 1 to n_k
8                           do if (10.2) fails
9                                 then key ← 1 and go to 10
10               if   key = 0
11                    then (s_1^(i_1), ..., s_N^(i_N)) is equilibrium
12 return (s_1^(i_1), ..., s_N^(i_N))
```

Consider next the two-person case, N=2, and introduce the $n_1 \times n_2$ real matrixes $\mathbf{A}^{(1)}$ and $\mathbf{A}^{(2)}$ with (i,j) elements $f_1(i,j)$ and $f_2(i,j)$ respectively. Matrixes $\mathbf{A}^{(1)}$ and $\mathbf{A}^{(2)}$ are called the ***payoff matrixes*** of the two players. A strategy vector $\left(s_1^{(i_1)}, s_2^{(i_2)}\right)$ is an equilibrium if and only if the (i_1, i_2) element in matrix $\mathbf{A}^{(1)}$ is the largest in its column, and in matrix $\mathbf{A}^{(2)}$ it is the largest in its row. In the case when $f_2 = -f_1$, the game is called ***zero-sum***, and $\mathbf{A}^{(2)} = -\mathbf{A}^{(1)}$, so the game can be completely described by the payoff matrix $\mathbf{A}^{(1)}$ of the first player. In this special case a strategy vector $(s_1^{(i_1)}, s_2^{(i_2)})$ is an equilibrium if and only if the element (i_1, i_2) is the largest in its column and smallest in its row. In the zero-sum cases the equilibria are also called the ***saddle points*** of the games. Clearly, the enumeration method to find equilibria becomes more simple since we have to deal with a single matrix only.

The simplified algorithm is as follows:

EQUILIBRIUM$\mathbf{A}^{(2)}$

```
1  for i_1 ← 1 to n_1
2      do for i_2 ← 1 to n_2
3         do key ← 0
4         for j ← 1 to n_1
5            do if a^(1)_{j i_2} > a^(1)_{i_1 i_2}
6               then key ← 1
7                    go to 12
8         for j ← 1 to n_2
9            do if a^(2)_{i_1 j} > a^(2)_{i_1 i_2}
10              then key ← 1
11                   go to 12
12        if key = 0
13           then return (s^(1)_{i_1}, s^(2)_{i_2})
```

10.1.2. Games represented by finite trees

Many finite games have the common feature that they can be represented by a finite directed tree with the following properties:

1. there is a unique root of the tree (which is not the endpoint of any arc), and the game starts at this node;

2. to each node of the tree a player is assigned and if the game reaches this node at any time, then this player will decide on the continuation of the game by selecting an arc originating from this node. Then the game moves to the endpoint of the chosen arc;

3. to each terminal node (in which no arc originates) an N-dimensional real vector is assigned which gives the payoff values for the players if the game terminates at this node;

4. each player knows the tree, the nodes he is assigned to, and all payoff values at the terminal nodes.

For example, the chess-game satisfies the above properties in which $N = 2$, the nodes of the tree are all possible configurations on the chessboard twice: once with the white player and once with the black player assigned to it. The arcs represent all possible moves of the assigned player from the originating configurations. The endpoints are those configurations in which the game terminates. The payoff values are from the set $\{1, 0, -1\}$ where 1 means win, -1 represents loss, and 0 shows that the game ends with a tie.

Theorem 10.1 *All games represented by finite trees have at least one equilibrium.*

Proof We present the proof of this result here, since it suggests a practical algorithm to find equilibria. The proof goes by induction with respect to the number of nodes of the game tree. If the game has only one node, then clearly it is the only equilibrium.

Assume next that the theorem holds for any tree with less than n nodes ($n \geq 2$), and consider a game T_0 with n nodes. Let R be the root of the tree and let $r_1, r_2, \ldots, r_m$ ($m < n$) be the nodes connected to R by an arc. If $T_1, T_2, \ldots, T_m$ denote the disjoint subtrees of T_0 with roots $r_1, r_2, \ldots, r_m$, then each subtree has less than n nodes, so each of them has an equilibrium. Assume that player $\mathcal{P}_k$ is assigned to R. Let $e_1, e_2, \ldots, e_m$ be the equilibrium payoffs of player $\mathcal{P}_k$ on the subtrees $T_1, T_2, \ldots, T_m$ and let $e_j = \max\{e_1, e_2, \ldots, e_m\}$. Then player $\mathcal{P}_k$ will move to node r_j from the root, and then the equilibrium continues with the equilibrium obtained on the subtree T_j. We note that not all equilibria can be obtained by this method, however the payoff vectors of all equilibria, which can obtained by this method, are identical. ■

We note that not all equilibria can be obtained by this method, however the payoff vectors of all equilibria, which can be obtained by this method, are identical.

The proof of the theorem suggests a dynamic programming-type algorithm which is called ***backward induction***. It can be extended to the more general case when the tree has chance nodes from which the continuations of the game are random

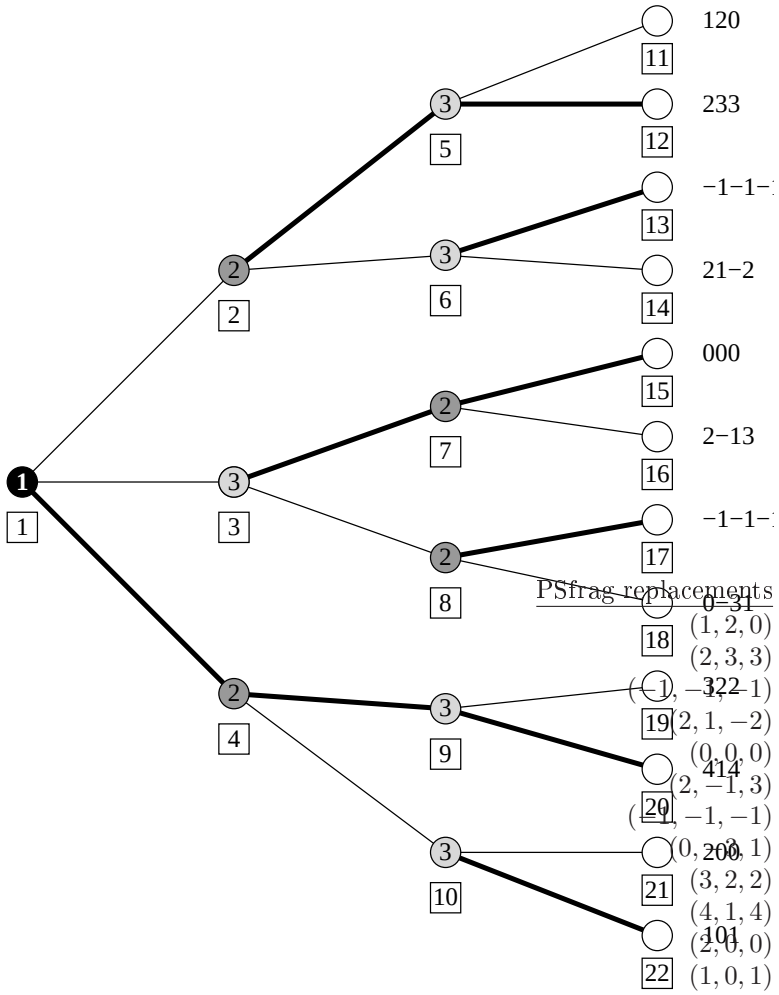


Figure 10.3. Finite tree of Example 10.3.

according to given discrete distributions.

The solution algorithm can be formally presented as follows. Assume that the nodes are numbered so each arc connects nodes i and j only for $i < j$. The root has to get the smallest number 1, and the largest number n is given to one of the terminal nodes. For each node i let $J(i)$ denote the set of all nodes j such that there is an arc from i to j. For each terminal node i, $J(i)$ is empty, and let $\mathbf{p}^{(i)} = (p_1^{(i)}, \ldots, p_N^{(i)})$ denote the payoff vector associated to this node. And finally we will denote player assigned to node i by K_i for all i. The algorithm starts at the last node n and moves backward in the order $n, n-1, n-2, \ldots, 2$ and 1. Node n is an endpoint, so vector $\mathbf{p}^{(n)}$ has been already assigned. If in the process the next node i is an endpoint, then $\mathbf{p}^{(i)}$ is already given, otherwise we find the largest among the values $p_{K_i}^{(j)}$, $j \in J(i)$. Assume that the maximal value occurs at node j_i, then we assign $\mathbf{p}^{(i)} = \mathbf{p}^{(j_i)}$ to node

i, and move to node $i-1$. After all payoff vectors $\mathbf{p}^{(n)}$, $\mathbf{p}^{(n-1)}$, $\ldots, \mathbf{p}^{(2)}$ and $\mathbf{p}^{(1)}$ are determined, then vector $\mathbf{p}^{(1)}$ gives the equilibrium payoffs and the equilibrium path is obtained by nodes:

$$1 \to i_1 = j_1 \to i_2 = j_{i_1} \to i_3 = j_{i_2} \to \ldots,$$

until an endpoint is reached, when the equilibrium path terminates.

At each node the number of comparisons equals the number of arcs starting at that node minus 1. Therefore the total number of comparisons in the algorithm is the total number of arcs minus the number of nodes.

This algorithm can be formally given as follows:

BACKWARD-INDUCTION

1 **for** $i \leftarrow n$ **to** 1
2 **do** $p_{K_i}^{(j_i)} \leftarrow \max\{p_{K_i}^{(l)}, l \in J(i)\}$
3 $\mathbf{p}^{(i)} \leftarrow \mathbf{p}^{(j_i)}$
4 print sequence 1, $i_1(=j_1)$, $i_2(=j_{i_1})$, $i_3(=j_{i_2})$, $\ldots$
until an endpoint is reached

Example 10.3 Figure 10.3 shows a finite tree. In the circle at each nonterminal node we indicate the player assigned to that node. The payoff vectors are also shown at all terminal nodes. We have three players, so the payoff vectors have three elements.

First we number the nodes such that the beginning of each arc has a smaller number than its endpoint. We indicated these numbers in a box under each node. All nodes i for $i \geq 11$ are terminal nodes, as we start the backward induction with node 10. Since player $\mathcal{P}_3$ is assigned to this node we have to compare the third components of the payoff vectors $(2,0,0)$ and $(1,0,1)$ associated to the endpoints of the two arcs originating from node 10. Since $1 > 0$, player $\mathcal{P}_3$ will select the arc to node 22 as his best choice. Hence $j_{10} = 22$, and $\mathbf{p}^{(10)} = \mathbf{p}^{(22)} = (1,0,1)$. Then we check node 9. By comparing the third components of vectors $\mathbf{p}^{(19)}$ and $\mathbf{p}^{(20)}$ it is clear that player $\mathcal{P}_3$ will select node 20, so $j_9 = 20$, and $\mathbf{p}^{(9)} = \mathbf{p}^{(20)} = (4,1,4)$. In the graph we also indicated the choices of the players by thicker arcs. Continuing the procedure in the same way for nodes $8, 7, \ldots, 1$ we finally obtain the payoff vector $\mathbf{p}^{(1)} = (4,1,4)$ and equilibrium path $1 \to 4 \to 9 \to 20$.

Exercises

10.1-1 An entrepreneur (E) enters to a market, which is controlled by a chain store (C). Their competition is a two-person game. The strategies of the chain store are soft (S), when it allows the competitor to operate or tough (T), when it tries to drive out the competitor. The strategies of the entrepreneur are staying in (I) or leaving (L) the market. The payoff tables of the two player are assumed to be

	I	L
S	2	5
T	0	5

payoffs of C

	I	L
S	2	1
T	0	1

payoffs of E

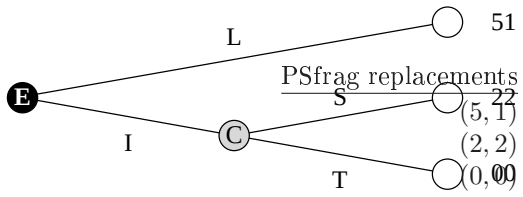


Figure 10.4. Tree for Exercise 10.1-5

Find the equilibrium.

10.1-2 A salesman sells an equipment to a buyer, which has 3 parts, under the following conditions. If all parts are good, then the customer pays $\$\alpha$ to the salesman, otherwise the salesman has to pay $\$\beta$ to the customer. Before selling the equipment, the salesman is able to check any one or more of the parts, but checking any one costs him $\$\gamma$. Consider a two-person game in which player $\mathcal{P}_1$ is the salesman with strategies $0, 1, 2, 3$ (how many parts he checks before selling the equipment), and player $\mathcal{P}_2$ is the equipment with strategies $0, 1, 2, 3$ (how many parts are defective). Show that the payoff matrix of player $\mathcal{P}_1$ is given as below when we assume that the different parts can be defective with equal probability.

		player $\mathcal{P}_2$			
		0	1	2	3
player $\mathcal{P}_1$	0	α	$-\beta$	$-\beta$	$-\beta$
	1	$\alpha-\gamma$	$-\frac{2}{3}\beta-\gamma$	$-\frac{1}{3}\beta-\gamma$	$-\gamma$
	2	$\alpha-2\gamma$	$-\frac{1}{3}\beta-\frac{5}{3}\gamma$	$-\frac{4}{3}\gamma$	$-\gamma$
	3	$\alpha-3\gamma$	-2γ	$-\frac{4}{3}\gamma$	$-\gamma$

10.1-3 Assume that in the previous problem the payoff of the second player is the negative of the payoff of the salesman. Give a complete description of the number of equilibria as a function of the parameter values α, β, γ. Determine the equilibria in all cases.

10.1-4 Assume that the payoff function of the equipment is its value (V if all parts are good, and zero otherwise) in the previous exercise. Is there an equilibrium point?

10.1-5 Exercise 10.1-1 can be represented by the tree shown in Figure 10.4. Find the equilibrium with backward induction.

10.1-6 Show that in the one-player case backward induction reduces to the classical dynamic programming method.

10.1-7 Assume that in the tree of a game some nodes are so called "chance nodes" from which the game continuous with given probabilities assigned to the possible next nodes. Show the existence of the equilibrium for this more general case.

10.1-8 Consider the tree given in Figure 10.3, and double the payoff values of player $\mathcal{P}_1$, change the sign of the payoff values of player $\mathcal{P}_2$, and do not change those for Player $\mathcal{P}_3$. Find the equilibrium of this new game.

10.2. Continuous games

If the strategy sets S_k are connected subsets of finite dimensional Euclidean Spaces and the payoff functions are continuous, then the game is considered continuous.

10.2.1. Fixed-point methods based on best responses

It is very intuitive and usefull from algorithmic point of view to reformulate the equilibrium concept as follows. For all players $\mathcal{P}_k$ and $\mathbf{s} = (s_1, s_2, \ldots, s_N) \in S = S_1 \times S_2 \times \cdots \times S_N$ define the mapping:

$$\begin{aligned} B_k(\mathbf{s}) = \{s_k \in S_k | f_k(s_1, s_2, \ldots, s_{k-1}, s_k, s_{k+1}, \ldots, s_N) \\ = \max_{t_k \in S_k} f_k(s_1, s_2, \ldots, s_{k-1}, t_k, s_{k+1}, \ldots, s_N)\}, \end{aligned} \tag{10.3}$$

which is the set of the best choices of player $\mathcal{P}_k$ with given strategies $s_1, s_2, \ldots, s_{k-1}$, $s_{k+1}, \ldots, s_N$ of the other players. Note that $B_k(\mathbf{s})$ does not depend on s_k, it depends only on all other strategies $\mathbf{s}_l$, $k \neq l$. There is no guarantee that maximum exists for all $\mathbf{s} \in S_1 \times S_2 \times \cdots \times S_N$. Let $\sum \subseteq S$ be the subset of S such that $B_k(\mathbf{s})$ exists for all k and $\mathbf{s} \in \sum$. A simultaneous strategy vector $\mathbf{s}^\star = (s_1^\star, s_2^\star, \ldots, s_N^\star)$ is an equilibrium if and only if $\mathbf{s}^\star \in \sum$, and $s_k^\star \in B_k(\mathbf{s}^\star)$ for all k. By introducing the ***best reply mapping,*** $\mathbf{B}_k(\mathbf{s}) = (B_1(\mathbf{s}), \ldots, B_N(\mathbf{s}))$ we can further simplify the above reformulation:

Theorem 10.2 *Vector $\mathbf{s}^\star$ is equilibrium if and only if $\mathbf{s}^\star \in \sum$ and $\mathbf{s}^\star \in \mathbf{B}(\mathbf{s}^\star)$.*

Hence we have shown that the equilibrium-problem of N-person games is equivalent to find fixed points of certain point-to-set mappings.

The most frequently used existence theorems of equilibria are based on fixed point theorems such as the theorems of Brouwer, Kakutani, Banach, Tarski etc. Any algorithm for finding fixed points can be successfully applied for computing equilibria.

The most popular existence result is a straightforward application of the Kakutani-fixed point theorem.

Theorem 10.3 *Assume that in an N-person game*

1. *the strategy sets S_k are nonempty, closed, bounded, convex subsets of finite dimensional Euclidean spaces;*
   for all k,
2. *the payoff function f_k are continuous on S;*
3. *f_k is concave in s_k with all fixed $s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_N$.*

Then there is at least one equilibrium.

Example 10.4 Consider a 2-person game, $N = 2$, with strategy sets $S_1 = S_2 = [0, 1]$, and payoff functions $f_1(s_1, s_2) = s_1 s_2 - 2s_1^2 + 5$, and $f_2(s_1, s_2) = s_1 s_2 - 2s_2^2 + s_2 + 3$. We will

first find the best responses of both players. Both payoff functions are concave parabolas in their variables with vertices:

$$s_1 = \frac{s_2}{4} \text{ and } s_2 = \frac{s_1 + 1}{4}.$$

For all $s_2 \in [0,1]$ and $s_1 \in [0,1]$ these values are clearly feasible strategies, so

$$B_1(\mathbf{s}) = \frac{s_2}{4} \text{ and } B_2(\mathbf{s}) = \frac{s_1 + 1}{4}.$$

So $(s_1^\star, s_2^\star)$ is equilibrium if and only if it satisfies equations:

$$s_1^\star = \frac{s_2^\star}{4} \text{ and } s_2^\star = \frac{s_1^\star + 1}{4}.$$

It is easy to see that the unique solution is:

$$s_1^\star = \frac{1}{15} \text{ and } s_2^\star = \frac{4}{15},$$

which is therefore the unique equilibrium of the game.

Example 10.5 Consider a portion of a sea-channel, assume it is the unit interval $[0,1]$. Player $\mathcal{P}_2$ is a submarine hiding in location $s_2 \in [0,1]$, player $\mathcal{P}_1$ is an airplane dropping a bomb at certain location $s_1 \in [0,1]$ resulting in a damage $\alpha e^{-\beta(s_1 - s_2)^2}$ to the submarine. Hence a special two-person game is defined in which $S_1 = S_2 = [0,1]$, $f_1(s_1, s_2) = \alpha e^{-\beta(s_1 - s_2)^2}$ and $f_2(s_1, s_2) = -f_1(s_1, s_2)$. With fixed s_2, $f_1(s_1, s_2)$ is maximal if $s_1 = s_2$, therefore the best response of player $\mathcal{P}_1$ is $B_1(\mathbf{s}) = s_2$. Player $\mathcal{P}_2$ wants to minimize f_1 which occurs if $|s_1 - s_2|$ is as large as possible, which implies that

$$B_2(\mathbf{s}) = \begin{cases} 1, & \text{if } s_1 < 1/2, \\ 0, & \text{if } s_1 > 1/2, \\ \{0,1\}, & \text{if } s_1 = 1/2. \end{cases}$$

Clearly, there is no $(s_1, s_2) \in [0,1] \times [0,1]$ such that $s_1 = B_1(\mathbf{s})$ and $s_2 \in B_2(\mathbf{s})$, consequently no equilibrium exists.

10.2.2. Applying Fan's inequality

Define the ***aggregation function*** $H : S \times S \to \mathbb{R}$ as:

$$H_{\mathbf{r}}(\mathbf{s}, \mathbf{z}) = \sum_{k=1}^{N} r_k f_k(s_1, \ldots, s_{k-1}, z_k, s_{k+1}, \ldots, s_N) \tag{10.4}$$

for all $\mathbf{s} = (s_1, \ldots, s_N)$ and $\mathbf{z} = (z_1, \ldots, z_N)$ from S and some $\mathbf{r} = (r_1, r_2, \ldots, r_N) > \mathbf{0}$.

Theorem 10.4 *Vector $\mathbf{s}^\star \in S$ is an equilibrium if and only if*

$$H_{\mathbf{r}}(\mathbf{s}^\star, \mathbf{z}) \leq H_{\mathbf{r}}(\mathbf{s}^\star, \mathbf{s}^\star) \tag{10.5}$$

for all $\mathbf{z} \in S$.

Proof Assume first that $\mathbf{s}^\star$ is an equilibrium, then inequality (10.1) holds for all k and $s_k \in S_k$. Adding the r_k-multiples of these relations for $k = 1, 2, \ldots, N$ we immediately have (10.5).

Assume next that (10.5) holds for all $\mathbf{z} \in S$. Select any k and $s_k \in S_k$, define $\mathbf{z} = (s_1^\star, \ldots, s_{k-1}^\star, s_k, s_{k+1}^\star, \ldots, s_N^\star)$, and apply inequality (10.5). All but the k-th terms cancel and the remaining term shows that inequality (10.1) holds. Hence $\mathbf{s}^\star$ is an equilibrium. ■

Introduce function $\phi(\mathbf{s}, \mathbf{z}) = H_\mathbf{r}(\mathbf{s}, \mathbf{z}) - H_\mathbf{r}(\mathbf{s}, \mathbf{s})$, then clearly $\mathbf{s}^\star \in S$ is an equilibrium if and only if

$$\phi(\mathbf{s}^\star, \mathbf{z}) \leq 0 \text{ for all } \mathbf{z} \in S. \tag{10.6}$$

Relation (10.6) is known as ***Fan's inequality*** . It can be rewritten as a variational inequality (see Section 10.2.9 later), or as a fixed point problem. We show here the second approach. For all $\mathbf{s} \in S$ define

$$\Phi(\mathbf{s}) = \{\mathbf{z} | \mathbf{z} \in S,\ \phi(\mathbf{s}, \mathbf{z}) = \max_{\mathbf{t} \in S} \phi(\mathbf{s}, \mathbf{t})\}. \tag{10.7}$$

Since $\phi(\mathbf{s}, \mathbf{s}) = 0$ for all $\mathbf{s} \in S$, relation (10.6) holds if and only if $\mathbf{s}^\star \in \Phi(\mathbf{s}^\star)$, that is $\mathbf{s}^\star$ is a fixed-point of mapping $\Phi : S \to 2^S$. Therefore any method to find fixed point is applicable for computing equilibria.

The computation cost depends on the type and size of the fixed point problem and also on the selected method.

Example 10.6 Consider again the problem of Example 10.4. In this case

$$f_1(z_1, s_2) = z_1 s_2 - 2z_1^2 + 5 \ ,$$

$$f_2(s_1, z_2) = s_1 z_2 - 2z_2^2 + z_2 + 3 \ ,$$

so the aggregate function has the form with $r_1 = r_2 = 1$:

$$H_\mathbf{r}(\mathbf{s}, \mathbf{z}) = z_1 s_2 - 2z_1^2 + s_1 z_2 - 2z_2^2 + z_2 + 8 \ .$$

Therefore

$$H_\mathbf{r}(\mathbf{s}, \mathbf{s}) = 2s_1 s_2 - 2s_1^2 - 2s_2^2 + s_2 + 8 \ ,$$

and

$$\phi(\mathbf{s}, \mathbf{z}) = z_1 s_2 - 2z_1^2 + s_1 z_2 - 2z_2^2 + z_2 - 2s_1 s_2 + 2s_1^2 + 2s_2^2 - s_2 \ .$$

Notice that this function is strictly concave in z_1 and z_2, and is separable in these variables. At the stationary points:

$$\frac{\partial \phi}{\partial z_1} = s_2 - 4z_1 = 0$$

$$\frac{\partial \phi}{\partial z_2} = s_1 - 4z_2 + 1 = 0$$

implying that at the optimum

$$z_1 = \frac{s_2}{4} \text{ and } z_2 = \frac{s_1 + 1}{4} \ ,$$

since both right hand sides are feasible. At the fixed point:

$$s_1 = \frac{s_2}{4} \text{ and } s_2 = \frac{s_1 + 1}{4} \ ,$$

giving the unique solution:

$$s_1 = \frac{1}{15} \text{ and } s_2 = \frac{4}{15} .$$

10.2.3. Solving the Kuhn-Tucker conditions

Assume that for all k,

$$S_k = \{s_k | \mathbf{g}_k(s_k) \geq \mathbf{0}\} ,$$

where $\mathbf{g}_k : \mathbb{R}^{n_k} \to \mathbb{R}^{m_k}$ is a vector variable vector valued function which is continuously differentiable in an open set O_k containing S_k. Assume furthermore that for all k, the payoff function f_k is continuously differentiable in s_k on O_k with any fixed $s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_N$.

If $\mathbf{s}^\star = (s_1^\star, \ldots, s_N^\star)$ is an equilibrium, then for all k, $s_k^\star$ is the optimal solution of problem:

$$\begin{array}{ll} \text{maximize} & f_k(s_1^\star, \ldots, s_{k-1}^\star, s_k, s_{k+1}^\star, \ldots, s_N^\star) \\ \text{sugject to} & \mathbf{g}_k(s_k) \geq \mathbf{0}. \end{array} \tag{10.8}$$

By assuming that at s_k the Kuhn-Tucker regularity condition is satisfied, the solution has to satisfy the Kuhn-Tucker necessary condition:

$$\begin{array}{rcl} \mathbf{u}_k & \geq & \mathbf{0} \\ \mathbf{g}_k(s_k) & \geq & \mathbf{0} \\ \nabla_k f_k(\mathbf{s}) + \mathbf{u}_k^T \nabla_k \mathbf{g}_k(s_k) & = & \mathbf{0}^T \\ \mathbf{u}_k^T \mathbf{g}_k(s_k) & = & 0 , \end{array} \tag{10.9}$$

where $\mathbf{u}_k$ is an m_k-element column vector, $\mathbf{u}_k^T$ is its transpose, $\nabla_k f_k$ is the gradient of f_k (as a row vector) with respect to s_k and $\nabla_k \mathbf{g}_k$ is the Jacobian of function g_k.

Theorem 10.5 *If $\mathbf{s}^\star$ is an equilibrium, then there are vectors $\mathbf{u}_k^\star$ such that relations (10.9) are satisfied.*

Relations (10.9) for $k = 1, 2, \ldots, N$ give a (usually large) system of equations and inequalities for the unknowns s_k and $\mathbf{u}_k$ ($k = 1, 2, \ldots, N$). Any equilibrium (if exists) has to be among the solutions. If in addition for all k, all components of $\mathbf{g}_k$ are concave, and f_k is concave in s_k, then the Kuhn-Tucker conditions are also sufficient, and therefore all solutions of (10.9) are equilibria.

The computation cost in solving system (10.9) depends on its type and the chosen method.

Example 10.7 Consider again the two-person game of the previous example. Clearly,

$$S_1 = \{s_1 | s_1 \geq 0,\ 1 - s_1 \geq 0\},$$

$$S_2 = \{s_2 | s_2 \geq 0,\ 1 - s_2 \geq 0\},$$

so we have

$$\mathbf{g}_1(s_1) = \begin{pmatrix} s_1 \\ 1 - s_1 \end{pmatrix} \text{ and } \mathbf{g}_2(s_2) = \begin{pmatrix} s_2 \\ 1 - s_2 \end{pmatrix}.$$

Simple differentiation shows that

$$\nabla_1 \mathbf{g}_1(s_1) = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \quad \nabla_2 \mathbf{g}_2(s_2) = \begin{pmatrix} 1 \\ -1 \end{pmatrix},$$

$$\nabla_1 f_1(s_1, s_2) = s_2 - 4s_1, \quad \nabla_2 f_2(s_1, s_2) = s_1 - 4s_2 + 1,$$

therefore the Kuhn-Tucker conditions can be written as follows:

$$\begin{aligned}
u_1^{(1)}, u_2^{(1)} &\geq 0 \\
s_1 &\geq 0 \\
s_1 &\leq 1 \\
s_2 - 4s_1 + u_1^{(1)} - u_2^{(1)} &= 0 \\
u_1^{(1)} s_1 + u_2^{(1)}(1 - s_1) &= 0 \\
u_1^{(2)}, u_2^{(2)} &\geq 0 \\
s_2 &\geq 0 \\
s_2 &\leq 1 \\
s_1 - 4s_2 + 1 + u_1^{(2)} - u_2^{(2)} &= 0 \\
u_1^{(2)} s_2 + u_2^{(2)}(1 - s_2) &= 0.
\end{aligned}$$

Notice that f_1 is concave in s_1, f_2 is concave in s_2, and all constraints are linear, therefore all solutions of this equality-inequality system are really equilibria. By systematically examining the combination of cases

$$s_1 = 0, \quad 0 < s_1 < 1, \quad s_1 = 1,$$

and

$$s_2 = 0, \quad 0 < s_2 < 1, \quad s_2 = 1,$$

it is easy to see that there is a unique solution

$$u_1^{(1)} = u_1^{(2)} = u_2^{(1)} = u_2^{(2)} = 0, \quad s_1 = \frac{1}{15}, \quad s_2 = \frac{4}{15}.$$

By introducing slack and surplus variables the Kuhn-Tucker conditions can be rewritten as a system of equations with some nonnegative variables. The nonnegativity conditions can be formally eliminated by considering them as squares of some new variables, so the result becomes a system of (usually) nonlinear equations without additional constraints. There is a large set of numerical methods for solving such systems.

10.2.4. Reduction to optimization problems

Assume that all conditions of the previous section hold. Consider the following optimization problem:

$$\begin{array}{ll}
\text{minimize} & \sum_{k=1}^{N} \mathbf{u}_k^T \mathbf{g}_k(s_k) \\
\text{subjective to} & \mathbf{u}_k \geq \mathbf{0} \\
& \mathbf{g}_k(s_k) \geq \mathbf{0} \\
& \nabla_k f_k(\mathbf{s}) + \mathbf{u}_k^T \nabla_k \mathbf{g}_k(s_k) = 0.
\end{array} \tag{10.10}$$

The two first constraints imply that the objective function is nonnegative, so is the minimal value of it. Therefore system (10.9) has feasible solution if and only if the optimal value of the objective function of problem (10.10) is zero, and in this case any optimal solution satisfies relations (10.9).

Theorem 10.6 *The N-person game has equilibrium only if the optimal value of the objective function is zero. Then any equilibrium is optimal solution of problem* (10.10). *If in addition all components of $\mathbf{g}_k$ are concave and f_k is concave in s_k for all k, then any optimal solution of problem* (10.10) *is equilibrium.*

Hence the equilibrium problem of the N-person game has been reduced to finding the optimal solutions of this (usually nonlinear) optimization problem. Any nonlinear programming method can be used to solve the problem.

The computation cost in solving the optimization problem (10.10) depends on its type and the chosen method. For example, if (10.10) is an LP, and solved by the simplex method, then the maximum number of operations is exponential. However in particular cases the procedure terminates with much less operations.

Example 10.8 In the case of the previous problem the optimization problem has the following form:

$$\begin{aligned}
\text{minimize} \quad & u_1^{(1)}s_1 + u_2^{(1)}(1-s_1) + u_1^{(2)}s_2 + u_2^{(2)}(1-s_2) \\
\text{subject to} \quad & u_1^{(1)}, u_1^{(2)}, u_2^{(1)}, u_2^{(2)} \geq 0 \\
& s_1 \geq 0 \\
& s_1 \leq 1 \\
& s_2 \geq 0 \\
& s_2 \leq 1 \\
& s_2 - 4s_1 + u_1^{(1)} - u_2^{(1)} = 0 \\
& s_1 - 4s_2 + 1 + u_1^{(2)} - u_2^{(2)} = 0.
\end{aligned}$$

Notice that the solution $u_1^{(1)} = u_1^{(2)} = u_2^{(1)} = u_2^{(2)} = 0$, $s_1 = 1/15$ and $s_2 = 4/15$ is feasible with zero objective function value, so it is also optimal. Hence it is a solution of system (10.9) and consequently an equilibrium.

Mixed Extension of Finite Games We have seen earlier that a finite game does not necessary have equilibrium. Even if it does, in the case of repeating the game many times the players wish to introduce some randomness into their actions in order to make the other players confused and to seek an equilibrium in the stochastic sense. This idea can be modeled by introducing probability distributions as the strategies of the players and the expected payoff values as their new payoff functions.

Keeping the notation of Section 10.1. assume that we have N players, the finite strategy set of player $\mathcal{P}_k$ is $S_k = \{s_k^{(1)}, \ldots, s_k^{(n_k)}\}$. In the mixed extension of this finite game each player selects a discrete probability distribution on its strategy set and in each realization of the game an element of S_k is chosen according to the

selected distribution. Hence the new strategy set of player $\mathcal{P}_k$ is

$$\overline{S}_k = \{\mathbf{x}_k | \mathbf{x}_k = (x_k^{(1)}, \ldots, x_k^{(n_k)}),\ \sum_{i=1}^{n_k} x_k^{(i)} = 1,\ x_k^{(i)} \geq 0 \text{ for all } i\}, \tag{10.11}$$

which is the set of all n_k-element probability vectors. The new payoff function of this player is the expected value:

$$\overline{f}_k(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \ldots \sum_{i_N=1}^{n_N} f_k(s_1^{(i_1)}, s_2^{(i_2)}, \ldots, s_N^{(i_N)}) x_1^{(i_1)} x_2^{(i_2)} \ldots x_N^{(i_N)}. \tag{10.12}$$

Notice that the original "pure" strategies $s_k^{(i)}$ can be obtained by selecting $\mathbf{x}_k$ as the k-th basis vector. This is a continuous game and as a consequence of Theorem 10.3 it has at least one equilibrium. Hence if a finite game is without an equilibrium, its mixed extension has always at least one equilibrium, which can be obtained by using the methods outlined in the previous sections.

Example 10.9 Consider the two-person case in which $N = 2$, and as in section 10.1 introduce matrices $\mathbf{A}^{(1)}$ and $\mathbf{A}^{(2)}$ with (i, j) elements $f_1(s_1^{(i)}, s_2^{(j)})$ and $f_2(s_1^{(i)}, s_2^{(j)})$. In this special case

$$\overline{f}_k(\mathbf{x}_1, \mathbf{x}_2) = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} a_{ij}^{(k)} x_i^{(1)} x_j^{(2)} = \mathbf{x}_1^T \mathbf{A}^{(k)} \mathbf{x}_2\ . \tag{10.13}$$

The constraints of $\overline{S}_k$ can be rewritten as:

$$\begin{aligned} x_k^{(i)} &\geq 0 \qquad (i = 1, 2, \ldots, n_k), \\ -1 + \sum_{i=1}^{n_k} x_k^{(i)} &\geq 0, \\ 1 - \sum_{i=1}^{n_k} x_k^{(i)} &\geq 0. \end{aligned}$$

so we may select

$$\mathbf{g}_k(\mathbf{x}_k) = \begin{pmatrix} x_k^{(1)} \\ \vdots \\ x_k^{(n_k)} \\ \sum_{i=1}^{n_k} x_k^{(i)} - 1 \\ -\sum_{i=1}^{n_k} x_k^{(i)} + 1 \end{pmatrix}. \tag{10.14}$$

The optimization problem (10.10) now reduces to the following:

$$\begin{array}{ll} \text{minimize} & \sum_{k=1}^{2} [\sum_{i=1}^{n_k} u_k^{(i)} x_k^{(i)} + u_k^{(n_k+1)} (\sum_{j=1}^{n_k} x_k^{(j)} - 1) + u_k^{(n_k+2)} (-\sum_{j=1}^{n_k} x_k^{(j)} + 1)] \\ \text{subject to} & u_k^{(i)} \geq 0 \quad (1 \leq i \leq n_k + 2) \\ & x_k^{(i)} \geq 0 \quad (1 \leq i \leq n_k) \\ & \mathbf{1}^T \mathbf{x}_k = 1 \\ & \mathbf{x}_2^T (\mathbf{A}^{(1)})^T + \mathbf{v}_1^T + (u_1^{(n_1+1)} - u_1^{(n_1+2)}) \mathbf{1}_1^T = \mathbf{0}_1^T \\ & \mathbf{x}_1^T (\mathbf{A}^{(2)}) + \mathbf{v}_2^T + (u_2^{(n_2+1)} - u_2^{(n_2+2)}) \mathbf{1}_2^T = \mathbf{0}_2^T, \end{array} \tag{10.15}$$

where $\mathbf{v}_k^T = (u_k^{(1)}, \ldots . u_k^{(n_k)})$, $\mathbf{1}_k^T = (1^{(1)}, \ldots, 1^{(n_k)})$ and $\mathbf{0}_k^T = (0^{(1)}, \ldots, 0^{(n_k)})$, $k = 1, 2$.

Notice this is a quadratic optimization problem. Computation cost depends on the selected method. Observe that the problem is usually nonconvex, so there is the possibility of stopping at a local optimum.

Bimatrix games Mixed extensions of two-person finite games are called ***bimatrix games*** . They were already examined in Example 10.9. For notational convenience introduce the simplifying notation:

$$\mathbf{A} = \mathbf{A}^{(1)}, \mathbf{B} = \mathbf{A}^{(2)}, \mathbf{x} = \mathbf{x}_1, \mathbf{y} = \mathbf{x}_2, m = n_1 \text{ and } n = n_2.$$

We will show that problem (10.15) can be rewritten as quadratic programming problem with linear constraints.

Consider the objective function first. Let

$$\alpha = u_1^{(m+2)} - u_1^{(m+1)}, \text{ and } \beta = u_2^{(n+2)} - u_2^{(n+1)},$$

then the objective function can be rewritten as follows:

$$\mathbf{v}_1^T\mathbf{x} + \mathbf{v}_2^T\mathbf{y} - \alpha(\mathbf{1}_m^T\mathbf{x} - 1) - \beta(\mathbf{1}_n^T\mathbf{y} - 1). \tag{10.16}$$

The last two constraints also simplify:

$$\begin{aligned} \mathbf{y}^T\mathbf{A}^T + \mathbf{v}_1^T - \alpha\mathbf{1}_m^T &= \mathbf{0}_m^T, \\ \mathbf{x}^T\mathbf{B} + \mathbf{v}_2^T - \beta\mathbf{1}_n^T &= \mathbf{0}_n^T, \end{aligned}$$

implying that

$$\mathbf{v}_1^T = \alpha\mathbf{1}_m^T - \mathbf{y}^T\mathbf{A}^T \text{ and } \mathbf{v}_2^T = \beta\mathbf{1}_n^T - \mathbf{x}^T\mathbf{B}, \tag{10.17}$$

so we may rewrite the objective function again:

$$(\alpha\mathbf{1}_m^T - \mathbf{y}^T\mathbf{A}^T)\mathbf{x} + (\beta\mathbf{1}_n^T - \mathbf{x}^T\mathbf{B})\mathbf{y} - \alpha(\mathbf{1}_m^T\mathbf{x} - 1) - \beta(\mathbf{1}_n^T\mathbf{y} - 1) = \alpha + \beta - \mathbf{x}^T(\mathbf{A} + \mathbf{B})\mathbf{y},$$

since

$$\mathbf{1}_m^T\mathbf{x} = \mathbf{1}_n^T\mathbf{y} = 1.$$

Hence we have the following quadratic programming problem :

$$\begin{array}{ll} \text{maximize} & \mathbf{x}^T(\mathbf{A} + \mathbf{B})\mathbf{y} - \alpha - \beta \\ \text{subject to} & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{y} \geq \mathbf{0} \\ & \mathbf{1}_m^T\mathbf{x} = 1 \\ & \mathbf{1}_n^T\mathbf{y} = 1 \\ & \mathbf{A}\mathbf{y} \leq \alpha\mathbf{1}_m \\ & \mathbf{B}^T\mathbf{x} \leq \beta\mathbf{1}_n, \end{array} \tag{10.18}$$

where the last two conditions are obtained from (10.17) and the nonnegativity of vectors $\mathbf{v}_1$, $\mathbf{v}_2$.

Theorem 10.7 *Vectors $\mathbf{x}^\star$ and $\mathbf{y}^\star$ are equilibria of the bimatrix game if and only if with some $\alpha^\star$ and $\beta^\star$, $(\mathbf{x}^\star, \mathbf{y}^\star, \alpha^\star, \beta^\star)$ is optimal solution of problem (10.18). The optimal value of the objective function is zero.*

This is a quadratic programming problem. Computation cost depends on the selected method. Since it is usually nonconvex, the algorithm might terminate at local optimum. We know that at the global optimum the objective function must be zero, which can be used for optimality check.

Example 10.10 Select

$$\mathbf{A} = \begin{matrix} 2 & -1 \\ -1 & 1 \end{matrix}$$

and

$$\mathbf{B} = \begin{matrix} 1 & -1 \\ -1 & 2 \end{matrix} .$$

Then

$$\mathbf{A} + \mathbf{B} = \begin{matrix} 3 & -2 \\ -2 & 3 \end{matrix} ,$$

so problem (10.18) has the form:

$$\begin{array}{ll} \text{maximize} & 3x_1y_1 - 2x_1y_2 - 2x_2y_1 + 3x_2y_2 - \alpha - \beta \\ \text{subject to} & x_1, x_2, y_1, y_2 \geq 0 \\ & x_1 + x_2 = 1 \\ & y_1 + y_2 = 1 \\ & 2y_1 - y_2 \leq \alpha \\ & -y_1 + y_2 \leq \alpha \\ & x_1 - x_2 \leq \beta \\ & -x_1 + 2x_2 \leq \beta, \end{array}$$

where $\mathbf{x} = (x_1, x_2)^T$ and $\mathbf{y} = (y_1, y_2)^T$. We also know from Theorem 10.7 that the optimal objective function value is zero, therefore any feasible solution with zero objective function value is necessarily optimal. It is easy to see that the solutions

$$\mathbf{x} = \begin{matrix} 1 \\ 0 \end{matrix}\Bigg), \ \mathbf{y} = \begin{matrix} 0 \\ 1 \end{matrix}\Bigg), \ \alpha = 2, \beta = 1,$$

$$\mathbf{x} = \begin{matrix} 0 \\ 1 \end{matrix}\Bigg), \ \mathbf{y} = \begin{matrix} 1 \\ 0 \end{matrix}\Bigg), \ \alpha = 1, \ \beta = 2,$$

$$\mathbf{x} = \begin{matrix} 0.6 \\ 0.4 \end{matrix}\Bigg), \ \mathbf{y} = \begin{matrix} 0.4 \\ 0.6 \end{matrix}\Bigg), \ \alpha = 0.2, \ \beta = 0.2$$

are all optimal, so they provide equilibria.

One might apply relations (10.9) to find equilibria by solving the equality-inequality system instead of solving an optimization problem. In the case of bimatrix games problem (10.9) simplifies as

$$\begin{array}{rcl} \mathbf{x}^T\mathbf{A}\mathbf{y} & = & \alpha \\ \mathbf{x}^T\mathbf{B}\mathbf{y} & = & \beta \\ \mathbf{A}\mathbf{y} & \leq & \alpha\mathbf{1}_m \\ \mathbf{B}^T\mathbf{x} & \leq & \beta\mathbf{1}_n \\ \mathbf{x} & \geq & \mathbf{0}_m \\ \mathbf{y} & \geq & \mathbf{0}_n \\ \mathbf{1}_m^T\mathbf{x} = \mathbf{1}_n^T\mathbf{y} & = & 1, \end{array} \tag{10.19}$$

which can be proved along the lines of the derivation of the quadratic optimization problem.

The computation cost of the solution of system (10.19) depends on the particular method being selected.

Example 10.11 Consider again the bimatrix game of the previous example. Substitute the first and second constraints $\alpha = \mathbf{x}^T\mathbf{A}\mathbf{y}$ and $\beta = \mathbf{x}^T\mathbf{B}\mathbf{y}$ into the third and fourth condition to have

$$\begin{aligned} 2y_1 - y_2 &\leq 2x_1y_1 - x_1y_2 - x_2y_1 + x_2y_2 \\ -y_1 + y_2 &\leq 2x_1y_1 - x_1y_2 - x_2y_1 + x_2y_2 \\ x_1 - x_2 &\leq x_1y_1 - x_1y_2 - x_2y_1 + 2x_2y_2 \\ -x_1 + 2x_2 &\leq x_1y_1 - x_1y_2 - x_2y_1 + 2x_2y_2 \\ x_1, x_2, y_1, y_2 &\geq 0 \\ x_1 + x_2 = y_1 + y_2 &= 1. \end{aligned}$$

It is easy to see that the solutions given in the previous example solve this system, so they are equilibria.

We can also rewrite the equilibrium problem of bimatrix games as an equality-inequality system with mixed variables. Assume first that all elements of $\mathbf{A}$ and $\mathbf{B}$ are between 0 and 1. This condition is not really restrictive, since by using linear transformations

$$\overline{\mathbf{A}} = a_1\mathbf{A} + b_1\mathbf{1} \quad \text{and} \quad \overline{\mathbf{B}} = a_2\mathbf{B} + b_2\mathbf{1},$$

where $a_1, a_2 > 0$, and $\mathbf{1}$ is the matrix all elements of which equal 1, the equilibria remain the same and all matrix elements can be transformed into interval $[0, 1]$.

Theorem 10.8 *Vectors* $\mathbf{x}$, $\mathbf{y}$ *are an equilibrium if and only if there are real numbers* α, β *and zero-one vectors* $\mathbf{u}$, *and* $\mathbf{v}$ *such that*

$$\begin{array}{ccccccc} 0 & \leq & \alpha\mathbf{1}_m - \mathbf{A}\mathbf{y} & \leq & \mathbf{1}_m - \mathbf{u} & \leq & \mathbf{1}_m - \mathbf{x} \\ 0 & \leq & \beta\mathbf{1}_n - \mathbf{B}^T\mathbf{x} & \leq & \mathbf{1}_n - \mathbf{v} & \leq & \mathbf{1}_n - \mathbf{y} \\ & & & & \mathbf{x} & \geq & \mathbf{0}_m \\ & & & & \mathbf{y} & \geq & \mathbf{0}_n \\ & & \mathbf{1}_m^T\mathbf{x} & = & \mathbf{1}_n^T\mathbf{y} & = & 1, \end{array} \tag{10.20}$$

where $\mathbf{1}$ denotes the vector with all unit elements.

Proof Assume first that $\mathbf{x}$, $\mathbf{y}$ is an equilibrium, then with some α and β, (10.19) is satisfied. Define

$$u_i = \begin{cases} 1, & \text{if } x_i > 0, \\ 0, & \text{if } x_i = 0, \end{cases} \quad \text{and} \quad v_j = \begin{cases} 1, & \text{if } y_j > 0, \\ 0, & \text{if } y_j = 0. \end{cases}$$

Since all elements of $\mathbf{A}$ and $\mathbf{B}$ are between 0 and 1, the values $\alpha = \mathbf{x}^T\mathbf{A}\mathbf{y}$ and $\beta = \mathbf{x}^T\mathbf{B}\mathbf{y}$ are also between 0 and 1. Notice that

$$0 = \mathbf{x}^T(\alpha\mathbf{1}_m - \mathbf{A}\mathbf{y}) = \mathbf{y}^T(\beta\mathbf{1}_n - \mathbf{B}^T\mathbf{x}),$$

which implies that (10.20) holds.

Assume next that (10.20) is satisfied. Then

$$\mathbf{0} \leq \mathbf{x} \leq \mathbf{u} \leq \mathbf{1}_m \text{ and } \mathbf{0} \leq \mathbf{y} \leq \mathbf{v} \leq \mathbf{1}_n.$$

If $u_i = 1$, then $\alpha - \mathbf{e}_i^T \mathbf{A}\mathbf{y} = 0$, (where $\mathbf{e}_i$ is the i-th basis vector), and if $u_i = 0$, then $x_i = 0$. Therefore

$$\mathbf{x}^T(\alpha \mathbf{1}_m - \mathbf{A}\mathbf{y}) = 0,$$

implying that $\alpha = \mathbf{x}^T \mathbf{A}\mathbf{y}$. We can similarly show that $\beta = \mathbf{x}^T \mathbf{B}\mathbf{y}$. Thus (10.19) is satisfied, so, $\mathbf{x}, \mathbf{y}$ is an equilibrium. ■

The computation cost of the solution of system (10.20) depends on the particular method being seleced.

Example 10.12 In the case of the bimatrix game introduced earlier in Example 10.10 we have the following:

$$\begin{array}{ccccccc}
0 & \leq & \alpha - 2y_1 + y_2 & \leq & 1 - u_1 & \leq & 1 - x_1 \\
0 & \leq & \alpha + y_1 - y_2 & \leq & 1 - u_2 & \leq & 1 - x_2 \\
0 & \leq & \beta - x_1 + x_2 & \leq & 1 - v_1 & \leq & 1 - y_1 \\
0 & \leq & \beta + x_1 - 2x_2 & \leq & 1 - v_2 & \leq & 1 - y_2 \\
 & & x_1 + x_2 & = & y_1 + y_2 & = & 1 \\
 & & & & x_1, x_2, y_1, y_2 & \geq & 0 \\
 & & & & u_1, u_2, v_1, v_2 & \in & \{0, 1\}.
\end{array}$$

Notice that all three solutions given in Example 10.10 satisfy these relations with

$$\mathbf{u} = (1, 0), \quad \mathbf{v} = (0, 1)$$

$$\mathbf{u} = (0, 1), \quad \mathbf{v} = (1, 0)$$

and

$$\mathbf{u} = (1, 1), \quad \mathbf{v} = (1, 1),$$

respectively.

Matrix games In the special case of $\mathbf{B} = -\mathbf{A}$, bimatrix games are called ***matrix games*** and they are represented by matrix $\mathbf{A}$. Sometimes we refer to the game as matrix $\mathbf{A}$ game. Since $\mathbf{A} + \mathbf{B} = \mathbf{0}$, the quadratic optimization problem (10.18) becomes linear:

$$\begin{array}{ll}
\text{minimize} & \alpha + \beta \\
\text{subject to} & \mathbf{x} \geq \mathbf{0} \\
 & \mathbf{y} \geq \mathbf{0} \\
 & \mathbf{1}_m \mathbf{x} = 1 \\
 & \mathbf{1}_n \mathbf{y} = 1 \\
 & \mathbf{A}\mathbf{y} \leq \alpha \mathbf{1}_m \\
 & \mathbf{A}^T \mathbf{x} \geq -\beta \mathbf{1}_n.
\end{array} \tag{10.21}$$

From this formulation we see that the set of the equilibrium strategies is a convex polyhedron. Notice that variables $(\mathbf{x}, \beta)$ and $(\mathbf{y}, \alpha)$ can be separated, so we have the following result.

Theorem 10.9 *Vectors $\mathbf{x}^\star$ and $\mathbf{y}^\star$ give an equilibrium of the matrix game if and only if with some $\alpha^\star$ and $\beta^\star$, $(\mathbf{x}^\star, \beta^\star)$ and $(\mathbf{y}^\star, \alpha^\star)$ are optimal solutions of the linear programming problems:*

$$\begin{array}{lll} \textit{minimize} & \alpha & \\ \textit{subject to} & \mathbf{y} \geq \mathbf{0}_n & \\ & \mathbf{1}_n^T \mathbf{y} = 1 & \\ & \mathbf{A}\mathbf{y} \leq \alpha \mathbf{1}_m & \end{array} \qquad \begin{array}{lll} \textit{minimize} & \beta & \\ \textit{subject to} & \mathbf{x} \geq \mathbf{0}_m & \\ & \mathbf{1}_m^T \mathbf{x} = 1 & \\ & \mathbf{A}^T \mathbf{x} \geq -\beta \mathbf{1}_n. & \end{array} \tag{10.22}$$

Notice that at the optimum, $\alpha + \beta = 0$. The optimal α value is called the ***value of the matrix game*** .

Solving problem (10.22) requires exponential number of operations if the simplex method is chosen. With polynomial algorithm (such as the interior point method) the number of operations is only polynomial.

Example 10.13 Consider the matrix game with matrix:

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 0 & 3 \\ -1 & 3 & 3 \end{pmatrix}.$$

In this case problems (10.22) have the from:

$$\begin{array}{lll} \text{minimize} & \alpha & \\ \text{subject to} & y_1, y_2, y_3 \geq 0 & \\ & y_1 + y_2 + y_3 = 1 & \\ & 2y_1 + y_2 - \alpha \leq 0 & \\ & 2y_1 + 3y_3 - \alpha \leq 0 & \\ & -y_1 + 3y_2 + 3y_3 - \alpha \leq 0 & \end{array} \quad \text{and} \quad \begin{array}{lll} \text{minimize} & \beta & \\ \text{subject to} & x_1, x_2, x_3 \geq 0 & \\ & x_1 + x_2 + x_3 = 1 & \\ & 2x_1 + 2x_2 - x_3 + \beta \geq 0 & \\ & x_1 + 3x_3 + \beta \geq 0 & \\ & 3x_2 + 3x_3 + \beta \geq 0. & \end{array}$$

The application of the simplex method shows that the optimal solutions are $\alpha = 9/7$, $\mathbf{y} = (3/7, 3/7, 1/7)^T$, $\beta = -9/7$, and $\mathbf{x} = (4/7, 4/21, 5/21)^T$.

We can also obtain the equilibrium by finding feasible solutions of a certain set of linear constraints. Since at the optimum of problem (10.21), $\alpha + \beta = 0$, vectors $\mathbf{x}$, $\mathbf{y}$ and scalers α and β are optimal solutions if and only if

$$\begin{array}{rcl} \mathbf{x}, \mathbf{y} & \geq & \mathbf{0} \\ \mathbf{1}_m^T \mathbf{x} & = & 1 \\ \mathbf{1}_n^T \mathbf{y} & = & 1 \\ \mathbf{A}\mathbf{y} & \leq & \alpha \mathbf{1}_m \\ \mathbf{A}^T \mathbf{x} & \geq & \alpha \mathbf{1}_n. \end{array} \tag{10.23}$$

The first phase of the simplex method has to be used to solve system (10.23), where the number of operations might be experimental. However in most practical examples much less operations are needed.

Example 10.14 Consider again the matrix game of the previous example. In this case

system (10.23) has the following form:

$$\begin{aligned} x_1, x_2, x_3, y_1, y_2, y_3 &\geq 0 \\ x_1 + x_2 + x_3 = y_1 + y_2 + y_3 &= 1 \\ 2y_1 + y_2 &\leq \alpha \\ 2y_1 + 3y_3 &\leq \alpha \\ -y_1 + 3y_2 + 3y_3 &\leq \alpha \\ 2x_1 + 2x_2 - x_3 &\geq \alpha \\ x_1 + 3x_3 &\geq \alpha \\ 3x_2 + 3x_3 &\geq \alpha. \end{aligned}$$

It is easy to see that $\alpha = 9/7$, $\mathbf{x} = (4/7, 4/21, 5/21)^T$, $\mathbf{y} = (3/7, 3/7, 1/7)^T$ satisfy these relations, so $\mathbf{x}$, $\mathbf{y}$ is an equilibrium.

10.2.5. Method of fictitious play

Consider now a matrix game with matrix $\mathbf{A}$. The main idea of this method is that at each step both players determine their best pure strategy choices against the average strategies of the other player of all previous steps. Formally the method can be described as follows.

Let $\mathbf{x}_1$ be the initial (mixed) strategy of player $\mathcal{P}_1$. Select $\mathbf{y}_1 = \mathbf{e}_{j_1}$ (the j_1st basis vector) such that

$$\mathbf{x}_1^T \mathbf{A} \mathbf{e}_{j_1} = \min_j \{\mathbf{x}_1^T \mathbf{A} \mathbf{e}_j\}. \tag{10.24}$$

In any further step $k \geq 2$, let

$$\overline{\mathbf{y}}_{k-1} = \frac{1}{k-1}((k-2)\overline{\mathbf{y}}_{k-2} + \mathbf{y}_{k-1}), \tag{10.25}$$

and select $\mathbf{x}_k = \mathbf{e}_{i_k}$so that

$$\mathbf{e}_{i_k}^T \mathbf{A} \overline{\mathbf{y}}_{k-1} = \max_i \{\mathbf{e}_i^T \mathbf{A} \overline{\mathbf{y}}_{k-1}\}. \tag{10.26}$$

Let then

$$\overline{\mathbf{x}}_k = \frac{1}{k}((k-1)\overline{\mathbf{x}}_{k-1} + \mathbf{x}_k), \tag{10.27}$$

and select $\mathbf{y}_k = \mathbf{e}_{j_k}$ so that

$$\overline{\mathbf{x}}_k^T \mathbf{A} \mathbf{e}_{j_k} = \min_j \{\overline{\mathbf{x}}_k^T \mathbf{A} \mathbf{e}_j\}. \tag{10.28}$$

By repeating the general step for $k = 2, 3, \ldots$ two sequences are generated: $\{\overline{\mathbf{x}}_k\}$, and $\{\overline{\mathbf{y}}_k\}$. We have the following result:

Theorem 10.10 *Any cluster point of these sequences is an equilibrium of the matrix game. Since all $\overline{\mathbf{x}}_k$ and $\overline{\mathbf{y}}_k$ are probability vectors, they are bounded. Therefore there is at least one cluster point.*

Assume, matrix $\mathbf{A}$ is $m \times n$. In (10.24) we need mn multiplications. In (10.25) and (10.27) $m+n$ multiplications and divisions. In (10.26) and (10.28) mn multiplications. If we make L iteration steps, then the total number of multiplications and divisions is:

$$mn + L[2(m+n) + 2mn] = \ominus(Lmn).$$

The formal algorithm is as follows:

MATRIX-EQUILIBRIUM$(\mathbf{A}, \overline{\mathbf{x}}_1^T)$

1 $k \leftarrow 1$
2 define j_1 such that $\mathbf{x}_1^T \mathbf{A} \mathbf{e}_{j_1} = \min_j \{\mathbf{x}_1^T \mathbf{A} \mathbf{e}_j\}$
3 $\mathbf{y}_1 \leftarrow \mathbf{e}_{j_1}$
4 $k \leftarrow k+1$
5 $\overline{\mathbf{y}}_{k-1} \leftarrow \frac{1}{k-1}((k-2)\overline{\mathbf{y}}_{k-2} + \mathbf{y}_{k-1})$
6 define i_k such that $\mathbf{e}_{i_k}^T \mathbf{A} \overline{\mathbf{y}}_{k-1} = \max_i \{\mathbf{e}_i^T \mathbf{A} \overline{\mathbf{y}}_{k-1}\}$
7 $\mathbf{x}_k \leftarrow \mathbf{e}_{i_k}$
8 $\overline{\mathbf{x}}_k \leftarrow \frac{1}{k}((k-1)\overline{\mathbf{x}}_{k-1} + \mathbf{x}_k)$
9 define j_k such that $\overline{\mathbf{x}}_k^T \mathbf{A} \mathbf{e}_{j_k} = \min_j \{\overline{\mathbf{x}}_k^T \mathbf{A} \mathbf{e}_j\}$
10 $\mathbf{y}_k \leftarrow \mathbf{e}_{j_k}$
11 **if** $||\overline{\mathbf{x}}_k - \overline{\mathbf{x}}_{k-1}|| < \varepsilon$ and $||\overline{\mathbf{y}}_{k-1} - \overline{\mathbf{x}}_{k-2}|| < \varepsilon$
12 **then** $(\overline{\mathbf{x}}_k, \overline{\mathbf{y}}_{k-1})$ is equilibrium
13 **else** go back to 4

Here $\varepsilon > 0$ is a user selected error tolerance.

Example 10.15 We applied the above method for the matrix game of the previous example and started the procedure with $\mathbf{x}_1 = (1,0,0)^T$. After 100 steps we obtained $\overline{\mathbf{x}}_{101} = (0.446, 0.287, 0.267)^T$ and $\overline{\mathbf{y}}_{101} = (0.386, 0.436, 0.178)^T$. Comparing it to the true values of the equilibrium strategies we see that the error is below 0.126, showing the very slow convergence of the method.

10.2.6. Symmetric matrix games

A matrix game with skew-symmetric matrix is called symmetric. In this case $\mathbf{A}^T = -\mathbf{A}$ and the two linear programming problems are identical. Therefore at the optimum $\alpha = \beta = 0$, and the equilibrium strategies of the two players are the same. Hence we have the following result:

Theorem 10.11 *A vector $\mathbf{x}^\star$ is equilibrium of the symmetric matrix game if and only if*

$$\begin{aligned} \mathbf{x} &\geq \mathbf{0} \\ \mathbf{1}^T\mathbf{x} &= 1 \\ \mathbf{A}\mathbf{x} &\leq \mathbf{0}. \end{aligned} \tag{10.29}$$

Solving system (10.29) the first phase of the simplex method is needed, the number of operations is exponential in the worst case but in practical case usually much less.

Example 10.16 Consider the symmetric matrix game with matrix $\mathbf{A} = \begin{matrix} 0 & 1 \\ -1 & 0 \end{matrix}$. In this case relations (10.29) simplify as follows:

$$\begin{aligned} x_1, x_2 &\geq 0 \\ x_1 + x_2 &= 1 \\ x_2 &\leq 0 \\ -x_1 &\leq 0. \end{aligned}$$

Clearly the only solution is $x_1 = 1$ and $x_2 = 0$, that is the first pure strategy.

We will see in the next subsection that linear programming problems are equivalent to symmetric matrix games so any method for solving such games can be applied to solve linear programming problems, so they serve as alternative methodology to the simplex method. As we will see next, symmetry is not a strong assumption, since any matrix game is equivalent to a symmetric matrix game.

Consider therefore a matrix game with matrix $\mathbf{A}$, and construct the skew-symmetric matrix

$$\mathbf{P} = \begin{pmatrix} \mathbf{0}_{m\times m} & \mathbf{A} & -\mathbf{1}_m \\ -\mathbf{A}^T & \mathbf{0}_{n\times n} & \mathbf{1}_n \\ \mathbf{1}_m^T & -\mathbf{1}_n^T & 0 \end{pmatrix},$$

where all components of vector $\mathbf{1}$ equal 1. Matrix games $\mathbf{A}$ and $\mathbf{P}$ are equivalent in the following sense. Assume that $\mathbf{A} > \mathbf{0}$, which is not a restriction, since by adding the same constant to all element of $\mathbf{A}$ they become positive without changing equilibria.

Theorem 10.12

1. *If* $\mathbf{z} = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \lambda \end{pmatrix}$ *is an equilibrium strategy of matrix game* $\mathbf{P}$ *then with* $\mathbf{a} = (1-\lambda)/2$, $\mathbf{x} = (1/a)\mathbf{u}$ *and* $\mathbf{y} = (1/a)\mathbf{v}$ *is an equilibrium of matrix game* $\mathbf{A}$ *with value* $v = \lambda/a$*;*

2. *If* $\mathbf{x}, \mathbf{y}$ *is an equilibrium of matrix game* $\mathbf{A}$ *and* v *is the value of the game, then*

$$\mathbf{z} = \frac{1}{2+v}\begin{pmatrix} \mathbf{x} \\ \mathbf{y} \\ v \end{pmatrix}$$

is equilibrium strategy of matrix game $\mathbf{P}$*.*

Proof Assume first that $\mathbf{z}$ is an equilibrium strategy of game $\mathbf{P}$, then $\mathbf{u} \geq \mathbf{0}$, $\mathbf{v} \geq \mathbf{0}$, $\mathbf{Pz} \leq \mathbf{0}$, so

$$\begin{aligned} \mathbf{A}\mathbf{v} - \lambda\mathbf{1}_m &\leq \mathbf{0} \\ -\mathbf{A}^T\mathbf{u} + \lambda\mathbf{1}_n &\leq \mathbf{0} \\ \mathbf{1}_m^T\mathbf{u} - \mathbf{1}_n^T\mathbf{v} &\leq 0. \end{aligned} \tag{10.30}$$

First we show that $0 < \lambda < 1$, that is $a \neq 0$. If $\lambda = 1$, then (since $\mathbf{z}$ is a probability vector) $\mathbf{u} = \mathbf{0}$ and $\mathbf{v} = \mathbf{0}$, contradicting the second inequality of (10.30). If $\lambda = 0$,

then $\mathbf{1}_m^T\mathbf{u} + \mathbf{1}_n^T\mathbf{v} = 1$, and by the third inequality of (10.30), $\mathbf{v}$ must have at least one positive component which makes the first inequality impossible.

Next we show that $\mathbf{1}^T\mathbf{u} = \mathbf{1}^T\mathbf{v}$. From (10.30) we have

$$\begin{aligned} \mathbf{u}^T\mathbf{A}\mathbf{v} - \lambda\mathbf{u}^T\mathbf{1}_m &\leq 0, \\ -\mathbf{v}^T\mathbf{A}^T\mathbf{u} + \lambda\mathbf{u}^T\mathbf{1}_n &\leq 0 \end{aligned}$$

and by adding these inequalities we see that

$$\mathbf{v}^T\mathbf{1}_n - \mathbf{u}^T\mathbf{1}_m \leq 0,$$

and combining this relation with the third inequality of (10.30) we see that $\mathbf{1}_m^T\mathbf{u} - \mathbf{1}_n^T\mathbf{v} = 0$.

Select $a = (1-\lambda)/2 \neq 0$, then $\mathbf{1}_m^T\mathbf{u} = \mathbf{1}_n^T\mathbf{v} = a$, so both $\mathbf{x} = \mathbf{u}/a$, and $\mathbf{y} = \mathbf{v}/a$ are probability vectors, furthermore from (10.30),

$$\begin{aligned} \mathbf{A}^T\mathbf{x} &= \tfrac{1}{a}\mathbf{A}^T\mathbf{u} \geq \tfrac{\lambda}{a}\mathbf{1}_n, \\ \mathbf{A}\mathbf{y} &= \tfrac{1}{a}\mathbf{A}\mathbf{v} \leq \tfrac{\lambda}{a}\mathbf{1}_m. \end{aligned}$$

So by selecting $\alpha = \lambda/a$ and $\beta = -\lambda/a$, $\mathbf{x}$ and $\mathbf{y}$ are feasible solutions of the pair (10.22) of linear programming problems with $\alpha + \beta = 0$, therefore $\mathbf{x}, \mathbf{y}$ is an equilibrium of matrix game $\mathbf{A}$. Part 2. can be proved in a similar way, the details are not given here. ∎

10.2.7. Linear programming and matrix games

In this section we will show that linear programming problems can be solved by finding the equilibrium strategies of symmetric matrix games and hence, any method for finding the equilibria of symmetric matrix games can be applied instead of the simplex method.

Consider the primal-dual linear programming problem pair:

$$\begin{array}{lllll} \text{maximize} & \mathbf{c}^T\mathbf{x} & \text{and} & \text{minimize} & \mathbf{b}^T\mathbf{y} \\ \text{subject to} & \mathbf{x} \geq \mathbf{0} & & \text{subject to} & \mathbf{y} \geq \mathbf{0} \\ & \mathbf{A}\mathbf{x} \leq \mathbf{b} & & & \mathbf{A}^T\mathbf{y} \geq \mathbf{c}. \end{array} \tag{10.31}$$

Construct the skew-symmetric matrix:

$$\mathbf{P} = \begin{pmatrix} \mathbf{0} & \mathbf{A} & -\mathbf{b} \\ -\mathbf{A}^T & \mathbf{0} & \mathbf{c} \\ \mathbf{b}^T & -\mathbf{c}^T & 0 \end{pmatrix}.$$

Theorem 10.13 *Assume* $\mathbf{z} = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \lambda \end{pmatrix}$ *is an equilibrium strategy of the symmetric matrix game* $\mathbf{P}$ *with* $\lambda > 0$. *Then*

$$\mathbf{x} = \frac{1}{\lambda}\mathbf{v} \quad \textit{and} \quad \mathbf{y} = \frac{1}{\lambda}\mathbf{u}$$

are optimal solutions of the primal and dual problems, respectively.

Proof If $\mathbf{z}$ is an equilibrium strategy, then $\mathbf{Pz} \leq \mathbf{0}$, that is,

$$\begin{array}{rcl} \mathbf{Av} - \lambda\mathbf{b} & \leq & \mathbf{0} \\ -\mathbf{A}^T\mathbf{u} + \lambda\mathbf{c} & \leq & \mathbf{0} \\ \mathbf{b}^T\mathbf{u} - \mathbf{c}^T\mathbf{v} & \leq & 0. \end{array} \tag{10.32}$$

Since $\mathbf{z} \geq \mathbf{0}$ and $\lambda > 0$, both vectors $\mathbf{x} = (1/\lambda)\mathbf{v}$, and $\mathbf{y} = (1/\lambda)\mathbf{u}$ are nonnegative, and by dividing the first two relations of (10.32) by λ,

$$\mathbf{Ax} \leq \mathbf{b} \quad \text{and} \quad \mathbf{A}^T\mathbf{y} \geq \mathbf{c},$$

showing that $\mathbf{x}$ and $\mathbf{y}$ are feasible for the primal and dual, respectively. From the last condition of (10.32) we have

$$\mathbf{b}^T\mathbf{y} \leq \mathbf{c}^T\mathbf{x}.$$

However

$$\mathbf{b}^T\mathbf{y} \geq (\mathbf{x}^T\mathbf{A}^T)\mathbf{y} = \mathbf{x}^T(\mathbf{A}^T\mathbf{y}) \geq \mathbf{x}^T\mathbf{c} = \mathbf{c}^T\mathbf{x},$$

consequently, $\mathbf{b}^T\mathbf{y} = \mathbf{c}^T\mathbf{x}$, showing that the primal and dual objective functions are equal. The duality theorem implies the optimality of $\mathbf{x}$ and $\mathbf{y}$. ■

Example 10.17 Consider the linear programming problem:

$$\begin{array}{ll} \text{maximize} & x_1 + 2x_2 \\ \text{subject to} & x_1 \geq 0 \\ & -x_1 + x_2 \geq 1 \\ & 5x_1 + 7x_2 \leq 25. \end{array}$$

First we have to rewrite the problem as a primal problem. Introduce the new variables:

$$x_2^+ = \begin{array}{ll} x_2, & \text{if } x_2 \geq 0, \\ 0 & \text{otherwise,} \end{array}$$

$$x_2^- = \begin{array}{ll} -x_2, & \text{if } x_2 < 0, \\ 0 & \text{otherwise.} \end{array}$$

and multiply the $\geq$-type constraint by -1. Then the problem becomes the following:

$$\begin{array}{ll} \text{maximize} & x_1 + 2x_2^+ - 2x_2^- \\ \text{subject to} & x_1, x_2^+, x_2^- \geq 0 \\ & x_1 - x_2^+ + x_2^- \leq -1 \\ & 5x_1 + 7x_2^+ - 7x_2^- \leq 25. \end{array}$$

Hence

$$\mathbf{A} = \begin{array}{ccc} 1 & -1 & 1 \\ 5 & 7 & -7 \end{array}, \quad \mathbf{b} = \begin{array}{c} -1 \\ 25 \end{array}\Bigg), \quad \mathbf{c}^T = (1, 2, -2),$$

and so matrix $\mathbf{P}$ becomes:

$$\mathbf{P} = \left(\begin{array}{cc:ccc:c} 0 & 0 & 1 & -1 & 1 & 1 \\ 0 & 0 & 5 & 7 & -7 & -25 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ -1 & -5 & 0 & 0 & 0 & 1 \\ 1 & -7 & 0 & 0 & 0 & 2 \\ -1 & 7 & 0 & 0 & 0 & -2 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ -1 & 25 & -1 & -2 & 2 & 0 \end{array}\right) .$$

10.2.8. The method of von Neumann

The fictitious play method is an iteration algorithm in which at each step the players adjust their strategies based on the opponent's strategies. This method can therefore be considered as the realization of a discrete system where the strategy selections of the players are the state variables. For symmetric matrix games John von Neumann introduced a continuous systems approach when the players continuously adjust their strategies. This method can be applied to general matrix games, since–as we have seen earlier–any matrix game is equivalent to a symmetric matrix game. The method can also be used to solve linear programming problems as we have seen earlier that any primal-dual pair can be reduced to the solution of a symmetric matrix game.

Let now $\mathbf{P}$ be a skew-symmetric $n \times n$ matrix. The strategy of player $\mathcal{P}_2$, $\mathbf{y}(t)$ is considered as the function of time $t \geq 0$. Before formulating the dynamism of the system, introduce the following notation:

$$\begin{array}{lcccl} u_i: & \mathbb{R}^n & \to & \mathbb{R}, & u_i(\mathbf{y}(t)) = \mathbf{e}_i^T \mathbf{P}\mathbf{y}(t) \quad (i = 1, 2, \ldots, n) , \\ \phi: & \mathbb{R} & \to & \mathbb{R}, & \phi(u_i) = \max\{0, u_i\} , \\ \Phi: & \mathbb{R}^n & \to & \mathbb{R}, & \Phi(\mathbf{y}(t)) = \sum_{i=1}^{n} \phi(u_i(\mathbf{y}(t))) . \end{array} \tag{10.33}$$

For arbitrary probability vector $\mathbf{y}_0$ solve the following nonlinear initial-value problem:

$$y_j'(t) = \phi(u_j(\mathbf{y}(t))) - \Phi(\mathbf{y}(t)) y_j(t), \quad y_j(0) = y_{j0} \quad (1 \leq j \leq n) . \tag{10.34}$$

Since the right-hand side is continuous, there is at least one solution. The right hand side of the equation can be interpreted as follows. Assume that $\phi(u_j(\mathbf{y}(t))) > 0$. If player $\mathcal{P}_2$ selects strategy $\mathbf{y}(t)$, then player $\mathcal{P}_1$ is able to obtain a positive payoff by choosing the pure strategy $\mathbf{e}_j$, which results in a negative payoff for player $\mathcal{P}_2$. However if player $\mathcal{P}_2$ increases $y_j(t)$ to one by choosing the same strategy $\mathbf{e}_j$ its payoff $\mathbf{e}_j^T \mathbf{P}\mathbf{e}_j$ becomes zero, so it increases. Hence it is the interest of player $\mathcal{P}_2$ to increase $y_j(t)$. This is exactly what the first term represents. The second term is needed to ensure that $\mathbf{y}(t)$ remains a probability vector for all $t \geq 0$.

The computation of the right hand side of equations (10.34) for all t requires n^2+n multiplications. The total computation cost depends on the length of solution interval, on the selected step size, and on the choice of the differential equation solver.

Theorem 10.14 *Assume that $t_1, t_2, \ldots$ is a positive strictly increasing sequence converging to ∞, then any cluster point of the sequence $\{\mathbf{y}(t_k)\}$ is equilibrium strategy, furthermore there is a constant c such that*

$$\mathbf{e}_i^T \mathbf{P}\mathbf{y}(t_k) \le \frac{\sqrt{n}}{c+t_k} \quad (i=1,2,\ldots,n)\ . \tag{10.35}$$

Proof First we have to show that $\mathbf{y}(t)$ is a probability vector for all $t \ge 0$. Assume that with some j and $t_1 > 0$, $y_j(t_1) < 0$. Define

$$t_0 = \sup\{t | 0 < t < t_1, y_j(t) \ge 0\}\ .$$

Since $y_j(t)$ is continuous and $y_j(0) \ge 0$, clearly $y_j(t_0) = 0$, and for all $\tau \in (t_0, t_1)$, $y_j(\tau) < 0$. Then for all $\tau \in (t_0, t_1]$,

$$y_j'(\tau) = \phi(u_j(\mathbf{y}(\tau))) - \Phi(\mathbf{y}(\tau)) y_j(\tau) \ge 0,$$

and the Lagrange mean-value theorem implies that with some $\tau \in (t_0, t_1)$,

$$y_j(t_1) = y_j(t_0) + y_j'(\tau)(t_1 - t_0) \ge 0\ ,$$

which is a contradiction. Hence $y_j(t)$ is nonnegative. Next we show that $\sum_{j=1}^n y_j(t) = 1$ for all t. Let $f(t) = 1 - \sum_{j=1}^n y_j(t)$, then

$$f'(t) = -\sum_{j=1}^n y_j'(t) = -\sum_{j=1}^n \phi(u_j(\mathbf{y}(t))) + \Phi(\mathbf{y}(t))(\sum_{j=1}^n y_j(t)) = -\Phi(\mathbf{y}(t))(1 - \sum_{j=1}^n y_j(t))\ ,$$

so $f(t)$ satisfies the homogeneous equation

$$f'(t) = -\Phi(\mathbf{y}(t)) f(t)$$

with the initial condition $f(0) = 1 - \sum_{j=1}^n y_{j0} = 0$. Hence for all $t \ge 0$, $f(t) = 0$, showing that $\mathbf{y}(t)$ is a probability vector.

Assume that for some t, $\phi(u_i(\mathbf{y}(t))) > 0$. Then

$$\begin{aligned} \frac{d}{dt}\phi(u_i(\mathbf{y}(t))) &= \sum_{j=1}^n p_{ij} y_j'(t) = \sum_{j=1}^n p_{ij}[\phi(u_j(\mathbf{y}(t))) - \Phi(\mathbf{y}(t)) y_j(t)] \\ &= \sum_{j=1}^n p_{ij}\phi(u_j(\mathbf{y}(t))) - \Phi(\mathbf{y}(t))\phi(u_i(\mathbf{y}(t))). \end{aligned} \tag{10.36}$$

By multiplying both sides by $\phi(u_i(\mathbf{y}(t)))$ and adding the resulted equations for

$i = 1, 2, \ldots, n$ we have:

$$\sum_{i=1}^{n} \phi(u_i(\mathbf{y}(t))) \frac{d}{dt} \phi(u_i(\mathbf{y}(t))) = \sum_{i=1}^{n} \sum_{j=1}^{n} p_{ij} \phi(u_i(\mathbf{y}(t))) \phi(u_j(\mathbf{y}(t))) \\ -\Phi(\mathbf{y}(t))(\sum_{i=1}^{n} \phi^2(u_i(\mathbf{y}(t)))). \tag{10.37}$$

The first term is zero, since $\mathbf{P}$ is skew-symmetric. Notice that this equation remains valid even as $\phi(u_i(\mathbf{y}(t))) = 0$ except the break-points (where the derivative of $\phi(u_i(\mathbf{y}(t)))$ does not exist) since (10.36) remains true.

Assume next that with a positive t, $\Phi(\mathbf{y}(t)) = 0$. Then for all i, $\phi(u_i(\mathbf{y}(t))) = 0$. Since equation (10.37) can be rewritten as

$$\frac{1}{2} \frac{d}{dt} \Psi(\mathbf{y}(t)) = -\Phi(\mathbf{y}(t)) \Psi(\mathbf{y}(t)) \tag{10.38}$$

with

$$\Psi : \mathbb{R}^n \to \mathbb{R} \quad \text{and} \quad \Psi(\mathbf{y}(t)) = \sum_{i=1}^{n} \phi^2(u_i(\mathbf{y}(t))) \ ,$$

we see that $\Psi(\mathbf{y}(t))$ satisfies a homogeneous equation with zero initial solution at t, so the solution remains zero for all $\tau \geq t$. Therefore $\phi(u_i(\mathbf{y}(\tau))) = 0$ showing that $\mathbf{P}\mathbf{y}(\tau) \leq \mathbf{0}$, that is, $\mathbf{y}(\tau)$ is equilibrium strategy.

If $\Phi(\mathbf{y}(\mathbf{t})) > 0$ for all $t \geq 0$, then $\Psi(\mathbf{y}(t)) > 0$, and clearly

$$\frac{1}{2} \frac{d}{dt} \Psi(\mathbf{y}(t)) \leq -\sqrt{\Psi(\mathbf{y}(t))} \Psi(\mathbf{y}(t)) \ ,$$

that is

$$\frac{1}{2} \frac{d}{dt} \Psi(\mathbf{y}(t)) (\Psi(\mathbf{y}(t)))^{-\frac{3}{2}} \leq -1 \ .$$

Integrate both sides in interval $[0, t]$ to have

$$-\Psi(\mathbf{y}(t))^{-(1/2)} + c \leq -t \ ,$$

with $c = (\Psi(\mathbf{y}(0)))^{-(1/2)}$, which implies that

$$(\Psi(\mathbf{y}(t)))^{1/2} \leq \frac{1}{c+t} \ . \tag{10.39}$$

By using the Cauchy–Schwartz inequality we get

$$\mathbf{e}_i^T \mathbf{P}\mathbf{y}(t) = u_i(\mathbf{y}(t)) \leq \phi(u_i(\mathbf{y}(t))) \leq \Phi(\mathbf{y}(t)) \leq \sqrt{n \Psi(\mathbf{y}(t))} \leq \frac{\sqrt{n}}{c+t} \ , \tag{10.40}$$

which is valid even at the break points because of the continuity of functions u_i. And finally, take a sequence $\{\mathbf{y}(t_k)\}$ with t_k increasingly converging to ∞. The sequence is bounded (being probability vectors), so there is at least one cluster point $\mathbf{y}^\star$. From (10.40), by letting $t_k \to \infty$ we have that $\mathbf{P}\mathbf{y}^\star \leq \mathbf{0}$ showing that $\mathbf{y}^\star$ is an equilibrium strategy. ■

Example 10.18 Consider the matrix game with matrix

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 0 & 3 \\ -1 & 3 & 3 \end{pmatrix},$$

which was the subject of our earlier Example 10.13 In order to apply the method of von Neumann we have to find first an equivalent symmetric matrix game. The application of the method given in Theorem 10.12. requires that the matrix has to be positive. Without changing the equilibria we can add 2 to all matrix elements to have

$$\mathbf{A}_{new} = \begin{pmatrix} 4 & 3 & 2 \\ 4 & 2 & 5 \\ 1 & 5 & 5 \end{pmatrix},$$

and by using the method we get the skew-symmetric matrix

$$\mathbf{P} = \left(\begin{array}{ccc:ccc:c} 0 & 0 & 0 & 4 & 3 & 2 & -1 \\ 0 & 0 & 0 & 4 & 2 & 5 & -1 \\ 0 & 0 & 0 & 1 & 5 & 5 & -1 \\ \hdashline -4 & -4 & -1 & 0 & 0 & 0 & 1 \\ -3 & -2 & -5 & 0 & 0 & 0 & 1 \\ -2 & -5 & -5 & 0 & 0 & 0 & 1 \\ \hdashline 1 & 1 & 1 & -1 & -1 & -1 & 0 \end{array}\right).$$

The differential equations (10.34) were solved by using the 4th order Runge–Kutta method in the interval $[0, 100]$ with the step size $h = 0.01$ and initial vector $\mathbf{y}(0) = (1, 0, \ldots, 0)^T$. From $\mathbf{y}(100)$ we get the approximations

$$\mathbf{x} \approx (0.563619, 0.232359, 0.241988),$$

$$\mathbf{y} \approx (0.485258, 0.361633, 0.115144)$$

of the equilibrium strategies of the original game. Comparing these values to the exact values:

$$\mathbf{x} = \frac{4}{7}, \frac{4}{21}, \frac{5}{21} \quad \text{and} \quad \mathbf{y} = \frac{3}{7}, \frac{3}{7}, \frac{1}{7}$$

we see that the maximum error is about 0.067.

10.2.9. Diagonally strictly concave games

Consider an N-person continuous game and assume that all conditions presented at the beginning of Subsection 10.2.3 are satisfied. In addition, assume that for all k, S_k is bounded, all components of g_k are concave and f_k is concave in s_k with any fixed $s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_N$. Under these conditions there is at least one equilibrium (Theorem 10.3). The uniqueness of the equilibrium is not true in general, even if all

f_k are strictly concave in s_k. Such an example is shown next.

Example 10.19 Consider a two-person game with $S_1 = S_2 = [0,1]$ and $f_1(s_1,s_2) = f_2(s_1,s_2) = 1-(s_1-s_2)^2$. Clearly both payoff functions are strictly concave and there are infinitely many equilibria: $s_1^\star = s_2^\star \in [0,1]$.

Select an arbitrary nonnegative vector $\mathbf{r} \in \mathbb{R}^N$ and define function

$$\mathbf{h} : \mathbb{R}^M \to \mathbb{R}^M, \quad \mathbf{h}(\mathbf{s},\mathbf{r}) = \begin{pmatrix} r_1 \nabla_1 f_1(\mathbf{s})^T \\ r_2 \nabla_2 f_2(\mathbf{s})^T \\ \vdots \\ r_N \nabla_N f_N(\mathbf{s})^T \end{pmatrix}, \tag{10.41}$$

where $M = \sum_{k=1}^N n_k$, and $\nabla_k f_k$ is the gradient (as a row vector) of f_k with respect to s_k. The game is said to be ***diagonally strictly concave*** if for all $\mathbf{s}^{(1)} \neq \mathbf{s}^{(2)}$, $\mathbf{s}^{(1)}, \mathbf{s}^{(2)} \in S$ and for some $\mathbf{r} \geq \mathbf{0}$,

$$(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T (\mathbf{h}(\mathbf{s}^{(1)},\mathbf{r}) - \mathbf{h}(\mathbf{s}^{(2)},\mathbf{r})) < 0. \tag{10.42}$$

Theorem 10.15 *Under the above conditions the game has exactly one equilibrium.*

Proof The existence of the equilibrium follows from Theorem 10.3. In proving uniqueness assume that $\mathbf{s}^{(1)}$ and $\mathbf{s}^{(2)}$ are both equilibria, and both satisfy relations (10.9). Therefore for $l = 1, 2$,

$$\begin{aligned} {\mathbf{u}_k^{(l)}}^T \mathbf{g}_k(s_k^{(l)}) &= 0 \\ \nabla_k f_k(\mathbf{s}^{(l)}) + {\mathbf{u}_k^{(l)}}^T \nabla_k \mathbf{g}_k(s_k^{(l)}) &= \mathbf{0}^T, \end{aligned}$$

and the second equation can be rewritten as

$$\nabla_k f_k(\mathbf{s}^{(l)}) + \sum_{j=1}^{m_k} u_{kj}^{(l)} \nabla_k g_{kj}(s_k^{(l)}) = 0\ , \tag{10.43}$$

where $u_{kj}^{(l)}$ and g_{kj} are the jth components of $\mathbf{u}_k^{(l)}$ and g_k, respectively. Multiplying (10.43) by $(r_k(s_k^{(2)} - s_k^{(1)})^T)$ for $l = 1$ and by $r_k(s_k^{(1)} - s_k^{(2)})^T$ for $l = 2$ and adding the resulted equalities for $k = 1, 2, \ldots, N$ we have

$$\begin{aligned} 0 = \{(\mathbf{s}^{(2)} - \mathbf{s}^{(1)})^T \mathbf{h}(\mathbf{s}^{(1)},\mathbf{r}) + (\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T \mathbf{h}(\mathbf{s}^{(2)},\mathbf{r})\} \\ + \sum_{k=1}^N \sum_{j=1}^{m_k} r_k [u_{kj}^{(1)} (s_k^{(2)} - s_k^{(1)})^T \nabla_k g_{kj}(s_k^{(1)}) + u_{kj}^{(2)} (s_k^{(1)} - s_k^{(2)})^T \nabla_k g_{kj}(s_k^{(2)})]\ . \end{aligned} \tag{10.44}$$

Notice that the sum of the first two terms is positive by the diagonally strict concavity of the game, the concavity of the components of g_k implies that

$$(s_k^{(2)} - s_k^{(1)})^T \nabla_k g_{kj}(s_k^{(1)}) \geq g_{kj}(s_k^{(2)}) - g_{kj}(s_k^{(1)})$$

and

$$(s_k^{(1)} - s_k^{(2)})^T \nabla_k g_{kj}(s_k^{(2)}) \geq g_{kj}(s_k^{(1)}) - g_{kj}(s_k^{(2)})\ .$$

Therefore from (10.44) we have

$$0 > \sum_{k=1}^{N}\sum_{j=1}^{m_k} r_k[u_{kj}^{(1)}(g_{kj}(s_k^{(2)}) - g_{kj}(s_k^{(1)})) + u_{kj}^{(2)}(g_{kj}(s_k^{(1)}) - g_{kj}(s_k^{(2)}))]$$
$$= \sum_{k=1}^{N}\sum_{j=1}^{m_k} r_k[u_{kj}^{(1)} g_{kj}(s_k^{(2)}) + u_{kj}^{(2)} g_{kj}(s_k^{(1)})] \geq 0 \ ,$$

where we used the fact that for all k and l,

$$0 = \mathbf{u}_k^{(l)T}\mathbf{g}_k(s_k^{(l)}) = \sum_{j=1}^{m_k} u_{kj}^{(l)} g_{kj}(s_k^{(l)}).$$

This is an obvious contradiction, which completes the proof. ■

Checking for uniqueness of equilibrium In practical cases the following result is very useful in checking diagonally strict concavity of N-person games.

Theorem 10.16 *Assume S is convex, f_k is twice continuously differentiable for all k, and $\mathbf{J}(\mathbf{s},\mathbf{r}) + \mathbf{J}(\mathbf{s},\mathbf{r})^T$ is negative definite with some $\mathbf{r} \geq \mathbf{0}$, where $\mathbf{J}(\mathbf{s},\mathbf{r})$ is the Jacobian of $\mathbf{h}(\mathbf{s},\mathbf{r})$. Then the game is diagonally strictly concave.*

Proof Let $\mathbf{s}^{(1)} \neq \mathbf{s}^{(2)}$, $\mathbf{s}^{(1)}, \mathbf{s}^{(2)} \in S$. Then for all $\alpha \in [0,1]$, $\mathbf{s}(\alpha) = \alpha\mathbf{s}^{(1)} + (1-\alpha)\mathbf{s}^{(2)} \in S$ and

$$\frac{d}{d\alpha}\mathbf{h}(\mathbf{s}(\alpha),\mathbf{r}) = \mathbf{J}(\mathbf{s}(\alpha),\mathbf{r})(\mathbf{s}^{(1)} - \mathbf{s}^{(2)}) \ .$$

Integrate both side in $[0,1]$ to have

$$\mathbf{h}(\mathbf{s}^{(1)},\mathbf{r}) - \mathbf{h}(\mathbf{s}^{(2)},\mathbf{r}) = \int_0^1 \mathbf{J}(\mathbf{s}(\alpha),\mathbf{r})(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})d\alpha \ ,$$

and by premultiplying both sides by $(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T$ we see that

$$(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T(\mathbf{h}(\mathbf{s}^{(1)},\mathbf{r}) - \mathbf{h}(\mathbf{s}^{(2)},\mathbf{r})) = \int_0^1 (\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T\mathbf{J}(\mathbf{s}(\alpha),\mathbf{r})(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})d\alpha$$
$$= \frac{1}{2}\int_0^1 (\mathbf{s}^{(1)} - \mathbf{s}^{(2)})^T(\mathbf{J}(\mathbf{s}(\alpha),\mathbf{r}) + \mathbf{J}(\mathbf{s}(\alpha),\mathbf{r})^T)(\mathbf{s}^{(1)} - \mathbf{s}^{(2)})d\alpha < 0 \ ,$$

completing the proof. ■

Example 10.20 Consider a simple two-person game with strategy sets $S_1 = S_2 = [0,1]$, and payoff functions

$$f_1(s_1,s_2) = -s_1^2 + s_1 - s_1 s_2$$

and

$$f_2(s_1,s_2) = -s_2^2 + s_2 - s_1 s_2 \ .$$

Clearly all conditions, except diagonally strict concavity, are satisfied. We will use Theorem 10.16 to show this additional property. In this case

$$\nabla_1 f_1(s_1,s_2) = -2s_1 + 1 - s_2, \quad \nabla_2 f_2(s_1,s_2) = -2s_2 + 1 - s_1 \ ,$$

so

$$\mathbf{h}(\mathbf{s},\mathbf{r}) = \begin{pmatrix} r_1(-2s_1+1-s_2) \\ r_2(-2s_2+1-s_1 \end{pmatrix}$$

with Jacobian

$$\mathbf{J}(\mathbf{s},\mathbf{r}) = \begin{matrix} -2r_1 & -r_1 \\ -r_2 & -2r_2 \end{matrix} .$$

We will show that

$$\mathbf{J}(\mathbf{s},\mathbf{r}) + \mathbf{J}(\mathbf{s},\mathbf{r})^T = \begin{matrix} -4r_1 & -r_1 - r_2 \\ -r_1 - r_2 & -4r_2 \end{matrix}$$

is negative definite with some $\mathbf{r} \geq \mathbf{0}$. For example, select $r_1 = r_2 = 1$, then this matrix becomes

$$\begin{matrix} -4 & -2 \\ -2 & -4 \end{matrix}$$

with characteristic polynomial

$$\phi(\lambda) = \det \begin{matrix} -4-\lambda & -2 \\ -2 & -4-\lambda \end{matrix} = \lambda^2 + 8\lambda + 12 ,$$

having negative eigenvalues $\lambda_1 = -2$, $\lambda_2 = -6$.

Iterative computation of equilibrium We have see earlier in Theorem 10.4 that $\mathbf{s}^\star \in S$ is an equilibrium if and only if

$$\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}^\star) \geq \mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}) \tag{10.45}$$

for all $\mathbf{s} \in S$, where $\mathbf{H}_\mathbf{r}$ is the aggregation function (10.4). In the following analysis we assume that the N-person game satisfies all conditions presented at the beginning of Subsection 10.2.9 and (10.42) holds with some positive $\mathbf{r}$.

We first show the equivalence of (10.45) and a variational inequality.

Theorem 10.17 *A vector $\mathbf{s}^\star \in S$ satisfies (10.45) if and only if*

$$\mathbf{h}(\mathbf{s}^\star, \mathbf{r})^T(\mathbf{s} - \mathbf{s}^\star) \leq 0 \tag{10.46}$$

for all $\mathbf{s} \in S$, where $\mathbf{h}(\mathbf{s},\mathbf{r})$ is defined in (10.41).

Proof Assume $\mathbf{s}^\star$ satisfies (10.45). Then $\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s})$ as function of $\mathbf{s}$ obtains maximum at $\mathbf{s} = \mathbf{s}^\star$, therefore

$$\nabla_\mathbf{s}\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}^\star)(\mathbf{s} - \mathbf{s}^\star) \leq 0$$

for all $\mathbf{s} \in S$, and since $\nabla_\mathbf{s}\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}^\star)$ is $\mathbf{h}(\mathbf{s}^\star, \mathbf{r})$, we proved that $\mathbf{s}^\star$ satisfies (10.46).

Assume next that $\mathbf{s}^\star$ satisfies (10.46). By the concavity of $\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s})$ in $\mathbf{s}$ and the diagonally strict concavity of the game we have

$$\mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}^\star) - \mathbf{H}_\mathbf{r}(\mathbf{s}^\star, \mathbf{s}) \geq \mathbf{h}(\mathbf{s},\mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) \geq \mathbf{h}(\mathbf{s},\mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) + \mathbf{h}(\mathbf{s}^\star,\mathbf{r})^T(\mathbf{s} - \mathbf{s}^\star) > 0 ,$$

so $\mathbf{s}^\star$ satisfies (10.45). ∎

Hence any method available for solving variational inequalities can be used to find equilibria.

Next we construct a special two-person, game the equilibrium problem of which is equivalent to the equilibrium problem of the original N-person game.

Theorem 10.18 *Vector $\mathbf{s}^\star \in S$ satisfies (10.45) if and only if $(\mathbf{s}^\star, \mathbf{s}^\star)$ is an equilibrium of the two-person game $D = \{2; S, S; f, -f\}$ where $f(\mathbf{s}, \mathbf{z}) = \mathbf{h}(\mathbf{z}, \mathbf{r})^T(\mathbf{s} - \mathbf{z})$.*

Proof

- Assume first that $\mathbf{s}^\star \in S$ satisfies (10.45). Then it satisfies (10.46) as well, so

$$f(\mathbf{s}, \mathbf{s}^\star) \leq 0 = f(\mathbf{s}^\star, \mathbf{s}^\star).$$

  We need in addition to show that

$$-f(\mathbf{s}^\star, \mathbf{s}) \leq 0 = -f(\mathbf{s}^\star, \mathbf{s}^\star).$$

  In contrary assume that with some $\mathbf{s}$, $f(\mathbf{s}^\star, \mathbf{s}) < 0$. Then

$$\begin{aligned} 0 > f(\mathbf{s}^\star, \mathbf{s}) &= \mathbf{h}(\mathbf{s}, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) > \mathbf{h}(\mathbf{s}, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) + (\mathbf{s} - \mathbf{s}^\star)^T(\mathbf{h}(\mathbf{s}, \mathbf{r}) - \mathbf{h}(\mathbf{s}^\star, \mathbf{r})) \\ &= \mathbf{h}(\mathbf{s}^\star, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}) \geq 0, \end{aligned}$$

  where we used (10.42) and (10.46). This is a clear contradiction.
- Assume next that $(\mathbf{s}^\star, \mathbf{s}^\star)$ is an equilibrium of game D. Then for any $\mathbf{s}, \mathbf{z} \in S$,

$$f(\mathbf{s}, \mathbf{s}^\star) \leq f(\mathbf{s}^\star, \mathbf{s}^\star) = 0 \leq f(\mathbf{s}^\star, \mathbf{z}).$$

  The first part can be rewritten as

$$\mathbf{h}(\mathbf{s}^\star, \mathbf{r})^T(\mathbf{s} - \mathbf{s}^\star) \leq 0,$$

  showing that (10.46) is satisfied, so is (10.45).

■

Consider the following iteration procedure.
Let $\mathbf{s}^{(1)} \in S$ be arbitrary, and solve problem

$$\begin{array}{ll} \text{maximize} & f(\mathbf{s}, \mathbf{s}^{(1)}) \\ \text{subject to} & \mathbf{s} \in S. \end{array} \tag{10.47}$$

Let $\mathbf{s}^{(2)}$ denote an optimal solution and define $\mu_1 = f(\mathbf{s}^{(2)}, \mathbf{s}^{(1)})$. If $\mu_1 = 0$, then for all $\mathbf{s} \in S$,

$$f(\mathbf{s}, \mathbf{s}^{(1)}) = \mathbf{h}(\mathbf{s}^{(1)}, \mathbf{r})^T(\mathbf{s} - \mathbf{s}^{(1)}) \leq 0,$$

so by Theorem 10.17, $\mathbf{s}^{(1)}$ is an equilibrium. Since $f(\mathbf{s}^{(1)}, \mathbf{s}^{(1)}) = 0$, we assume that $\mu_1 > 0$. In the general step $k \geq 2$ we have already k vectors $\mathbf{s}^{(1)}, \mathbf{s}^{(2)}, \ldots, \mathbf{s}^{(k)}$, and $k - 1$ scalers $\mu_1, \mu_2, \ldots, \mu_{k-1} > 0$. Then the next vector $\mathbf{s}^{(k+1)}$ and next scaler μ_k are the solutions of the following problem:

$$\begin{array}{lll} \text{maximize} & \mu & \\ \text{subject to} & f(\mathbf{s}, \mathbf{s}^{(i)}) \geq \mu & (i = 1, 2, \ldots, k) \\ & \mathbf{s} \in S. & \end{array} \tag{10.48}$$

Notice that

$$f(\mathbf{s}^{(k)}, \mathbf{s}^{(i)}) \geq \mu_{k-1} \geq 0 \qquad (i = 1, 2, \ldots, k-1)$$

and

$$f(\mathbf{s}^{(k)}, \mathbf{s}^{(k)}) = 0,$$

so we know that $\mu_k \geq 0$.

The formal algorithm is as follows:

CONTINUOUS-EQUILIBRIUM

```
1  k ← 1
2  solve problem (10.47), let s^(2) be optimal solution
3  if f(s^(2), s^(1)) = 0
4      then s^(1) is equilibrium
5           return s^(1)
6  k ← k + 1
7  solve problem (10.48), let s^(k+1) be optimal solution
8  if ||s^(k+1) − s^(k)|| < ε
9      then s^(k+1) is equilibrium
10          return s^(k+1)
11     else go to 5
```

Before stating the convergence theorem of the algorithm we notice that in the special case when the strategy sets are defined by linear inequalities (that is, all functions g_k are linear) then all constraints of problem (10.48) are linear, so at each iteration step we have to solve a linear programming problem.

In this linear case the simplex method has to be used in each iteration step with exponential computational cost, so the overall cost is also exponential (with prefixed number of steps).

Theorem 10.19 *There is a subsequence* $\{\mathbf{s}^{(k_i)}\}$ *of* $\{\mathbf{s}^{(k)}\}$ *generated by the method that converges to the unique equilibrium of the* N*-person game.*

Proof The proof consists of several steps.

First we show that $\mu_k \to 0$ as $k \to \infty$. Since at each new iteration an additional constraint is added to (10.48), sequence $\{\mu_k\}$ is nonincreasing. Since it is also nonnegative, it must be convergent. Sequence $\{\mathbf{s}^{(k)}\}$ is bounded, since it is from the bounded set S, so it has a convergent subsequence $\{\mathbf{s}^{(k_i)}\}$. Notice that from (10.48) we have

$$0 \leq \mu_{k_i - 1} = \min_{1 \leq k \leq k_i - 1} \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T (\mathbf{s}^{(k_i)} - \mathbf{s}^{(k)}) \leq \mathbf{h}(\mathbf{s}^{(k_{i-1})}, \mathbf{r})^T (\mathbf{s}^{(k_i)} - \mathbf{s}^{(k_{i-1})}) \ ,$$

where the right hand side tends to zero. Thus $\mu_{k_i - 1} \to 0$ and since the entire sequence $\{\mu_k\}$ is monotonic, the entire sequence converges to zero.

Let next $\mathbf{s}^\star$ be an equilibrium of the N-person game, and define

$$\delta(t) = \min\{(\mathbf{h}(\mathbf{s}, \mathbf{r}) - \mathbf{h}(\mathbf{z}, \mathbf{r}))^T (\mathbf{z} - \mathbf{s}) | \|\mathbf{s} - \mathbf{z}\| \geq t, \ \mathbf{z}, \mathbf{s} \in S\}. \tag{10.49}$$

By (10.42), $\delta(t) > 0$ for all $t > 0$. Define the indices k_i so that

$$\delta(\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|) = \min_{1 \leq k \leq i} \delta(\|\mathbf{s}^{(k)} - \mathbf{s}^\star\|) \quad (i = 1, 2, \ldots),$$

then for all $k = 1, 2, \ldots, i$,

$$\begin{aligned} \delta(\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|) &\leq (\mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r}) - \mathbf{h}(\mathbf{s}^\star, \mathbf{r}))^T(\mathbf{s}^\star - \mathbf{s}^{(k)}) \\ &= \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}^{(k)}) - \mathbf{h}(\mathbf{s}^\star, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}^{(k)}) \\ &\leq \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}^{(k)}), \end{aligned}$$

which implies that

$$\begin{aligned} \delta(\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|) &\leq \min_{1\leq k\leq i} \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T(\mathbf{s}^\star - \mathbf{s}^{(k)}) \\ &\leq \max_{\mathbf{s}\in S} \min_{1\leq k\leq i} \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T(\mathbf{s} - \mathbf{s}^{(k)}) \\ &= \min_{1\leq k\leq i} \mathbf{h}(\mathbf{s}^{(k)}, \mathbf{r})^T(\mathbf{s}^{(i+1)} - \mathbf{s}^{(k)}) \\ &= \mu_i \end{aligned}$$

where we used again problem (10.48). From this relation we conclude that $\delta(\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|) \to 0$ as $i \to \infty$. And finally, notice that function $\delta(t)$ satisfies the following properties:

1. $\delta(t)$ is continuous in t;
2. $\delta(t) > 0$ if $t > 0$ (as it was shown just below relation (10.49));
3. if for a convergent sequence $\{t^{(k)}\}$, $\delta(t^{(k)}) \to 0$, then necessarily $t^{(k)} \to 0$.

By applying property 3. with sequence $\{\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\|\}$ it is clear that $\|\mathbf{s}^{(k_i)} - \mathbf{s}^\star\| \to 0$ so $\mathbf{s}^{(k_i)} \to \mathbf{s}^\star$. Thus the proof is complete. ■

Exercises

10.2-1 Consider a 2-person game with strategy sets $S_1 = S_2 = [0, 1]$, and payoff functions $f_1(x_1, x_2) = x_1^2 + x_1x_2 + 2$ and $f_2(x_1, x_2) = x_1 + x_2$. Show the existence of a unique equilibrium point by computing it. Show that Theorem 10.3. cannot be applied to prove existence.

10.2-2 Consider the "price war" game in which two firms are price setting. Assume that p_1 and p_2 are the strategies of the players, $p_1, p_2 \in [0, p_{max}]$ and the payoff functions are:

$$f_1(p_1, p_2) = \begin{cases} p_1, & \text{if } p_1 \leq p_2, \\ p_1 - c, & \text{if } p_1 > p_2, \end{cases}$$

$$f_2(p_1, p_2) = \begin{cases} p_2, & \text{if } p_2 \leq p_1, \\ p_2 - c, & \text{if } p_2 > p_1, \end{cases}$$

by assuming that $c < p_{max}$. Is there an equilibrium? How many equilibria were found?

10.2-3 A portion of the sea is modeled by the unit square in which a submarine is hiding. The strategy of the submarine is the hiding place $\mathbf{x} \in [0, 1] \times [0, 1]$. An airplane drops a bomb in a location $\mathbf{y} = [0, 1] \times [0, 1]$,j which is its strategy. The payoff of the airplane is the damage $\alpha e^{-\beta\|\mathbf{x}-\mathbf{y}\|}$ occurred by the bomb, and the payoff of the submarine is its negative. Does this 2-person game have an equilibrium?

10.2-4 In the second-price auction they sell one unit of an item to N bidders. They value the item as $v_1 < v_2 < \cdots < v_N$. Each of them offers a price for the item simultaneously without knowing the offers of the others. The bidder with the highest offer will get the item, but he has to pay only the second highest price. So the strategy of bidder k is $[0, \infty]$, so $x_k \in [0, \infty]$, and the payoff function for this bidder is:

$$f_k(x_1, x_2, \ldots, x_N) = \begin{cases} v_k - \max_{j \neq k} x_j, & \text{if } x_k = \max_j x_j, \\ 0 & \text{otherwise.} \end{cases}$$

What is the best response function of bidder k? Does this game have equilibrium?

10.2-5 Formulate Fan's inequality for Exercise 10.2-1

10.2-6 Formulate and solve Fan's inequality for Exercise 10.2-2

10.2-7 Formulate and solve Fan's inequality for Exercise 10.2-4

10.2-8 Consider a 2-person game with strategy sets $S_1 = S_2 = [0, 1]$, and payoff functions

$$f_1(x_1, x_2) = -(x_1 - x_2)^2 + 2x_1 - x_2 + 1$$
$$f_2(x_1, x_2) = -(x_1 - 2x_2)^2 - 2x_1 + x_2 - 1.$$

Formulate Fan's inequality.

10.2-9 Let $n = 2$, $S_1 = S_2 = [0, 10]$, $f_1(x_1, x_2) = f_2(x_1, x_2) = 2x_1 + 2x_2 - (x_1 + x_2)^2$. Formulate the Kuhn-Tucker conditions to find the equilibrium. Solve the resulted system of inequalities and equations.

10.2-10 Consider a 3-person game with $S_1 = S_2 = S_3 = [0, 1]$, $f_1(x_1, x_2, x_3) = (x_1 - x_2)^2 + x_3$, $f_2(x_1, x_2, x_3) = (x_2 - x_3)^2 + x_1$ and $f_3(x_1, x_2, x_3) = (x_3 - x_1)^2 + x_2$. Formulate the Kuhn-Tucker condition.

10.2-11 Formulate and solve system (10.9) for exercise 10.2-8

10.2-12 Repeat the previous problem for the game given in exercise 10.2-1

10.2-13 Rewrite the Kuhn-Tucker conditions for exercise 10.2-8 into the optimization problem (10.10) and solve it.

10.2-14 Formulate the mixed extension of the finite game given in exercise 10.1-1

10.2-15 Formulate and solve optimization problem (10.10) for the game obtained in the previous problem.

10.2-16 Formulate the mixed extension of the game introduced in Exercise 10.2-3 Formulate and solve the corresponding linear optimization problems (10.22) with $\alpha = 5$, $\beta = 3$, $\gamma = 1$.

10.2-17 Use fictitious play method for solving the matrix game of exercise 10.2-16

10.2-18 Generalize the fictitious play method for bimatrix games.

10.2-19 Generalize the fictitious play method for the mixed extensions of finite n-person games.

10.2-20 Solve the bimatrix game with matrics $\mathbf{A} = \begin{pmatrix} 2 & -1 \\ -1 & 1 \end{pmatrix}$ and $\mathbf{B} = \begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix}$ with the method you have developed in Exercise 10.2-18

10.2-21 Solve the symmetric matrix game $\mathbf{A} = \begin{pmatrix} 0 & 1 & 5 \\ -1 & 0 & -3 \\ -5 & 3 & 0 \end{pmatrix}$ by linear pro-

gramming.

10.2-22 Repeat exercise 10.2-21 with the method of fictitious play.

10.2-23 Develop the Kuhn-Tucker conditions (10.9) for the game given in Exercise 10.2-21 above.

10.2-24⋆ Repeat Exercises 10.2-21, 10.2-22 and 10.2-23 for the matrix game $\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ -1 & 0 & 1 \end{pmatrix}$. (First find the equivalent symmetric matrix game!).

10.2-25 Formulate the linear programming problem to solve the matrix game with matrix $\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 1 \end{pmatrix}$.

10.2-26 Formulate a linear programming solver based on the method of fictitious play and solve the LP problem:

$$\begin{aligned} \text{maximize} \quad & x_1 + x_2 \\ \text{subject to} \quad & x_1, x_2 \geq 0 \\ & 3x_1 + x_2 \leq 4 \\ & x_1 + 3x_2 \leq 4. \end{aligned}$$

10.2-27 Solve the LP problem given in Example 8.17 by the method of fictitious play.

10.2-28 Solve Exercise 10.2-21 by the method of von Neumann.

10.2-29 Solve Exercise 10.2-24. by the method of von Neumann.

10.2-30 Solve Exercise 10.2-17 by the method of von Neumann.

10.2-31⋆ Check the solution obtained in the previous exercises by verifying that all constraints of (10.21) are satisfied with zero objective function. *Hint.* What α and β should be selected?

10.2-32 Solve exercise 10.2-26 by the method of von Neumann.

10.2-33 Let $N = 2$, $S_1 = S_2 = [0, 10]$, $f_1(x_1, x_2) = f_2(x_1, x_2) = 2x_1 + 2x_2 - (x_1 + x_2)^2$. Show that both payoff functions are strictly concave in x_1 and x_2 respectively. Prove that there are infinitely many equilibria, that is , the strict concavity of the payoff functions does not imply the uniqueness of the equilibrium.

10.2-34 Can matrix games be strictly diagonally concave?

10.2-35 Consider a two-person game with strategy sets $S_1 = S_2 = [0, 1]$, and payoff functions $f_1(x_1, x_2) = -2x_1^2 + x_1(1 - x_2)$, $f_2(x_1, x_2) = -3x_2^2 + x_2(1 - x_1)$. Show that this game satisfies all conditions of Theorem 10.16.

10.2-36 Solve the problem of the previous exercise by algorithm (10.47)–(10.48).

10.3. The oligopoly problem

The previous sections presented general methodology, however special methods are available for almost all special classes of games. In the following parts of this chapter a special game, the ***oligopoly game*** will be examined. It describes a real-life economic situation when N-firms produce a homogeneous good to a market, or offers the same service. This model is known as the ***classical Cournot model***. The firms are the players. The strategy of each player is its production level x_k with strategy set

$S_k = [0, L_k]$, where L_k is its capacity limit. It is assumed that the market price depends on the total production level $s = x_1 + x_2 + \cdots + x_N$ offered to the market: $p(s)$, and the cost of each player depends on its own production level: $c_k(x_k)$. The profit of each firm is given as

$$f_k(x_1, \ldots, x_N) = x_k p\left(\sum_{l=1}^{N} x_l\right) - c_k(x_k). \tag{10.50}$$

In this way an N-person game $G = \{N; S_1, \ldots, S_N; f_1, \ldots, f_N\}$ is defined.

It is usually assumed that functions p and c_k $(k = 1, 2, \ldots, N)$ are twice continuously differentiable, furthermore

1. $p'(s) < 0$;
2. $p'(s) + x_k p''(s) \leq 0$;
3. $p'(s) - c_k''(x_k) < 0$

for all k, $x_k \in [0, L_k]$ and $s \in [0, \sum_{l=1}^{N} L_l]$. Under assumptions 1.–3. the game satisfies all conditions of Theorem 10.3, so there is at least one equilibrium.

Best reply mappings Notice that with the notation $s_k = \sum_{l \neq k} x_l$, the payoff function of player $\mathcal{P}_k$ can be rewritten as

$$x_k p(x_k + s_k) - c_k(x_k)\ . \tag{10.51}$$

Since S_k is a compact set and this function is strictly concave in x_k, with fixed s_k there is a unique profit maximizing production level of player $\mathcal{P}_k$, which is its best reply and is denoted by $B_k(s_k)$.

It is easy to see that there are three cases: $B_k(s_k) = 0$ if $p(s_k) - c_k'(0) \leq 0$, $B_k(s_k) = L_k$ if $p(s_k + L_k) + L_k p'(s_k + L_k) - c_k'(L_k) \geq 0$, and otherwise $B_k(s_k)$ is the unique solution of the monotonic equation

$$p(s_k + x_k) + x_k p'(s_k + x_k) - c_k'(x_k) = 0\ .$$

Assume that $x_k \in (0, L_k)$. Then implicit differentiation with respect to s_k shows that

$$p'(1 + B_k') + B_k' p' + x_k p''(1 + B_k') - c_k'' B_k' = 0$$

showing that

$$B_k'(s_k) = -\frac{p' + x_k p''}{2p' + x_k p'' - c_k''}\ .$$

Notice that from assumptions 2. and 3.,

$$-1 < B_k'(s_k) \leq 0\ , \tag{10.52}$$

which is also true for the other two cases except for the break points.

As in Subsection 10.2.1. we can introduce the best reply mapping:

$$\mathbf{B}(x_1,\ldots,x_N)=\left(B_1\left(\sum_{l\neq 1}x_l\right),\ldots,B_N\left(\sum_{l\neq N}x_l\right)\right) \tag{10.53}$$

and look for its fixed points. Another alternative is to introduce dynamic process which converges to the equilibrium.

Similarly to the method of fictitious play a discrete system can be developed in which each firm selects its best reply against the actions of the competitors chosen at the previous time period:

$$x_k(t+1)=B_k(\sum_{l\neq k}x_l(t)) \qquad (k=1,2,\ldots,N)\ . \tag{10.54}$$

Based on relation (10.52) we see that for $N=2$ the right hand side mapping $\mathbb{R}^2\to\mathbb{R}^2$ is a contraction, so it converges, however if $N>2$, then no convergence can be established. Consider next a slight modification of this system: with some $K_k>0$:

$$x_k(t+1)=x_k(t)+K_k(B_k(\sum_{l\neq k}x_l(t))-x_k(t)) \tag{10.55}$$

for $k=1,2,\ldots,N$. Clearly the steady-states of this system are the equilibria, and it can be proved that if K_k is sufficiently small, then sequences $x_k(0),x_k(1),x_k(2),\ldots$ are all convergent to the equilibrium strategies.

Consider next the continuous counterpart of model (10.55), when (similarly to the method of von Neumann) continuous time scales are assumed:

$$\dot{x}_k(t)=K_k(B_k(\sum_{l\neq k}x_l(t))-x_k(t)) \qquad (k=1,2,\ldots,N)\ . \tag{10.56}$$

The following result shows the convergence of this process.

Theorem 10.20 *Under assumptions 1–3, system* (10.56) *is asymptotically stable, that is, if the initial* $x_k(0)$ *values are selected close enough to the equilibrium, then as* $t\to\infty$, $x_k(t)$ *converges to the equilibrium strategy for all* k.

Proof It is sufficient to show that the eigenvalues of the Jacobian of the system have negative real parts. Clearly the Jacobian is as follows:

$$\mathbf{J}=\begin{pmatrix} -K_1 & K_1b_1 & \cdots & K_1b_1 \\ K_2b_2 & -K_2 & \cdots & K_2b_2 \\ \vdots & \vdots & & \vdots \\ K_Nb_N & K_Nb_N & \cdots & -K_N \end{pmatrix}\ , \tag{10.57}$$

where $b_k=B_k'(\sum_{l\neq k}x_l)$ at the equilibrium. From (10.52) we know that $-1<b_k\leq 0$ for all k. In order to compute the eigenvalues of $\mathbf{J}$ we will need a simple but very useful fact. Assume that $\mathbf{a}$ and $\mathbf{b}$ are N-element real vectors. Then

$$\det(\mathbf{I}+\mathbf{a}\mathbf{b}^T)=1+\mathbf{b}^T\mathbf{a}\ , \tag{10.58}$$

where $\mathbf{I}$ is the $N \times N$ identity matrix. This relation can be easily proved by using finite induction with respect to N. By using (10.58), the characteristic polynomial of $\mathbf{J}$ can be written as

$$\begin{aligned}\phi(\lambda) &= \det(\mathbf{J}-\lambda\mathbf{I}) = \det(\mathbf{D}+\mathbf{a}\mathbf{b}^T-\lambda\mathbf{I}) \\ &= \det(\mathbf{D}-\lambda\mathbf{I})\det(\mathbf{I}+(\mathbf{D}-\lambda\mathbf{I})^{-1}\mathbf{a}\mathbf{b}^T) \\ &= \det(\mathbf{D}-\lambda\mathbf{I})[1+\mathbf{b}^T(\mathbf{D}-\lambda\mathbf{I})^{-1}\mathbf{a}] \\ &= \Pi_{k=1}^N(-K_k(1+b_k)-\lambda)[1+\sum_{k=1}^N \frac{K_k b_k}{-K_k(1+b_k)-\lambda}],\end{aligned}$$

where we used the notation

$$\mathbf{a} = \begin{pmatrix} K_1 b_1 \\ K_2 b_2 \\ \vdots \\ K_N b_N \end{pmatrix}, \ \mathbf{b}^T = (1,1,\ldots,1), \ \mathbf{D} = \begin{pmatrix} -K_1(1+b_1) & & \\ & \ddots & \\ & & -K_N(1+b_N) \end{pmatrix}.$$

The roots of the first factor are all negative: $\lambda = -K_k(1+b_k)$, and the other eigenvalues are the roots of equation

$$1+\sum_{k=1}^N \frac{K_k b_k}{-K_k(1+b_k)-\lambda} = 0.$$

Notice that by adding the terms with identical denominators this equation becomes

$$1+\sum_{l=1}^m \frac{\alpha_k}{\beta_k+\lambda} = 0 \tag{10.59}$$

with $\alpha_k, \beta_k > 0$, and the $\beta_k s$ are different. If $g(\lambda)$ denotes the left hand side then clearly the values $\lambda = -\beta_k$ are the poles,

$$\lim_{\lambda\to\pm\infty} g(\lambda) = 1, \quad \lim_{\lambda\to-\beta_k\pm 0} g(\lambda) = \pm\infty,$$
$$g'(\lambda) = \sum_{l=1}^m \frac{-\alpha_l}{(\beta_l+\lambda)^2} < 0 \ ,$$

so $g(\lambda)$ strictly decreases locally. The graph of the function is shown in Figure 10.3. Notice first that (10.59) is equivalent to a polynomial equation of degree m, so there are m real or complex roots. The properties of function $g(\lambda)$ indicate that there is one root below $-\beta_1$, and one root between each $-\beta_k$ and $-\beta_{k+1}$ $(k = 1, 2, \ldots, m-1)$. Therefore all roots are negative, which completes the proof. ■

The general discrete model (10.55) can be examined in the same way. If $K_k = 1$ for all k, then model (10.55) reduces to the simple dynamic process (10.54).

Example 10.21 Consider now a 3-person oligopoly with price function

$$p(s) = \begin{array}{ll} 2-2s-s^2, & \text{if} \quad 0 \le s \le \sqrt{3}-1 \ , \\ 0 & \text{otherwise} \ , \end{array}$$

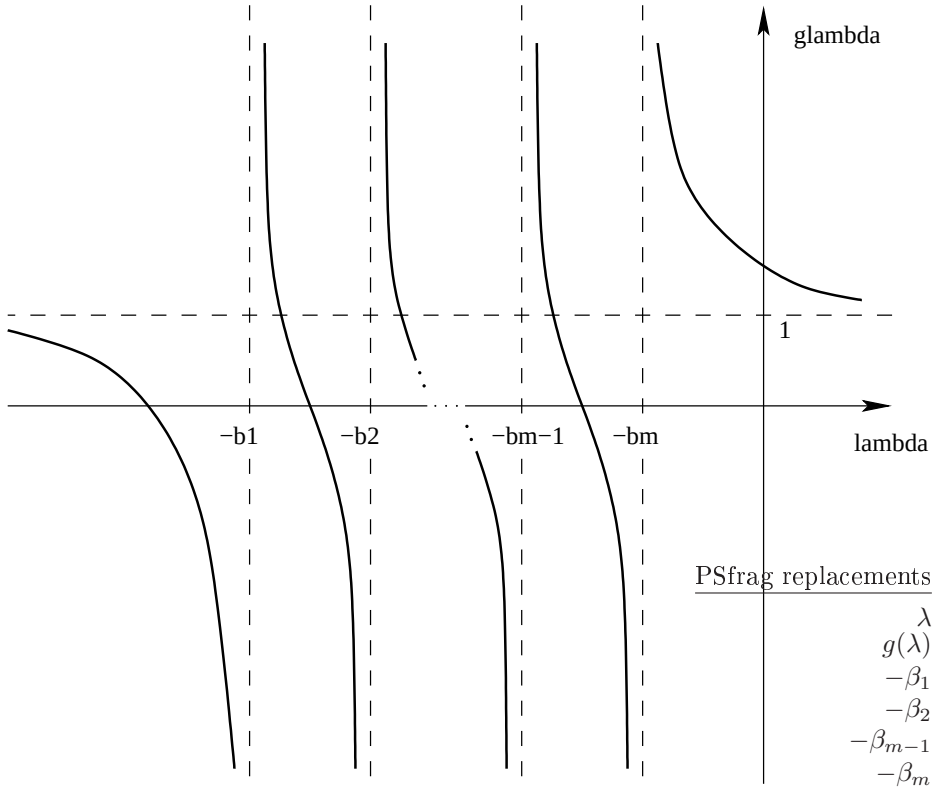


Figure 10.5. The graph of function $g(\lambda)$.

strategy sets $S_1 = S_2 = S_3 = [0, 1]$, and cost functions

$$c_k(x_k) = kx_k^3 + x_k \qquad (k = 1, 2, 3) \ .$$

The profit of firm k is therefore the following:

$$x_k(2 - 2s - s^2) - (kx_k^3 + x_k) = x_k(2 - 2x_k - 2s_k - x_k^2 - 2x_k s_k - s_k^2) - kx_k^3 - x_k \ .$$

The best reply of play k can be obtained as follows. Following the method outlined at the beginning of Section 10.3 we have the following three cases. If $1 - 2s_k - s_k^2 \leq 0$, then $x_k = 0$ is the best choice. If $(-6-3k) - 6s_k - s_k^2 \geq 0$, then $x_k = 1$ is the optimal decision. Otherwise x_k is the solution of equation

$$\begin{aligned} &\frac{\partial}{\partial x_k}[x_k(2 - 2x_k - 2s_k - s_k^2 - 2s_k x_k - x_k^2) - kx_k^3 - x_k] \\ &= 2 - 4x_k - 2s_k - s_k^2 - 4s_k x_k - 3x_k^2 - 3kx_k^2 - 1 = 0 \ , \end{aligned}$$

where the only positive solution is

$$x_k = \frac{-(4 + 4s_k) + \sqrt{(4 + 4s_k)^2 - 12(1 + k)(s_k^2 + 2s_k - 1)}}{6(1 + k)} \ .$$

After the best replies are found, we can easily construct any of the methods presented before.

Reduction to single-dimensional fixed point problems Consider an N-firm oligopoly with price function p and cost functions c_k $(k = 1, 2, \ldots, N)$. Introduce the following function

$$\Psi_k(s, x_k, t_k) = t_k p(s - x_k + t_k) - c_k(t_k) \ , \tag{10.60}$$

and define

$$X_k(s) = \{x_k | x_k \in S_k, \quad \Psi_k(s, x_k, x_k) = \max_{t_k \in S_k} \Psi_k(s, x_k, t_k)\} \tag{10.61}$$

for $k = 1, 2, \ldots, N$ and let

$$X(s) = \{u | u = \sum_{k=1}^{N} x_k, \quad x_k \in X_k(s), \quad k = 1, 2, \ldots, N\}. \tag{10.62}$$

Notice that if $s \in [0, \sum_{k=1}^{N} L_k]$, then all elements of $X(s)$ are also in this interval, therefore X is a single-dimensional point-to-set mapping. Clearly $(x_1^\star, \ldots, x_N^\star)$ is an equilibrium of the N-firm oligopoly game if and only if $s^\star = \sum_{k=1}^{N} x_k^\star$ is a fixed point of mapping X and for all k, $x_k^\star \in X_k(s^\star)$. Hence the equilibrium problem has been reduced to find fixed points of only one-dimensional mappings. This is a significant reduction in the difficulty of the problem, since best replies are N-dimensional mappings.

If conditions 1–3 are satisfied, then $X_k(s)$ has exactly one element for all s and k:

$$X(s) = \begin{cases} 0, & \text{if } p(s) - c_k'(0) \le 0 \ , \\ L_k, & \text{if } p(s) + L_k p_k'(s) - c_k'(L_k) \ge 0 \ , \\ z^\star & \text{otherwise,} \end{cases} \tag{10.63}$$

where $z^\star$ is the unique solution of the monotonic equation

$$p(s) + z p'(s) - c_k'(z) = 0 \tag{10.64}$$

in the interval $(0, L_k)$. In the third case, the left hand side is positive at $z = 0$, negative at $z = L_k$, and by conditions 2–3, it is strictly decreasing, so there is a unique solution.

In the entire interval $[0, \sum_{k=1}^{N} L_k]$, $X_k(s)$ is nonincreasing. In the first two cases it is constant and in the third case strictly decreasing. Consider finally the single-dimensional equation

$$\sum_{k=1}^{N} X_k(s) - s = 0 \ . \tag{10.65}$$

At $s = 0$ the left hand side is nonnegative, at $s = \sum_{k=1}^{N} L_k$ it is nonpositive, and is strictly decreasing. Therefore there is a unique solution (that is, fixed point of mapping X), which can be obtained by any method known to solve single-dimensional equations.

Let $[0, S_{max}]$ be the initial interval for the solution of equation (10.65). After K bisection steps the accuracy becomes $S_{max}/2^K$, which will be smaller than an error tolerance $\epsilon > 0$ if $K > \log_2(S_{max}/\epsilon)$.

OLIGOPOLY-EQUILIBRIUM$(p(s), X(s))$

1 solve equation (10.65) for s
2 **for** $k \leftarrow 1$ to n
3 **do** solve equation (10.64), and let $x_k \leftarrow z$
4 $(x_1, \ldots, x_N)$ is equilibrium

Example 10.22 Consider the 3-person oligopoly examined in the previous example. From (10.63) we have

$$X(s) = \begin{cases} 0, & \text{if} \quad 1 - 2s - s^2 \leq 0, \\ 1, & \text{if} \quad -(1+3k) - 4s - s^2 \geq 0, \\ z^\star & \text{otherwise,} \end{cases}$$

where $z^\star$ is the unique solution of equation

$$3kz^2 + z(2s+2) + (-1 + 2s + s^2) = 0.$$

The first case occurs for $s \geq \sqrt{2} - 1$, the second case never occurs, and in the third case there is a unique positive solution:

$$z^\star = \frac{-(2s+2) + \sqrt{(2s+2)^2 - 12k(-1+2s+s^2)}}{6k} \,. \tag{10.66}$$

And finally equation (10.65) has the special form

$$\sum_{k=1}^{3} \frac{-(s+1) + \sqrt{(s+1)^2 - 3k(-1+2s+s^2)}}{3k} - s = 0 \,.$$

A single program based on the bisection method gives the solution $s^\star \approx 0.2982$ and then equation (10.66) gives the equilibrium strategies $x_1^\star \approx 0.1077$, $x_2^\star \approx 0.0986$, $x_3^\star \approx 0.0919$.

Methods based on Kuhn-Tucker conditions Notice first that in the case of N-player oligopolies $S_k = \{x_k | x_k \geq 0,\ L_k - x_k \geq 0\}$, so we select

$$\mathbf{g}_k(x_k) = \begin{pmatrix} x_k \\ L_k - x_k \end{pmatrix} , \tag{10.67}$$

and since the payoff functions are

$$f_k(x_1, \ldots, x_N) = x_k p(x_k + s_k) - c_k(x_k) \,, \tag{10.68}$$

the Kuhn-Tucker conditions (10.9) have the following form. The components of the 2-dimensional vectors $\mathbf{u}_k$ will be denoted by $u_k^{(1)}$ and $u_k^{(2)}$. So we have for $k = 1, 2, \ldots, N$,

$$\begin{aligned} u_k^{(1)}, u_k^{(2)} &\geq 0 \\ x_k &\geq 0 \\ L_k - x_k &\geq 0 \\ p(\textstyle\sum_{l=1}^N x_l) + x_k p'(\sum_{l=1}^N x_l) - c_k'(x_k) + (u_k^{(1)}, u_k^{(2)}) \binom{1}{-1} &= 0 \\ u_k^{(1)} x_k + u_k^{(2)}(L_k - x_k) &= 0 \,. \end{aligned} \tag{10.69}$$

One might either look for feasible solutions of these relations or rewrite them as the optimization problem (10.10), which has the following special form in this case:

$$\begin{array}{ll} \text{minimize} & \sum_{k=1}^{N}(u_k^{(1)}x_k + u_k^{(2)}(L_k - x_k)) \\ \text{subject to} & u_k^{(1)}, u_k^{(2)} \geq 0 \\ & x_k \geq 0 \\ & L_k - x_k \geq 0 \\ & p(\sum_{l=1}^{N} x_l) + x_k p'(\sum_{l=1}^{N} x_l) - c_k'(x_k) + u_k^{(1)} - u_k^{(2)} = 0 \\ & \qquad\qquad (k = 1, 2, \ldots, N). \end{array} \tag{10.70}$$

Computational cost in solving (10.69) or (10.70) depends on the type of functions p and c_k. No general characterization can be given.

Example 10.23 In the case of the three-person oligopoly introduced in Example 10.21 we have

$$\begin{array}{ll} \text{minimize} & \displaystyle\sum_{k=1}^{3}(u_k^{(1)}x_k + u_k^{(2)}(1 - x_k)) \\ \text{subject to} & u_k^{(1)}, u_k^{(2)} \geq 0 \\ & x_k \geq 0 \\ & 1 - x_k \geq 0 \\ & 1 - 2s - s^2 - 2x_k - 2x_k s - 3kx_k^2 + u_k^{(1)} - u_k^{(2)} = 0 \\ & x_1 + x_2 + x_3 = s. \end{array}$$

A professional optimization software was used to obtain the optimal solutions:

$$x_1^\star \approx 0.1077, \quad x_2^\star \approx 0.0986, \quad x_3^\star \approx 0.0919\ ,$$

and all $u_k^{(1)} = u_k^{(2)} = 0$.

Reduction to complementarity problems If $(x_1^\star, \ldots, x_N^\star)$ is an equilibrium of an N-person oligopoly, then with fixed $x_1^\star, \ldots, x_{k-1}^\star, x_{k+1}^\star, \ldots, x_N^\star$, $x_k = x_k^\star$ maximizes the payoff f_k of player $\mathcal{P}_k$. Assuming that condition 1–3 are satisfied, f_k is concave in x_k, so $x_k^\star$ maximizes f_k if and only if at the equilibrium

$$\frac{\partial f_k}{\partial x_k}(\mathbf{x}^\star) = \begin{cases} \leq 0, & \text{if} \quad x_k^\star = 0, \\ = 0, & \text{if} \quad 0 < x_k^\star < L_k, \\ \geq 0, & \text{if} \quad x_k^\star = L_k. \end{cases}$$

So introduce the slack variables

$$z_k = \begin{cases} = 0, & \text{if} \quad x_k > 0, \\ \geq 0, & \text{if} \quad x_k = 0 \end{cases}$$

$$v_k = \begin{cases} = 0, & \text{if} \quad x_k < L_k, \\ \geq 0, & \text{if} \quad x_k = L_k \end{cases}$$

and

$$w_k = L_k - x_k. \tag{10.71}$$

Then clearly at the equilibrium

$$\frac{\partial f_k}{\partial x_k}(\mathbf{x}) - v_k + z_k = 0 \tag{10.72}$$

and by the definition of the slack variables

$$z_k x_k = 0 \tag{10.73}$$

$$v_k w_k = 0, \tag{10.74}$$

and if we add the nonnegativity conditions

$$x_k, z_k, v_k, w_k \geq 0\ , \tag{10.75}$$

then we obtain a system of nonlinear relations (10.71)–(10.75) which are equivalent to the equilibrium problem.

We can next show that relations (10.71)–(10.75) can be rewritten as a nonlinear complementarity problem, for the solution of which standard methods are available. For this purpose introduce the notation

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix}, \quad \mathbf{L} = \begin{pmatrix} L_1 \\ L_2 \\ \vdots \\ L_N \end{pmatrix}, \quad \mathbf{h}(\mathbf{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\mathbf{x}) \\ \frac{\partial f_2}{\partial x_2}(\mathbf{x}) \\ \vdots \\ \frac{\partial f_N}{\partial x_N}(\mathbf{x}) \end{pmatrix},$$

$$\mathbf{t} = \begin{pmatrix} \mathbf{x} \\ \mathbf{v} \end{pmatrix}, \quad \text{and} \quad \mathbf{g}(\mathbf{t}) = \begin{pmatrix} -\mathbf{h}(\mathbf{x}) + \mathbf{v} \\ \mathbf{L} - \mathbf{x} \end{pmatrix},$$

then system (10.72)–(10.75) can be rewritten as

$$\begin{aligned} \mathbf{t} &\geq \mathbf{0} \\ \mathbf{g}(\mathbf{t}) &\geq \mathbf{0} \\ \mathbf{t}^T \mathbf{g}(\mathbf{t}) &= 0. \end{aligned} \tag{10.76}$$

This problem is the usual formulation of ***nonlinear complementarity*** problems. Notice that the last condition requires that in each component either $\mathbf{t}$ or $\mathbf{g}(\mathbf{t})$ or both must be zero.

The computational cost in solving problem (10.76) depends on the type of the involved functions and the choice of method.

Example 10.24 In the case of the 3-person oligopoly introduced and examined in the previous examples we have:

$$\mathbf{t} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix} \quad \text{and} \quad \mathbf{g}(\mathbf{t}) = \begin{pmatrix} -1 + 2\sum_{l=1}^3 x_l + (\sum_{l=1}^3 x_l)^2 + 2x_1 + 2x_1 \sum_{l=1}^3 x_l + 3x_1^2 + v_1 \\ -1 + 2\sum_{l=1}^3 x_l + (\sum_{l=1}^3 x_l)^2 + 2x_2 + 2x_2 \sum_{l=1}^3 x_l + 6x_2^2 + v_2 \\ -1 + 2\sum_{l=1}^3 x_l + (\sum_{l=1}^3 x_l)^2 + 2x_3 + 2x_3 \sum_{l=1}^3 x_l + 9x_3^2 + v_3 \\ 1 - x_1 \\ 1 - x_2 \\ 1 - x_3 \end{pmatrix}.$$

Linear oligopolies and quadratic programming In this section N-player oligopolies will be examined under the special condition that the price and all cost functions are linear :

$$p(s) = As + B, \quad c_k(x_k) = b_k x_k + c_k \quad (k = 1, 2, \ldots, N) ,$$

where B, b_k, and c_k are positive, but $A < 0$. Assume again that the strategy set of player $\mathcal{P}_k$ is the interval $[0, L_k]$. In this special case

$$f_k(x_1, \ldots, x_N) = x_k(Ax_1 + \cdots + Ax_N + B) - (b_k x_k + c_k) \tag{10.77}$$

for all k, therefore

$$\frac{\partial f_k}{\partial x_k}(\mathbf{x}) = 2Ax_k + A\sum_{l \neq k} x_l + B - b_k, \tag{10.78}$$

and relations (10.71)–(10.75) become more special:

$$\begin{aligned} 2Ax_k + A\sum_{l \neq k} x_l + B - b_k - v_k + z_k &= 0 \\ z_k x_k = v_k w_k &= 0 \\ x_k + w_k &= L_k \\ x_k, v_k, z_k, w_k &\geq 0, \end{aligned}$$

where we changed the order of them. Introduce the following vectors and matrixes:

$$\mathbf{Q} = \begin{pmatrix} 2A & A & \cdots & A \\ A & 2A & \cdots & A \\ \vdots & \vdots & & \vdots \\ A & A & \cdots & 2A \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B \\ B \\ \vdots \\ B \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix},$$

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix}, \quad \mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{pmatrix}, \quad \mathbf{z} = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{pmatrix}, \quad \text{and} \quad \mathbf{L} = \begin{pmatrix} L_1 \\ L_2 \\ \vdots \\ L_N \end{pmatrix} .$$

Then the above relations can be summarized as:

$$\begin{aligned} \mathbf{Qx} + \mathbf{B} - \mathbf{b} - \mathbf{v} + \mathbf{z} &= \mathbf{0} \\ \mathbf{x} + \mathbf{w} &= \mathbf{L} \\ \mathbf{x}^T\mathbf{z} = \mathbf{v}^T\mathbf{w} &= 0 \\ \mathbf{x}, \mathbf{v}, \mathbf{z}, \mathbf{w} &\geq \mathbf{0} . \end{aligned} \tag{10.79}$$

Next we prove that matrix $\mathbf{Q}$ is negative definite. With any nonzero vector $\mathbf{a} = (a_i)$,

$$\mathbf{a}^T\mathbf{Q}\mathbf{a} = 2A\sum_i a_i^2 + A\sum_i \sum_{j \neq i} a_i a_j = A(\sum_i a_i^2 + (\sum_i a_i)^2) < 0 ,$$

which proves the assertion.

Observe that relations (10.79) are the Kuhn-Tucker conditions of the strictly

concave quadratic programming problem:

$$\begin{array}{ll} \text{maximize} & \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} + (\mathbf{B}-\mathbf{b})\mathbf{x} \\ \text{subject to} & \mathbf{0} \le \mathbf{x} \le \mathbf{L}, \end{array} \tag{10.80}$$

and since the feasible set is a bounded linear polyhedron and the objective function is strictly concave, the Kuhn-Tucker conditions are sufficient and necessary. Consequently a vector $\mathbf{x}^\star$ is an equilibrium if and only if it is the unique optimal solution of problem (10.80). There are standard methods to solve problem (10.80) known from the literature.

Since (10.79) is a convex quadratic programming problem, several algorithms are available. Their costs are different, so computation cost depends on the particular method being selected.

Example 10.25 Consider now a duopoly (two-person oligopoly) where the price function is $p(s) = 10 - s$ and the cost functions are $c_1(x_1) = 4x_1 + 1$ and $c_2(x_2) = x_2 + 1$ with capacity limits $L_1 = L_2 = 5$. That is,

$$B = 10, \quad A = -1, \quad b_1 = 4, \quad b_2 = 1, \quad c_1 = c_2 = 1\ .$$

Therefore,

$$\mathbf{Q} = \begin{array}{cc} -2 & -1 \\ -1 & -2 \end{array}, \quad \mathbf{B} = \begin{array}{c} 10 \\ 10 \end{array}\Big), \quad \mathbf{b} = \begin{array}{c} 4 \\ 1 \end{array}\Big), \quad \mathbf{L} = \begin{array}{c} 5 \\ 5 \end{array}\Big),$$

so the quadratic programming problem can be written as:

$$\begin{array}{ll} \text{maximize} & \dfrac{1}{2}(-2x_1^2 - 2x_1x_2 - 2x_2^2) + 6x_1 + 9x_2 \\ \text{subject to} & 0 \le x_1 \le 5 \\ & 0 \le x_2 \le 5. \end{array}$$

It is easy to see by simple differentiation that the global optimum at the objective function without the constraints is reached at $x_1^\star = 1$ and $x_2^\star = 4$. They however satisfy the constraints, so they are the optimal solutions. Hence they provide the unique equilibrium of the duopoly.

Exercises

10.3-1 Consider a duopoly with $S_1 = S_2 = [0,1]$, $p(s) = 2 - s$ and costs $c_1(x) = c_2(x) = x^2 + 1$. Examine the convergence of the iteration scheme (10.55).

10.3-2 Select $n = 2$, $S_1 = S_2 = [0, 1.5]$, $c_k(x_k) = 0.5x_k$ $(k = 1, 2)$ and

$$p(s) = \begin{cases} 1.75 - 0.5s, & \text{if} \quad 0 \le s \le 1.5, \\ 2.5 - s, & \text{if} \quad 1.5 \le s \le 2.5, \\ 0, & \text{if} \quad 2.5 \le s \ . \end{cases}$$

Show that there are infinitely many equilibria:

$$\{(x_1^\star, x_2^\star) | 0.5 \le x_1 \le 1, \quad 0.5 \le x_2 \le 1, \quad x_1 + x_2 = 1.5\}.$$

10.3-3 Consider the duopoly of Exercise 10.3-1 above. Find the best reply mappings of the players and determine the equilibrium.

10.3-4 Consider again the duopoly of the previous problem.
(a) Construct the one-dimensional fixed point problem of mapping (10.62) and solve it to obtain the equilibrium.
(b) Formulate the Kuhn-Tucker equations and inequalities (10.69).
(c) Formulate the complementarity problem (10.76) in this case.

Chapter notes

(Economic) Nobel Prize was given only once, in 1994 in the field of game theory. One of the winner was John Nash , who received this honor for his equilibrium concept, which was introduced in 1951 [176].

Backward induction is a more restrictive equilibrium concept. It was developed by Kuhn and can be found in [142]. Since it is more restrictive equilibrium, it is also a Nash equilibrium.

The existence and computation of equilibria can be reduced to those of fixed points. the different variants of fixed point theorems-such as that of Brouwer [31], Kakutani[123], Tarski [240] are successfully used to prove existence in many game classes. The article [180] uses the fixed point theorem of Kakutani. The books [239] and [71] discuss computer methods for computing fixed points. The most popular existence result is the well known theorem of Nikaido and Isoda [180].

The Fan inequality is discussed in the book of Aubin [14]. The Kuhn-Tucker conditions are presented in the book of Martos [161]. By introducing slack and surplus variables the Kuhn-Tucker conditions can be rewritten as a system of equations. For their computer solutions well known methods are available ([239] and [161]).

The reduction of bimatrix games to mixed optimization problems is presented in the papers of Mills [169] and Shapiro [222]. The reduction to quadratic programming problem is given in ([159]).

The method of fictitious play is discussed in the paper of Robinson [201]. In

order to use the Neumann method we have to solve a system of nonlinear ordinary differential equations. The Runge–Kutta method is the most popular procedure for doing it. It can be found in [239].

The paper of Rosen [202] introduces diagonally strictly concave games. The computer method to find the equilibria of N-person concave games is introduced in Zuhovitsky et al. [265].

The different extensions and generalizations of the classical Cournot model can be found in the books of Okuguchi and Szidarovszky [182, 183]. The proof of Theorem 10.20 is given in [238]. For the proof of lemma (10.58) see the monograph [183]. The bisection method is described in [239]. The paper [125] contains methods which are applicable to solve nonlinear complementarity problems. The solution of problem (10.80) is discussed in the book of Hadley [98].

The book of von Neumann and Morgenstern [178] is considered the classical textbook of game theory. There is a large variety of game theory textbooks (see for example [71]).

11. Recurrences

The recursive definition of the Fibonacci numbers is well-known: if F_n is the n^{th} Fibonacci number then

$$F_0 = 0, \quad F_1 = 1 \ ,$$
$$F_{n+2} = F_{n+1} + F_n, \qquad \text{if} \quad n \geq 0 \ .$$

We are interested in an explicit form of the numbers F_n for all natural numbers n. Actually, the problem is to solve an equation where the unknown is given recursively, in which case the equation is called a ***recurrence equation***. The solution can be considered as a function over natural numbers, because F_n is defined for all n. Such recurrence equations are also known as ***difference equations***, but could be named as ***discrete differential equations*** for their similarities to differential equations.

Definition 11.1 *A* ***k^{th}th order recurrence equation*** $(k \geq 1)$ *is an equation of the form*

$$f(x_n, x_{n+1}, \ldots, , x_{n+k}) = 0, \qquad n \geq 0 \ , \tag{11.1}$$

where x_n must be given in an explicit form.

For a unique determination of x_n, k initial values must be given. Usually these values are $x_0, x_1, \ldots, x_{k-1}$. These can be considered as ***initial conditions***. In case of the equation for Fibonacci-numbers, which is of second order, two initial values must be given.

The sequence $x_n = g(n)$ satisfying equation (11.1) and the corresponding initial conditions is called a ***particular solution***. If all particular solutions of equation (11.1) can be obtained from the sequence $x_n = h(n, C_1, C_2, \ldots, C_k)$, by adequately choosing of the constants $C_1, C_2, \ldots, C_k$, then this sequence x_n is a ***general solution***.

Solving recurrence equations is not an easy task. In this chapter we will discuss methods which can be used in special cases. For simplicity of writing we will use the notation x_n instead of $x(n)$ as it appears in several books (sequences can be considered as functions over natural numbers).

The chapter is divided into three sections. In Section 11.1 we deal with solving linear recurrence equations, in Section 11.2 with generating functions and their use in solving recurrence equations and in Section 11.3 we focus our attention on the numerical solution of recurrence equations.

11.1. Linear recurrence equations

If the recurrence equation is of the form

$$f_0(n)x_n + f_1(n)x_{n+1} + \cdots + f_k(n)x_{n+k} = f(n), \quad n \geq 0 \,,$$

where $f, f_0, f_1, \ldots, f_k$ are functions defined over natural numbers, $f_0, f_k \neq 0$, and x_n must be given explicitly, then the recurrence equation is ***linear.*** If f is the zero function, then the equation is ***homogeneous,*** otherwise ***nonhomogeneous.*** If all the functions $f_0, f_1, \ldots, f_k$ are constant, the equation is called a ***linear recurrence equation with constant coefficients.***

11.1.1. Linear homogeneous equations with constant coefficients

Let the equation be

$$a_0x_n + a_1x_{n+1} + \cdots + a_kx_{n+k} = 0, \qquad n \geq k \,, \tag{11.2}$$

where $a_0, a_1, \ldots, a_k$ are real constants, $a_0, a_k \neq 0,\ k \geq 1$. If k initial conditions are given (usually $x_0, x_1, \ldots, x_{k-1}$), then the general solution of this equation can be uniquely given.

To solve the equation let us consider its ***characteristic equation***

$$a_0 + a_1r + \cdots + a_{k-1}r^{k-1} + a_kr^k = 0 \,, \tag{11.3}$$

a polynomial equation with real coefficients. This equation has k roots in the field of complex numbers. It can easily be seen after a simple substitution that if r_0 is a real solution of the characteristic equation, then $C_0r_0^n$ is a solution of (11.2), for arbitrary C_0.

The general solution of equation (11.2) is

$$x_n = C_1x_n^{(1)} + C_2x_n^{(2)} + \cdots + C_kx_n^{(k)} \,,$$

where $x_n^{(i)}$ $(i = 1, 2, \ldots, k)$ are the linearly independent solutions of equation (11.2). The constants $C_1, C_2, \ldots, C_k$ can be determined from the initial conditions by solving a system of k equations.

The linearly independent solutions are supplied by the roots of the characteristic equation in the following way. A ***fundamental solution*** of equation (11.2) can be associated with each root of the characteristic equation. Let us consider the following cases.

Distinct real roots

Let $r_1, r_2, \ldots, r_p$ be distinct real roots of the characteristic equation. Then

$$r_1^n,\ r_2^n,\ \ldots,\ r_p^n$$

are solutions of equation (11.2), and

$$C_1r_1^n + C_2r_2^n + \cdots + C_pr_p^n \tag{11.4}$$

is also a solution, for arbitrary constants $C_1, C_2, \ldots, C_p$. If $p = k$, then (11.4) is the general solution of the recurrence equation.

Example 11.1 Solve the recurrence equation

$$x_{n+2} = x_{n+1} + x_n, \quad x_0 = 0, \; x_1 = 1 \; .$$

The corresponding characteristic equation is

$$r^2 - r - 1 = 0 \; ,$$

with the solutions

$$r_1 = \frac{1+\sqrt{5}}{2} \; , \quad r_2 = \frac{1-\sqrt{5}}{2} \; .$$

These are distinct real solutions, so the general solution of the equation is

$$x_n = C_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + C_2 \left(\frac{1-\sqrt{5}}{2}\right)^n \; .$$

The constants C_1 and C_2 can be determined using the initial conditions. From $x_0 = 0$, $x_1 = 1$ the following system of equations can be obtained.

$$\begin{aligned} C_1 + C_2 &= 0 \; , \\ C_1 \frac{1+\sqrt{5}}{2} + C_2 \frac{1-\sqrt{5}}{2} &= 1 \; . \end{aligned}$$

The solution of this system of equations is $C_1 = 1/\sqrt{5}$, $C_2 = -1/\sqrt{5}$. Therefore the general solution is

$$x_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n \; ,$$

which is the nth Fibonacci number F_n.

Multiple real roots

Let r be a real root of the characteristic equation with multiplicity p. Then

$$r^n, \; nr^n, \; n^2 r^n, \; \ldots, \; n^{p-1} r^n$$

are solutions of equation (11.2) (fundamental solutions corresponding to r), and

$$\left(C_0 + C_1 n + C_2 n^2 + \cdots + C_{p-1} n^{p-1}\right) r^n \tag{11.5}$$

is also a solution, for any constants $C_0, C_1, \ldots, C_{p-1}$. If the characteristic equation has no other solutions, then (11.5) is a general solution of the recurrence equation.

Example 11.2 Solve the recurrence equation

$$x_{n+2} = 4x_{n+1} - 4x_n, \quad x_0 = 1, \; x_1 = 3 \; .$$

The characteristic equation is

$$r^2 - 4r + 4 = 0 \; ,$$

with $r = 2$ a solution with multiplicity 2. Then

$$x_n = (C_0 + C_1 n) 2^n$$

is a general solution of the recurrence equation.

From the initial conditions we have

$$\begin{aligned} C_0 &= 1\,, \\ 2C_0 + 2C_1 &= 3\,. \end{aligned}$$

From this system of equations $C_0 = 1$, $C_1 = 1/2$, so the general solution is

$$x_n = \left(1 + \frac{1}{2}n\right) 2^n \quad \text{or} \quad x_n = (n+2)2^{n-1}\,.$$

Distinct complex roots

If the complex number $a(\cos b + i \sin b)$, written in trigonometric form, is a root of the characteristic equation, then its conjugate $a(\cos b - i \sin b)$ is also a root, because the coefficients of the characteristic equation are real numbers. Then

$$a^n \cos bn \quad \text{and} \quad a^n \sin bn$$

are solutions of equation (11.2) and

$$C_1 a^n \cos bn + C_2 a^n \sin bn \tag{11.6}$$

is also a solution, for any constants C_1 and C_2. If these are the only solutions of a second order characteristic equation, then (11.6) is a general solution.

Example 11.3 Solve the recurrence equation

$$x_{n+2} = 2x_{n+1} - 2x_n, \quad x_0 = 0,\ x_1 = 1\,.$$

The corresponding characteristic equation is

$$r^2 - 2r + 2 = 0\,,$$

with roots $1 + i$ and $1 - i$. These can be written in trigonometric form as $\sqrt{2}(\cos(\pi/4) + i \sin(\pi/4))$ and $\sqrt{2}(\cos(\pi/4) - i \sin(\pi/4))$. Therefore

$$x_n = C_1(\sqrt{2})^n \cos\frac{n\pi}{4} + C_2(\sqrt{2})^n \sin\frac{n\pi}{4}$$

is a general solution of the recurrence equation. From the initial conditions

$$\begin{aligned} C_1 &= 0\,, \\ C_1\sqrt{2}\cos\frac{\pi}{4} + C_2\sqrt{2}\sin\frac{\pi}{4} &= 1\,. \end{aligned}$$

Therefore $C_1 = 0$, $C_2 = 1$. Hence the general solution is

$$x_n = \left(\sqrt{2}\right)^n \sin\frac{n\pi}{4}\,.$$

Multiple complex roots

If the complex number written in trigonometric form as $a(\cos b + i \sin b)$ is a root of the characteristic equation with multiplicity p, then its conjugate $a(\cos b - i \sin b)$ is also a root with multiplicity p.

Then

$$a^n \cos bn,\ na^n \cos bn,\ \ldots,\ n^{p-1}a^n \cos bn$$

and

$$a^n \sin bn,\ na^n \sin bn,\ \ldots,\ n^{p-1}a^n \sin bn$$

are solutions of the recurrence equation (11.2). Then

$$(C_0 + C_1 n + \cdots + C_{p-1}n^{p-1})a^n \cos bn + (D_0 + D_1 n + \cdots + D_{p-1}n^{p-1})a^n \sin bn$$

is also a solution, where $C_0, C_1, \ldots, C_{p-1}, D_0, D_1, \ldots, D_{p-1}$ are arbitrary constants, which can be determined from the initial conditions. This solution is general if the characteristic equation has no other roots.

Example 11.4 Solve the recurrence equation

$$x_{n+4} + 2x_{n+2} + x_n = 0,\ \ x_0 = 0,\ x_1 = 1,\ x_2 = 2,\ x_3 = 3\ .$$

The characteristic equation is

$$r^4 + 2r^2 + 1 = 0\ ,$$

which can be written as $(r^2+1)^2 = 0$. The complex numbers i and $-i$ are double roots. The trigonometric form of these are

$$i = \cos\frac{\pi}{2} + i\sin\frac{\pi}{2},\ \text{ and }\ -i = \cos\frac{\pi}{2} - i\sin\frac{\pi}{2}$$

respectively. Therefore the general solution is

$$x_n = (C_0 + C_1 n)\cos\frac{n\pi}{2} + (D_0 + D_1 n)\sin\frac{n\pi}{2}\ .$$

From the initial conditions we obtain

$$\begin{aligned} C_0 &= 0\ , \\ (C_0 + C_1)\cos\frac{\pi}{2} + (D_0 + D_1)\sin\frac{\pi}{2} &= 1\ , \\ (C_0 + 2C_1)\cos\pi + (D_0 + 2D_1)\sin\pi &= 2\ , \\ (C_0 + 3C_1)\cos\frac{3\pi}{2} + (D_0 + 3D_1)\sin\frac{3\pi}{2} &= 3\ , \end{aligned}$$

that is

$$\begin{aligned} C_0 &= 0\ , \\ D_0 + D_1 &= 1\ , \\ -2C_1 &= 2\ , \\ -D_0 - 3D_1 &= 3\ . \end{aligned}$$

Solving this system of equations $C_0 = 0$, $C_1 = -1$, $D_0 = 3$ and $D_1 = -2$. Thus the general solution is

$$x_n = (3 - 2n)\sin\frac{n\pi}{2} - n\cos\frac{n\pi}{2}\ .$$

Using these four cases all linear homogeneous equations with constant coefficients can be solved, if we can solve their characteristic equations.

Example 11.5 Solve the recurrence equation

$$x_{n+3} = 4x_{n+2} - 6x_{n+1} + 4x_n, \quad x_0 = 0,\ x_1 = 1,\ x_2 = 1\ .$$

The characteristic equation is

$$r^3 - 4r^2 + 6r - 4 = 0\ ,$$

with roots 2, $1+i$ and $1-i$. Therefore the general solution is

$$x_n = C_1 2^n + C_2\ \sqrt{2}^{\ n} \cos\frac{n\pi}{4} + C_3\ \sqrt{2}^{\ n} \sin\frac{n\pi}{4}\ .$$

After determining the constants we obtain

$$x_n = -2^{n-1} + \frac{\sqrt{2}^{\ n}}{2}\ \cos\frac{n\pi}{4} + 3\sin\frac{n\pi}{4}\quad .$$

The general solution

The characteristic equation of the kth order linear homogeneous equation (11.2) has k roots in the field of complex numbers, which are not necessarily distinct. Let these roots be the following:

r_1 real, with multiplicity p_1 $(p_1 \geq 1)$,
r_2 real, with multiplicity p_2 $(p_2 \geq 1)$,
...
r_t real, with multiplicity p_t $(p_t \geq 1)$,
$s_1 = a_1(\cos b_1 + i \sin b_1)$ complex, with multiplicity q_1 $(q_1 \geq 1)$,
$s_2 = a_2(\cos b_2 + i \sin b_2)$ complex, with multiplicity q_2 $(q_2 \geq 1)$,
...
$s_m = a_m(\cos b_m + i \sin b_m)$ complex, with multiplicity q_m $(q_m \geq 1)$.

Since the equation has k roots, $p_1 + p_2 + \cdots + p_t + 2(q_1 + q_2 + \cdots + q_m) = k$.
In this case the general solution of equation (11.2) is

$$\begin{aligned} x_n &= \sum_{j=1}^{t} \left(C_0^{(j)} + C_1^{(j)} n + \cdots + C_{p_j-1}^{(j)} n^{p_j-1}\right) r_j^n \\ &+ \sum_{j=1}^{m} \left(D_0^{(j)} + D_1^{(j)} n + \cdots + D_{q_j-1}^{(j)} n^{q_j-1}\right) a_j^n \cos b_j n \\ &+ \sum_{j=1}^{m} \left(E_0^{(j)} + E_1^{(j)} n + \cdots + E_{q_j-1}^{(j)} n^{q_j-1}\right) a_j^n \sin b_j n\ , \end{aligned} \tag{11.7}$$

where
$C_0^{(j)},\ C_1^{(j)},\ \ldots,\ C_{p_j-1}^{(j)},\ j = 1, 2, \ldots, t$,
$D_0^{(l)},\ E_0^{(l)},\ D_1^{(l)},\ E_1^{(l)},\ \ldots,\ D_{p_l-1}^{(l)},\ E_{p_l-1}^{(l)},\ l = 1, 2, \ldots, m$ are constants, which can be determined from the initial conditions.

The above statements can be summarised in the following theorem.

Theorem 11.2 *Let $k \geq 1$ be an integer and $a_0,\ a_1,\ \ldots, a_k$ real numbers with $a_0,\ a_k \neq 0$. The general solution of the linear recurrence equation (11.2) can be obtained as a linear combination of the terms $n^j r_i^n$, where r_i are the roots of the characteristic equation (11.3) with multiplicity p_i $(0 \leq j < p_i)$ and the coefficients of the linear combination depend on the initial conditions.*

The proof of the theorem is left to the Reader (see Exercise 11.1-5.).

The algorithm for the general solution is the following.

LINEAR-HOMOGENEOUS

1 determine the characteristic equation of the recurrence equation
2 find all roots of the characteristic equation with their multiplicities
3 find the general solution (11.7) based on the roots
4 determine the constants of (11.7) using the initial conditions, if these exists.

11.1.2. Linear nonhomogeneous recurrence equations with constant coefficients

Consider the linear nonhomogeneous recurrence equation with constant coefficients

$$a_0 x_n + a_1 x_{n+1} + \cdots + a_k x_{n+k} = f(n)\ , \tag{11.8}$$

where $a_0, a_1, \ldots, a_k$ are real constants, $a_0, a_k \neq 0,\ k \geq 1$, and f is not the zero function.

The corresponding linear homogeneous equation (11.2) can be solved using Theorem 11.2. If a particular solution of equation (11.8) is known, then equation (11.8) can be solved.

Theorem 11.3 *Let $k \geq 1$ be an integer, $a_0,\ a_1,\ \ldots, a_k$ real numbers, $a_0,\ a_k \neq 0$. If $x_n^{(1)}$ is a particular solution of the linear nonhomogeneous equation (11.8) and $x_n^{(0)}$ is a general solution of the linear homogeneous equation (11.2), then*

$$x_n = x_n^{(0)} + x_n^{(1)}$$

is a general solution of the equation (11.8).

The proof of the theorem is left to the Reader (see Exercise 11.1-6).

Example 11.6 Solve the recurrence equation

$$x_{n+2} + x_{n+1} - 2x_n = 2^n,\ \ x_0 = 0,\ x_1 = 1\ .$$

First we solve the homogeneous equation

$$x_{n+2} + x_{n+1} - 2x_n = 0\ ,$$

and obtain the general solution

$$x_n^{(0)} = C_1(-2)^n + C_2\ ,$$

$f(n)$	$x_n^{(1)}$
$n^p a^n$	$(C_0 + C_1 n + \cdots + C_p n^p)a^n$
$a^n n^p \sin bn$	$(C_0 + C_1 n + \cdots + C_p n^p)a^n \sin bn + (D_0 + D_1 n + \cdots + D_p n^p)a^n \cos bn$
$a^n n^p \cos bn$	$(C_0 + C_1 n + \cdots + C_p n^p)a^n \sin bn + (D_0 + D_1 n + \cdots + D_p n^p)a^n \cos bn$

Figure 11.1. The form of particular solutions.

since the roots of the characteristic equation are -2 and 1. It is easy to see that

$$x_n = C_1(-2)^n + C_2 + 2^{n-2}$$

is a solution of the nonhomogeneous equation. Therefore the general solution is

$$x_n = -\frac{1}{4}(-2)^n + 2^{n-2} \quad \text{or} \quad x_n = \frac{2^n - (-2)^n}{4} \ ,$$

The constants C_1 and C_2 can be determined using the initial conditions. Thus,

$$x_n = \begin{array}{ll} 0, & \text{if } n \text{ is even} \ , \\ 2^{n-1}, & \text{if } n \text{ is odd} \ . \end{array}$$

A particular solution can be obtained using the *method of variation of constants.* However, there are cases when there is an easier way of finding a particular solution. In Figure 11.1 we can see types of functions $f(n)$, for which a particular solution $x_n^{(1)}$ can be obtained in the given form in the table. The constants can be obtained by substitutions.

In the previous example $f(n) = 2^n$, so the first case can be used with $a = 2$ and $p = 0$. Therefore we try to find a particular solution of the form $C_0 2^n$. After substitution we obtain $C_0 = 1/4$, thus the particular solution is

$$x_n^{(1)} = 2^{n-2} \ .$$

Exercises

11.1-1 Solve the recurrence equation

$$H_n = 2H_{n-1} + 1, \ \text{ if } n \geq 1, \ \text{ and } \quad H_0 = 0 \ .$$

(Here H_n is the optimal number of moves in the problem of the Towers of Hanoi.)

11.1-2 Analyse the problem of the Towers of Hanoi if n discs have to be moved from stick A to stick C in such a way that no disc can be moved *directly* from A to C and vice versa.

Hint. Show that if the optimal number of moves is denoted by M_n, and $n \geq 1$, then $M_n = 3M_{n-1} + 2$.

11.1-3 Solve the recurrence equation

$$(n+1)R_n = 2(2n-1)R_{n-1}, \quad \text{if } n \geq 1, \text{ and} \quad R_0 = 1 \ .$$

11.1-4 Solve the linear nonhomogeneous recurrence equation

$$x_n = 2^n - 2 + 2x_{n-1}, \quad \text{if } n \geq 2, \quad \text{and} \quad x_1 = 0 \ .$$

Hint. Try to find a particular solution of the form $C_1 n 2^n + C_2$.

11.1-5⋆ Prove Theorem 11.2.
11.1-6 Prove Theorem 11.3.

11.2. Generating functions and recurrence equations

Generating functions can be used, among others, to solve recurrence equations, count objects (e.g. binary trees), prove identities and solve partition problems. Counting the number of objects can be done by stating and solving recurrence equations. These equations are usually not linear, and generating functions can help us in solving them.

11.2.1. Definition and operations

Associate a series with the infinite sequence $(a_n)_{n\geq 0} = \langle a_0, a_1, a_2, \ldots, a_n, \ldots \rangle$ the following way

$$A(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_n z^n + \cdots = \sum_{n\geq 0} a_n z^n \ .$$

This is called the ***generating function*** of the sequence $(a_n)_{n\geq 0}$.

For example, in case of the Fibonacci numbers this generating function is

$$F(z) = \sum_{n\geq 0} F_n z^n = z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + \cdots \ .$$

Multiplying both sides of the equation by z, then by z^2, we obtain

$$\begin{aligned} F(z) &= F_0 + F_1 z + F_2 z^2 + F_3 z^3 + \cdots + F_n z^n \ + \cdots \ , \\ zF(z) &= F_0 z + F_1 z^2 + F_2 z^3 + \cdots + F_{n-1} z^n + \cdots \ , \\ z^2 F(z) &= F_0 z^2 + F_1 z^3 + \cdots + F_{n-2} z^n + \cdots \ . \end{aligned}$$

If we subtract the second and the third equations from the first one term by term, then use the defining formula of the Fibonacci numbers, we get

$$F(z)(1 - z - z^2) = z \ ,$$

that is

$$F(z) = \frac{z}{1 - z - z^2} \ . \tag{11.9}$$

The correctness of these operations can be proved mathematically, but here we do not want to go into details. The formulae obtained using generating functions can usually also be proved using other methods.

Let us consider the following generating functions

$$A(z) = \sum_{n \geq 0} a_n z^n \text{ and } B(z) = \sum_{n \geq 0} b_n z^n.$$

The generating functions $A(z)$ and $B(z)$ are *equal,* if and only if $a_n = b_n$ for all n natural numbers.

Now we define the following operations with the generating functions: addition, multiplication by real number, shift, multiplication, derivation and integration.

Addition and multiplication by real number

$$\alpha A(z) + \beta B(z) = \sum_{n \geq 0} (\alpha a_n + \beta b_n) z^n \ .$$

Shift

The generating function

$$z^k A(z) = \sum_{n \geq 0} a_n z^{n+k} = \sum_{n \geq k} a_{n-k} z^n$$

represents the sequence $< \underbrace{0, 0, \ldots, 0}_{k}, a_0, a_1, \ldots >$, while the generating function

$$\frac{1}{z^k}(A(z) - a_0 - a_1 z - a_2 z^2 - \cdots - a_{k-1} z^{k-1}) = \sum_{n \geq k} a_n z^{n-k} = \sum_{n \geq 0} a_{k+n} z^n$$

represents the sequence $< a_k, a_{k+1}, a_{k+2}, \ldots >$.

Example 11.7 Let $A(z) = 1 + z + z^2 + \cdots$. Then

$$\frac{1}{z} \ A(z) - 1 \ = A(z) \quad \text{and} \quad A(z) = \frac{1}{1 - z} \ .$$

Multiplication
If $A(z)$ and $B(z)$ are generating functions, then

$$\begin{aligned} A(z)B(z) &= (a_0 + a_1 z + \cdots + a_n z^n + \cdots)(b_0 + b_1 z + \cdots + b_n z^n + \cdots) \\ &= a_0 b_0 + (a_0 b_1 + a_1 b_0) z + (a_0 b_2 + a_1 b_1 + a_2 b_0) z^2 + \cdots \\ &= \sum_{n \geq 0} s_n z^n, \end{aligned}$$

where $s_n = \sum_{k=0}^{n} a_k b_{n-k}$.

Special case. If $b_n = 1$ for all natural numbers n, then

$$A(z) \frac{1}{1-z} = \sum_{n \geq 0} \left(\sum_{k=0}^{n} a_k \right) z^n \ . \tag{11.10}$$

If, in addition, $a_n = 1$ for all n, then

$$\frac{1}{(1-z)^2} = \sum_{n \geq 0} (n+1) z^n \ . \tag{11.11}$$

Derivation

$$A'(z) = a_1 + 2a_2 z + 3a_3 z^2 + \cdots = \sum_{n \geq 0} (n+1) a_{n+1} z^n \ .$$

Example 11.8 After differentiating both sides of the generating function

$$A(z) = \sum_{n \geq 0} z^n = \frac{1}{1-z},$$

we obtain

$$A'(z) = \sum_{n \geq 1} n z^{n-1} = \frac{1}{(1-z)^2} \ .$$

Integration

$$\int_0^z A(t) dt = a_0 z + \frac{1}{2} a_1 z^2 + \frac{1}{3} a_2 z^3 + \cdots = \sum_{n \geq 1} \frac{1}{n} a_{n-1} z^n \ .$$

Example 11.9 Let

$$\frac{1}{1-z} = 1 + z + z^2 + z^3 + \cdots$$

After integrating both sides we get

$$\ln \frac{1}{1-z} = z + \frac{1}{2} z^2 + \frac{1}{3} z^3 + \cdots = \sum_{n \geq 1} \frac{1}{n} z^n \ .$$

Multiplying the above generating functions we obtain

$$\frac{1}{1-z}\ln\frac{1}{1-z} = \sum_{n\geq 1} H_n z^n \ ,$$

where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ $(H_0 = 0, \ \ H_1 = 1)$ are the so-called *harmonic numbers*.

Changing the arguments

Let $A(z) = \sum_{n\geq 0} a_n z^n$ represent the sequence $\langle a_0, a_1, a_2, \ldots\rangle$, then $A(cz) = \sum_{n\geq 0} c^n a_n z^n$ represents the sequence $\langle a_0, ca_1, c^2 a_2, \ldots c^n a_n, \ldots\rangle$. The following statements holds

$$\frac{1}{2}\Big(A(z) + A(-z)\Big) = a_0 + a_2 z^2 + \cdots + a_{2n} z^{2n} + \cdots \ ,$$

$$\frac{1}{2}\Big(A(z) - A(-z)\Big) = a_1 z + a_3 z^3 + \cdots + a_{2n-1} z^{2n-1} + \cdots \ .$$

Example 11.10 Let $A(z) = 1 + z + z^2 + z^3 + \cdots = \dfrac{1}{1-z}$. Then

$$1 + z^2 + z^4 + \cdots = \frac{1}{2}\ A(z) + A(-z) \ = \frac{1}{2}\quad \frac{1}{1-z} + \frac{1}{1+z} \quad = \frac{1}{1-z^2} \ ,$$

which can also be obtained by substituting z by z^2 in $A(z)$. We can obtain the sum of the odd power terms in the same way,

$$z + z^3 + z^5 + \cdots = \frac{1}{2}\ A(z) - A(-z) \ = \frac{1}{2}\quad \frac{1}{1-z} - \frac{1}{1+z} \quad = \frac{z}{1-z^2} \ .$$

Using generating functions we can obtain interesting formulae. For example, let $A(z) = 1/(1-z) = 1 + z + z^2 + z^3 + \cdots$. Then $zA\big(z(1+z)\big) = F(z)$, which is the generating function of the Fibonacci numbers. From this

$$zA\big(z(1+z)\big) = z + z^2(1+z) + z^3(1+z)^2 + z^4(1+z)^3 + \cdots \ .$$

The coefficient of z^{n+1} on the left-hand side is F_{n+1}, that is the $(n+1)$th Fibonacci number, while the coefficient of z^{n+1} on the right-hand side is

$$\sum_{k\geq 0} \binom{n-k}{k} \ ,$$

after using the binomial formula in each term. Hence

$$F_{n+1} = \sum_{k\geq 0} \binom{n-k}{k} = \sum_{k=0}^{\lfloor \frac{n+1}{2} \rfloor} \binom{n-k}{k} \ . \tag{11.12}$$

Remember that the binomial formula can be generalised for all real r, namely

$$(1+z)^r = \sum_{n\geq 0} \binom{r}{n} z^n \ ,$$

which is the generating function of the binomial coefficients. Here $\binom{r}{n}$ is a generalisation of the number of combinations for any real number r, that is

$$\binom{r}{n} = \begin{cases} \dfrac{r(r-1)(r-2)\dots(r-n+1)}{n(n-1)\dots 1}, & \text{if } n>0 \ , \\ 1, & \text{fi } n=0 \ , \\ 0, & \text{if } n<0 \ . \end{cases}$$

We can obtain useful formulae using this generalisation for negative r. Let

$$\frac{1}{(1-z)^m} = (1-z)^{-m} = \sum_{k\geq 0} \binom{-m}{k} (-z)^k \ .$$

Since, by a simple computation, we get

$$\binom{-m}{k} = (-1)^k \binom{m+k-1}{k} \ ,$$

the following formula can be obtained

$$\frac{1}{(1-z)^{m+1}} = \sum_{k\geq 0} \binom{m+k}{k} z^k \ .$$

Then

$$\frac{z^m}{(1-z)^{m+1}} = \sum_{k\geq 0} \binom{m+k}{k} z^{m+k} = \sum_{k\geq 0} \binom{m+k}{m} z^{m+k} = \sum_{k\geq 0} \binom{k}{m} z^k \ ,$$

and

$$\sum_{k\geq 0} \binom{k}{m} z^k = \frac{z^m}{(1-z)^{m+1}} \ , \tag{11.13}$$

where m is a natural number.

11.2.2. Solving recurrence equations by generating functions

If the generating function of the general solution of a recurrence equation to be solved can be expanded in such a way that the coefficients are in closed form, then this method is successful.

Let the recurrence equation be

$$F(x_n, x_{n-1}, \dots, x_{n-k}) = 0 \ . \tag{11.14}$$

To solve it, let us consider the generating function

$$X(z) = \sum_{n\geq 0} x_n z^n .$$

If (11.14) can be written as $G\big(X(z)\big) = 0$ and can be solved for $X(z)$, then $X(z)$ can be expanded into series in such a way that x_n can be written in closed form, equation (11.14) can be solved.

Now we give a general method for solving linear nonhomogeneous recurrence equations. After this we give three examples for the nonlinear case. In the first two examples the number of elements in some sets of binary trees, while in the third example the number of leaves of binary trees is computed. The corresponding recurrence equations (11.15), (11.17) and (11.18) will be solved using generating functions.

Linear nonhomogeneous recurrence equations with constant coefficients

Multiply both sides of equation (11.8) by z^n. Then

$$a_0 x_n z^n + a_1 x_{n+1} z^n + \cdots + a_k x_{n+k} z^n = f(n) z^n \ .$$

Summing up both sides of the equation term by term we get

$$a_0 \sum_{n\geq 0} x_n z^n + a_1 \sum_{n\geq 0} x_{n+1} z^n + \cdots + a_k \sum_{n\geq 0} x_{n+k} z^n = \sum_{n\geq 0} f(n) z^n \ .$$

Then

$$a_0 \sum_{n\geq 0} x_n z^n + \frac{a_1}{z} \sum_{n\geq 0} x_{n+1} z^{n+1} + \cdots + \frac{a_k}{z^k} \sum_{n\geq 0} x_{n+k} z^{n+k} = \sum_{n\geq 0} f(n) z^n \ .$$

Let

$$X(z) = \sum_{n\geq 0} x_n z^n \quad \text{and} \quad F(z) = \sum_{n\geq 0} f(n) z^n \ .$$

The equation can be written as

$$a_0 X(z) + \frac{a_1}{z}\Big(X(z) - x_0\Big) + \cdots + \frac{a_k}{z^k}\Big(X(z) - x_0 - x_1 z - \cdots - x_{k-1} z^{k-1}\Big) = F(z) \ .$$

This can be solved for $X(z)$. If $X(z)$ is a rational fraction, then it can be decomposed into partial (elementary) fractions which, after expanding them into series, will give us the general solution x_n of the original recurrence equation. We can also try to use the expansion into series in the case when the function is not a rational fraction.

Example 11.11 Solve the following equation using the above method

$$x_{n+1} - 2x_n = 2^{n+1} - 2, \text{ if } n \geq 0 \quad \text{and } x_0 = 0 \ .$$

After multiplying and summing we have

$$\frac{1}{z} \sum_{n\geq 0} x_{n+1} z^{n+1} - 2 \sum_{n\geq 0} x_n z^n = 2 \sum_{n\geq 0} 2^n z^n - 2 \sum_{n\geq 0} z^n \ ,$$

and

$$\frac{1}{z}\ X(z) - x_0\ \ - 2X(z) = \frac{2}{1-2z} - \frac{2}{1-z} \ .$$

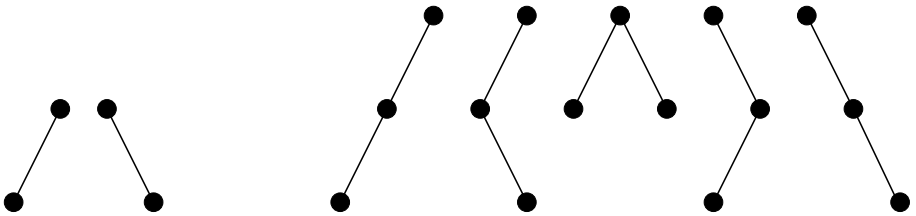


Figure 11.2. Binary trees with two and three vertices.

Since $x_0 = 0$, after decomposing the right-hand side into partial fractions[1]), the solution of the equation is

$$X(z) = \frac{2z}{(1-2z)^2} + \frac{2}{1-z} - \frac{2}{1-2z} \ .$$

After differentiating the generating function

$$\frac{1}{1-2z} = \sum_{n\geq 0} 2^n z^n$$

term by term we get

$$\frac{2}{(1-2z)^2} = \sum_{n\geq 1} n2^n z^{n-1} \ .$$

Thus

$$X(z) = \sum_{n\geq 0} n2^n z^n + 2\sum_{n\geq 0} z^n - 2\sum_{n\geq 0} 2^n z^n = \sum_{n\geq 0} \ (n-2)2^n + 2 \ \ z^n \ ,$$

therefore

$$x_n = (n-2)2^n + 2 \ .$$

The number of binary trees Let us denote by b_n the number of binary trees with n vertices. Then $b_1 = 1$, $b_2 = 2$, $b_3 = 5$ (see Figure 11.2). Let $b_0 = 1$. (We will see later that this is a good choice.)

In a binary tree with n vertices, there are altogether $n-1$ vertices in the left and right subtrees. If the left subtree has k vertices and the right subtree has $n-1-k$ vertices, then there exists $b_k b_{n-1-k}$ such binary trees. Summing over $k = 0, 1, \ldots, n-1$, we obtain exactly the number b_n of binary trees. Thus for any natural number $n \geq 1$ the recurrence equation in b_n is

$$b_n = b_0 b_{n-1} + b_1 b_{n-2} + \cdots + b_{n-1} b_0 \ . \tag{11.15}$$

This can also be written as

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} \ .$$

[1] For decomposing the fraction into partial fractions we can use the Method of Undetermined Coefficients.

Multiplying both sides by z^n, then summing over all n, we obtain

$$\sum_{n\geq 1} b_n z^n = \sum_{n\geq 1}\left(\sum_{k=0}^{n-1} b_k b_{n-1-k}\right) z^n \ . \tag{11.16}$$

Let $B(z) = \sum_{n\geq 0} b_n z^n$ be the generating function of the numbers b_n. The left-hand side of (11.16) is exactly $B(z)-1$ (because $b_0 = 1$). The right-hand side looks like a product of two generating functions. To see which functions are in consideration, let us use the notation

$$A(z) = zB(z) = \sum_{n\geq 0} b_n z^{n+1} = \sum_{n\geq 1} b_{n-1} z^n \ .$$

Then the right-hand side of (11.16) is exactly $A(z)B(z)$, which is $zB^2(z)$. Therefore

$$B(z) - 1 = zB^2(z), \quad B(0) = 1 \ .$$

Solving this equation for $B(z)$ gives

$$B(z) = \frac{1 \pm \sqrt{1-4z}}{2z} \ .$$

We have to choose the negative sign because $B(0) = 1$. Thus

$$\begin{aligned}
B(z) &= \frac{1}{2z}\left(1-\sqrt{1-4z}\right) = \frac{1}{2z}\left(1-(1-4z)^{1/2}\right) \\
&= \frac{1}{2z}\left(1 - \sum_{n\geq 0}\binom{1/2}{n}(-4z)^n\right) = \frac{1}{2z}\left(1 - \sum_{n\geq 0}\binom{1/2}{n}(-1)^n 2^{2n} z^n\right) \\
&= \frac{1}{2z} - \binom{1/2}{0}\frac{2^0 z^0}{2z} + \binom{1/2}{1}\frac{2^2 z}{2z} - \cdots - \binom{1/2}{n}(-1)^n \frac{2^{2n} z^n}{2z} + \cdots \\
&= \binom{1/2}{1}2 - \binom{1/2}{2}2^3 z + \cdots - \binom{1/2}{n}(-1)^n 2^{2n-1} z^{n-1} + \cdots \\
&= \sum_{n\geq 0}\binom{1/2}{n+1}(-1)^n 2^{2n+1} z^n = \sum_{n\geq 0}\frac{1}{n+1}\binom{2n}{n} z^n \ .
\end{aligned}$$

Therefore $b_n = \dfrac{1}{n+1}\binom{2n}{n}$. The numbers b_n are also called the Catalan numbers.

Remark. In the previous computation we used the following formula that can be proved easily

$$\binom{1/2}{n+1} = \frac{(-1)^n}{2^{2n+1}(n+1)}\binom{2n}{n} \ .$$

The number of leaves of all binary trees of n vertices Let us count the number of leaves (vertices with degree 1) in the set of all binary trees of n vertices. Denote this number by f_n. We remark that the root is not considered leaf even if it is of degree 1. It is easy to see that $f_2 = 2$, $f_3 = 6$. Let $f_0 = 0$ and $f_1 = 1$,

conventionally. Later we will see that this choice of the initial values is appropriate.

As in the case of numbering the binary trees, consider the binary trees of n vertices having k vertices in the left subtree and $n-k-1$ vertices in the right subtree. There are b_k such left subtrees and b_{n-1-k} right subtrees. If we consider such a left subtree and all such right subtrees, then together there are f_{n-1-k} leaves in the right subtrees. So for a given k there are $b_{n-1-k}f_k + b_k f_{n-1-k}$ leaves. After summing we have

$$f_n = \sum_{k=0}^{n-1} (f_k b_{n-1-k} + b_k f_{n-1-k}) \ .$$

By an easy computation we get

$$f_n = 2(f_0 b_{n-1} + f_1 b_{n-2} + \cdots + f_{n-1} b_0), \quad n \geq 2 \ . \tag{11.17}$$

This is a recurrence equation, with solution f_n. Let

$$F(z) = \sum_{n\geq 0} f_n z^n \quad \text{and} \quad B(z) = \sum_{n\geq 0} b_n z^n \ .$$

Multiplying both sides of (11.17) by z^n and summing gives

$$\sum_{n\geq 2} f_n z^n = 2 \sum_{n\geq 2} \left(\sum_{k=0}^{n-1} f_k b_{n-1-k} \right) z^n \ .$$

Since $f_0 = 0$ and $f_1 = 1$,

$$F(z) - z = 2zF(z)B(z) \ .$$

Thus

$$F(z) = \frac{z}{1-2zB(z)} \ ,$$

and since

$$B(z) = \frac{1}{2z} \left(1 - \sqrt{1-4z}\,\right) \ ,$$

we have

$$F(z) = \frac{z}{\sqrt{1-4z}} = z(1-4z)^{-1/2} = z \sum_{n\geq 0} \binom{-1/2}{n} (-4z)^n \ .$$

After the computations

$$F(z) = \sum_{n\geq 0} \binom{2n}{n} z^{n+1} = \sum_{n\geq 1} \binom{2n-2}{n-1} z^n \ ,$$

and

$$f_n = \binom{2n-2}{n-1} \quad \text{or} \quad f_{n+1} = \binom{2n}{n} = (n+1)b_n \ .$$

The number of binary trees with n vertices and k leaves A bit harder problem: how many binary trees are there with n vertices and k leaves? Let us denote this number by $b_n^{(k)}$. It is easy to see that $b_n^{(k)} = 0$, if $k > \lfloor (n+1)/2 \rfloor$. By a simple reasoning the case $k = 1$ can be solved. The result is $b_n^{(1)} = 2^{n-1}$ for any natural number $n \geq 1$. Let $b_0^{(0)} = 1$, conventionally. We will see later that this choice of the initial value is appropriate. Let us consider, as in case of the previous problems, the left and right subtrees. If the left subtree has i vertices and j leaves, then the right subtree has $n - i - 1$ vertices and $k - j$ leaves. The number of these trees is $b_i^{(j)} b_{n-i-1}^{(k-j)}$. Summing over k and j gives

$$b_n^{(k)} = 2b_{n-1}^{(k)} + \sum_{i=1}^{n-2} \sum_{j=1}^{k-1} b_i^{(j)} b_{n-i-1}^{(k-j)} \,. \tag{11.18}$$

For solving this recurrence equation the generating function

$$B^{(k)}(z) = \sum_{n \geq 0} b_n^{(k)} z^n, \qquad \text{where } k \geq 1$$

will be used. Multiplying both sides of equation (11.18) by z^n and summing over $n = 0, 1, 2, \ldots$, we get

$$\sum_{n \geq 1} b_n^{(k)} z^n = 2 \sum_{n \geq 1} b_{n-1}^{(k)} z^n + \sum_{n \geq 1} \left(\sum_{i=1}^{n-2} \sum_{j=1}^{k-1} b_i^{(j)} b_{n-i-1}^{(k-j)} \right) z^n \,.$$

Changing the order of summation gives

$$\sum_{n \geq 1} b_n^{(k)} z^n = 2 \sum_{n \geq 1} b_{n-1}^{(k)} z^n + \sum_{j=1}^{k-1} \sum_{n \geq 1} \left(\sum_{i=1}^{n-2} b_i^{(j)} b_{n-i-1}^{(k-j)} \right) z^n \,.$$

Thus

$$B^{(k)}(z) = 2zB^{(k)}(z) + z \left(\sum_{j=1}^{k-1} B^{(j)}(z) B^{(k-j)}(z) \right)$$

or

$$B^{(k)}(z) = \frac{z}{1-2z} \left(\sum_{j=1}^{k-1} B^{(j)}(z) B^{(k-j)}(z) \right) \,. \tag{11.19}$$

Step by step, we can write the following:

$$B^{(2)}(z) = \frac{z}{1-2z} \Big(B^{(1)}(z) \Big)^2 \,,$$

$$B^{(3)}(z) = \frac{2z^2}{(1-2z)^2} \Big(B^{(1)}(z) \Big)^3 \,,$$

$$B^{(4)}(z) = \frac{5z^3}{(1-2z)^3} \Big(B^{(1)}(z) \Big)^4 \,.$$

Let us try to find the solution in the form

$$B^{(k)}(z) = \frac{c_k z^{k-1}}{(1-2z)^{k-1}} \Big(B^{(1)}(z) \Big)^k \ ,$$

where $c_2 = 1$, $c_3 = 2$, $c_4 = 5$. Substituting in (11.19) gives a recursion for the numbers c_k

$$c_k = \sum_{i=1}^{k-1} c_i c_{k-i} \ .$$

We solve this equation using the generating function method. If $k = 2$, then $c_2 = c_1 c_1$, and so $c_1 = 1$. Let $c_0 = 1$. If $C(z) = \sum_{n \geq 0} c_n z^n$ is the generating function of the numbers c_n, then, using the formula of multiplication of the generating functions we obtain

$$C(z) - 1 - z = (C(z) - 1)^2 \quad \text{or} \quad C^2(z) - 3C(z) + z + 2 = 0 \ ,$$

thus

$$C(z) = \frac{3 - \sqrt{1-4z}}{2} \ .$$

Since $C(0) = 1$, only the negative sign can be chosen. After expanding the generating function we get

$$\begin{aligned} C(z) &= \frac{3}{2} - \frac{1}{2}(1-4z)^{1/2} = \frac{3}{2} - \frac{1}{2} \sum_{n \geq 0} \frac{-1}{2n-1} \binom{2n}{n} z^n \\ &= \frac{3}{2} + \sum_{n \geq 0} \frac{1}{2(2n-1)} \binom{2n}{n} z^n = 1 + \sum_{n \geq 1} \frac{1}{2(2n-1)} \binom{2n}{n} z^n \ . \end{aligned}$$

From this

$$c_n = \frac{1}{2(2n-1)} \binom{2n}{n}, \quad n \geq 1 \ .$$

Since $b_n^{(1)} = 2^{n-1}$ for $n \geq 1$, it can be proved easily that $B^{(1)} = z/(1-2z)$. Thus

$$B^{(k)}(z) = \frac{1}{2(2k-1)} \binom{2k}{k} \frac{z^{2k-1}}{(1-2z)^{2k-1}} \ .$$

Using the formula

$$\frac{1}{(1-z)^m} = \sum_{n \geq 0} \binom{n+m-1}{n} z^n \ ,$$

therefore

$$\begin{aligned} B^{(k)}(z) &= \frac{1}{2(2k-1)} \binom{2k}{k} \sum_{n \geq 0} \binom{2k+n-2}{n} 2^n z^{2k+n-1} \\ &= \frac{1}{2(2k-1)} \binom{2k}{k} \sum_{n \geq 2k-1} \binom{n-1}{n-2k+1} 2^{n-2k+1} z^n \ . \end{aligned}$$

Thus

$$b_n^{(k)} = \frac{1}{2k-1}\binom{2k}{k}\binom{n-1}{2k-2}2^{n-2k}$$

or

$$b_n^{(k)} = \frac{1}{n}\binom{2k}{k}\binom{n}{2k-1}2^{n-2k} .$$

11.2.3. The Z-transform method

When solving linear nonhomogeneous equations using generating functions, the solution is usually done by the expansion of a rational fraction. The Z-transform method can help us in expanding such a function. Let $P(z)/Q(z)$ be a rational fraction, where the degree of $P(z)$ is less, than the degree of $Q(z)$. If the roots of the denominator are known, the rational fraction can be expanded into partial fractions using the Method of Undetermined Coefficients.

Let us first consider the case when the denominator has distinct roots $\alpha_1, \alpha_2, \ldots, \alpha_k$. Then

$$\frac{P(z)}{Q(z)} = \frac{A_1}{z-\alpha_1} + \cdots + \frac{A_i}{z-\alpha_i} + \cdots + \frac{A_k}{z-\alpha_k} .$$

It is easy to see that

$$A_i = \lim_{z\to\alpha_i}(z-\alpha_i)\frac{P(z)}{Q(z)}, \qquad i = 1, 2, \ldots, k .$$

But

$$\frac{A_i}{z-\alpha_i} = \frac{A_i}{-\alpha_i\left(1-\dfrac{1}{\alpha_i}z\right)} = \frac{-A_i\beta_i}{1-\beta_i z} ,$$

where $\beta_i = 1/\alpha_i$. Now, by expanding this partial fraction, we get

$$\frac{-A_i\beta_i}{1-\beta_i z} = -A_i\beta_i\left(1+\beta_i z+\cdots+\beta_i^n z^n+\cdots\right).$$

Denote the coefficient of z^n by $C_i(n)$, then $C_i(n) = -A_i\beta_i^{n+1}$, so

$$C_i(n) = -A_i\beta_i^{n+1} = -\beta_i^{n+1}\lim_{z\to\alpha_i}(z-\alpha_i)\frac{P(z)}{Q(z)} ,$$

or

$$C_i(n) = -\beta_i^{n+1}\lim_{z\to\alpha_i}\frac{(z-\alpha_i)P(z)}{Q(z)} .$$

After the transformation $z \to 1/z$ and using $\beta_i = 1/\alpha_i$ we obtain

$$C_i(n) = \lim_{z\to\beta_i}\left((z-\beta_i)z^{n-1}\frac{p(z)}{q(z)}\right) ,$$

where

$$\frac{p(z)}{q(z)} = \frac{P(1/z)}{Q(1/z)} .$$

Thus in the expansion of $X(z) = \dfrac{P(z)}{Q(z)}$ the coefficient of z^n is

$$C_1(n) + C_2(n) + \cdots + C_k(n) \ .$$

If α is a root of the polynomial $Q(z)$, then $\beta = 1/\alpha$ is a root of $q(z)$. E.g. if

$$\frac{P(z)}{Q(z)} = \frac{2z^z}{(1-z)(1-2z)} \ , \quad \text{then} \quad \frac{p(z)}{q(z)} = \frac{2}{(z-1)(z-2)} \ .$$

If case of multiple roots, e.g. if β_i has multiplicity p, their contribution to the solution is

$$C_i(n) = \frac{1}{(p-1)!} \lim_{z \to \beta_i} \frac{d^{p-1}}{dz^{p-1}} \left((z-\beta_i)^p z^{n-1} \frac{p(z)}{q(z)} \right) \ .$$

Here $\dfrac{d^p}{dz^p} f(z)$ is the derivative of order p of the function $f(z)$.

All these can be summarised in the following algorithm. Suppose that the coefficients of the equation are in array A, and the constants of the solution are in array C.

LINEAR-NONHOMOGENEOUS(A, k, f)

1 let $a_0 x_n + a_1 x_{n+1} + \cdots + a_k x_{n+k} = f(n)$ be the equation, where $f(n)$ is a rational fraction; multiply both sides by z^n, and sum over all n
2 transform the equation into the form $X(z) = P(z)/Q(z)$, where $X(z) = \sum_{n \geq 0} x_n z^n$, $P(z)$ and $Q(z)$ are polynomials
3 use the transformation $z \to 1/z$, and let the result be $p(z)/q(z)$, where $p(z)$ are $q(z)$ are polynomials
4 denote the roots of $q(z)$ by
  β_1, with multiplicity p_1, $p_1 \geq 1$,
  β_2, with multiplicity p_2, $p_2 \geq 1$,
  ...
  β_k, with multiplicity p_k, $p_k \geq 1$;
  then the general solution of the original equation is
  $x_n = C_1(n) + C_2(n) + \cdots + C_k(n)$, where
  $C_i(n) = 1/((p_i - 1)!) \lim_{z \to \beta_i} \frac{d^{p_i - 1}}{dz^{p_i - 1}} \left((z - \beta_i)^{p_i} z^{n-1} (p(z)/q(z)) \right), i = 1, 2, \ldots, k.$
5 **return** C

If we substitute z by $1/z$ in the generating function, the result is the so-called Z-transform, for which similar operations can be defined as for the generating functions. The residue theorem for the Z-transform gives the same result. The name of the method is derived from this observation.

Example 11.12 Solve the recurrence equation

$$x_{n+1} - 2x_n = 2^{n+1} - 2, \quad \text{if} \quad n \geq 0, \quad x_0 = 0 \ .$$

Multiplying both sides by z^n and summing we obtain

$$\sum_{n \geq 0} x_{n+1} z^n - 2 \sum_{n \geq 0} x_n z^n = \sum_{n \geq 0} 2^{n+1} z^n - \sum_{n \geq 0} 2 z^n \ ,$$

or

$$\frac{1}{z}X(z) - 2X(z) = \frac{2}{1-2z} - \frac{2}{1-z}, \quad \text{where } X(z) = \sum_{n\geq 0} x_n z^n \ .$$

Thus

$$X(z) = \frac{2z^2}{(1-z)(1-2z)^2} \ .$$

After the transformation $z \to 1/z$ we get

$$\frac{p(z)}{q(z)} = \frac{2z}{(z-1)(z-2)^2} \ ,$$

where the roots of the denominator are 1 with multiplicity 1 and 2 with multiplicity 2. Thus

$$C_1 = \lim_{z\to 1} \frac{2z^n}{(z-2)^2} = 2 \qquad \text{and}$$

$$C_2 = \lim_{z\to 2} \frac{d}{dz} \quad \frac{2z^n}{z-1} \quad = 2 \lim_{z\to 2} \frac{nz^{n-1}(z-1) - z^n}{(z-1)^2} = 2^n(n-2) \ .$$

Therefore the general solution is

$$x_n = 2^n(n-2) + 2, \quad n \geq 0 \ .$$

Example 11.13 Solve the recurrence equation

$$x_{n+2} = 2x_{n+1} - 2x_n, \quad \text{if} \quad n \geq 0, \quad x_0 = 0, \quad x_1 = 1 \ .$$

Multiplying by z^n and summing gives

$$\frac{1}{z^2} \sum_{n\geq 0} x_{n+2} z^{n+2} = \frac{2}{z} \sum_{n\geq 0} x_{n+1} z^{n+1} - 2 \sum_{n\geq 0} x_n z^n \ ,$$

so

$$\frac{1}{z^2} \quad F(z) - z \quad = \frac{2}{z} F(z) - 2F(z) \ ,$$

that is

$$F(z) \quad \frac{1}{z^2} - \frac{2}{z} + 2 \quad = -\frac{1}{z} \ .$$

Then

$$F(1/z) = \frac{-z}{z^2 - 2z + 2} \ .$$

The roots of the denominator are $1+i$ and $1-i$. Let us compute $C_1(n)$ and $C_2(n)$:

$$C_1(n) = \lim_{z\to 1+i} \frac{-z^{n+1}}{z-(1-i)} = \frac{i(1+i)^n}{2} \quad \text{and}$$

$$C_2(n) = \lim_{z\to 1-i} \frac{-z^{n+1}}{z-(1+i)} = \frac{-i(1-i)^n}{2} \ .$$

Since

$$1+i = \sqrt{2} \quad \cos\frac{\pi}{4} + i \sin\frac{\pi}{4} \quad , \quad 1-i = \sqrt{2} \quad \cos\frac{\pi}{4} - i\sin\frac{\pi}{4} \quad ,$$

raising to the nth power gives

$$(1+i)^n = \quad \sqrt{2} \ ^n \quad \cos\frac{n\pi}{4} + i\sin\frac{n\pi}{4} \quad , \quad (1-i)^n = \quad \sqrt{2} \ ^n \quad \cos\frac{n\pi}{4} - i\sin\frac{n\pi}{4} \quad ,$$

$$x_n = C_1(n) + C_2(n) = \quad \sqrt{2} \ ^n \sin\frac{n\pi}{4} \ .$$

Exercises

11.2-1 How many binary trees are there with n vertices and no empty left and right subtrees?

11.2-2 How many binary trees are there with n vertices, in which each vertex which is not a leaf, has exactly two descendants?

11.2-3 Solve the following recurrent equation using generating functions.

$$H_n = 2H_{n-1} + 1, \quad H_0 = 0 \ .$$

(H_n is the number of moves in the problem of the Towers of Hanoi.)

11.2-4 Solve the following recurrent equation using the Z-transform method.

$$F_{n+2} = F_{n+1} + F_n + 1, \text{ if } n \geq 0, \quad \text{and} \quad F_0 = 0, F_1 = 1 \ .$$

11.2-5 Solve the following system of recurrence equations:

$$\begin{aligned} u_n &= v_{n-1} + u_{n-2} \ , \\ v_n &= u_n + u_{n-1} \ , \end{aligned}$$

where $u_0 = 1, u_1 = 2, v_0 = 1$.

11.3. Numerical solution

Using the following function we can solve the linear recurrent equations numerically. The equation is given in the form

$$a_0 x_n + a_1 x_{n+1} + \cdots + a_k x_{n+k} = f(n) \ ,$$

where $a_0, a_k \neq 0, k \geq 1$. The coefficients $a_0, a_1, \ldots, a_k$ are kept in array A, the initial values $x_0, x_1, \ldots, x_{k-1}$ in array X. To find x_n we will compute step by step the values $x_k, x_{k+1}, \ldots, x_n$, keeping in the previous k values of the sequence in the first k positions of X (i.e. in the positions with indices $0, 1, \ldots, k-1$).

RECURRENCE(A, X, k, n, f)

```
 1 for j ← k to n
 2     do v ← A[0] · X[0]
 3        for i ← 1 to k − 1
 4            do v ← v + A[i] · X[i]
 5        v ← (f(j − k) − v)/A[k]
 6        if j ≠ n
 7           then for i ← 0 to k − 2
 8                    do X[i] ← X[i + 1]
 9                X[k − 1] ← v
10 return v
```

Lines 2–5 compute the values x_j $(j = k, k+1, \ldots, n)$ (using the previous k values), denoted by v in the algorithm. In lines 7–9, if n is not yet reached, we copy the last k values in the first k positions of X. In line 10 x_n is obtained. It is easy to see that the computation time is $\Theta(kn)$, if we disregard the time to compute the values of the function.

Exercises

11.3-1 How many additions, subtractions, multiplications and divisions are required using the algorithm RECURRENCE, while it computes x_{1000} using the data given in Example 11.4?

Problems

11-1 Existence of a solution of a homogeneous equation using generating function

Prove that a linear homogeneous equation cannot be solved using generating functions (because $X(z) = 0$ is obtained) if and only if $x_n = 0$ for all n.

11-2 Complex roots in case of Z-transform

What happens if the roots of the denominator are complex when applying the Z-transform method? The solution of the recurrence equation must be real. Does the method ensure this?

Chapter notes

Recurrence equations are discussed in detail by Elaydi [65], Flajolet and Sedgewick [216], Greene and Knuth [93], Mickens [166].

Knuth [134] and Graham, Knuth and Patashnik [92] deal with generating functions. In the book of Vilenkin [248] there are a lot of simple and interesting problems about recurrences and generating functions.

In [155] Lovász also presents problems on generating functions.

Counting the binary trees is from Knuth [134], counting the leaves in the set of all binary trees and counting the binary trees with n vertices and k leaves are from Z. Kása [141].

12. Scientific computing

This title refers to a fast developing interdisciplinary area between mathematics, computers and applications. The subject is also often called as Computational Science and Engineering. Its aim is the efficient use of computer algorithms to solve engineering and scientific problems. One can say with a certain simplification that our subject is related to numerical mathematics, software engineering, computer graphics and applications. Here we can deal only with some basic elements of the subject such as the fundamentals of the *floating point computer arithmetic*, *error analysis*, the basic numerical methods of *linear algebra* and related mathematical software.

12.1. Floating point arithmetic and error analysis

12.1.1. Classical error analysis

Let x be the exact value and let a be an approximation of x ($a \approx x$). The error of the approximation a is defined by the formula $\Delta a = x - a$ (sometimes with opposite sign). The quantity $\delta a \geq 0$ is called an ***(absolute) error (bound)*** of approximation a, if $|x - a| = |\Delta a| \leq \delta a$. For example, the error of the approximation $\sqrt{2} \approx 1.41$ is at most 0.01. In other words, the error bound of the approximation is 0.01. The quantities x and a (and accordingly Δa and δa) may be vectors or matrices. In such cases the absolute value and relation operators must be understood componentwise. We also measure the error by using matrix and vector norms. In such cases, the quantity $\delta a \in \mathbb{R}$ is an error bound, if the inequality $\|\Delta a\| \leq \delta a$ holds.

The absolute error bound can be irrelevant in many cases. For example, an approximation with error bound 0.05 has no value in estimating a quantity of order 0.001. The goodness of an approximation is measured by the ***relative error*** $\delta a / |x|$ ($\delta a / \|x\|$ for vectors and matrices), which compares the error bound to the approximated quantity. Since the exact value is generally unknown, we use the approximate relative error $\delta a / |a|$ ($\delta a / \|a\|$). The committed error is proportional to the quantity $(\delta a)^2$, which can be neglected, if the absolute value (norm) of x and a is much greater than $(\delta a)^2$. The relative error is often expressed in percentages.

In practice, the (absolute) error bound is used as a substitute for the generally unknown true error.

In the *classical error analysis* we assume input data with given error bounds, exact computations (operations) and seek for the error bound of the final result. Let x and y be exact values with approximations a and b, respectively. Assume that the absolute error bounds of approximations a and b are δa and δb, respectively. Using the classical error analysis approach we obtain the following error bounds for the four basic arithmetic operations:

$$\begin{aligned}
\delta(a+b) &= \delta a + \delta b, \qquad \frac{\delta(a+b)}{|a+b|} = \max\left\{\frac{\delta a}{|a|}, \frac{\delta b}{|b|}\right\} \quad (ab > 0) \ , \\
\delta(a-b) &= \delta a + \delta b, \qquad \frac{\delta(a-b)}{|a-b|} = \frac{\delta a + \delta b}{|a-b|} \quad (ab > 0) \ , \\
\delta(ab) &\approx |a|\,\delta b + |b|\,\delta a \qquad \frac{\delta(ab)}{|ab|} \approx \frac{\delta a}{|a|} + \frac{\delta b}{|b|} \quad (ab \neq 0) \ , \\
\delta(a/b) &\approx \frac{|a|\,\delta b + |b|\,\delta a}{|b|^2} \qquad \frac{\delta(a/b)}{|a/b|} \approx \frac{\delta a}{|a|} + \frac{\delta b}{|b|} \quad (ab \neq 0) \ .
\end{aligned}$$

We can see that the division with a number near to 0 can make the absolute error arbitrarily big. Similarly, if the result of subtraction is near to 0, then its relative error can become arbitrarily big. One has to avoid these cases. Especially the subtraction operation can be quite dangerous.

Example 12.1 Calculate the quantity $\sqrt{1996}-\sqrt{1995}$ with approximations $\sqrt{1996} \approx 44.67$ and $\sqrt{1995} \approx 44.66$ whose common absolute and relative error bounds are 0.01 and 0.022%, respectively. One obtains the approximate value $\sqrt{1996}-\sqrt{1995} \approx 0.01$, whose relative error bound is

$$\frac{0.01 + 0.01}{0.01} = 2 \ ,$$

that is 200%. The true relative error is about 10.66%. Yet it is too big, since it is approximately 5×10^2 times bigger than the relative error of the initial data. We can avoid the subtraction operation by using the following trick

$$\sqrt{1996} - \sqrt{1995} = \frac{1996 - 1995}{\sqrt{1996} + \sqrt{1995}} = \frac{1}{\sqrt{1996} + \sqrt{1995}} \approx \frac{1}{89.33} \approx 0.01119 \ .$$

Here the nominator is exact, while the absolute error of the denominator is 0.02. Hence the relative error (bound) of the quotient is about $0.02/89.33 \approx 0.00022 = 0.022\%$. The latter result is in agreement with the relative error of the initial data and it is substantially smaller than the one obtained with direct subtraction operation.

The first order error terms of twice differentiable functions can be obtained by their first order *Taylor polynomial*:

$$\begin{aligned}
\delta(f(a)) &\approx |f'(a)|\,\delta a, \quad f : \mathbb{R} \to \mathbb{R} \ , \\
\delta(f(a)) &\approx \sum_{i=1}^{n} \left|\frac{\partial f(a)}{\partial x_i}\right| \delta a_i, \quad f : \mathbb{R}^n \to \mathbb{R} \ .
\end{aligned}$$

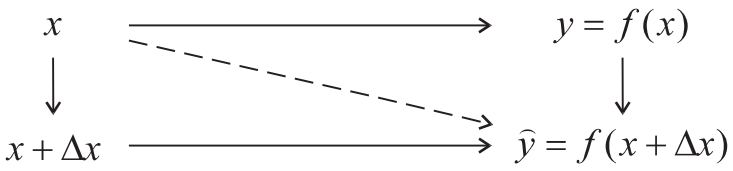


Figure 12.1. Forward and backward error.

The numerical sensitivity of functions at a given point is characterised by the ***condition number***, which is the ratio of the relative errors of approximate function value and the input data (the *Jacobian matrix* of functions $F : \mathbb{R}^n \to \mathbb{R}^m$ is denoted by $F'(a)$ at the point $a \in \mathbb{R}^n$):

$$c(f,a) = \frac{|f'(a)|\,|a|}{|f(a)|}, \quad f : \mathbb{R} \to \mathbb{R}\ ,$$
$$c(F,a) = \frac{\|a\|\,\|F'(a)\|}{\|F(a)\|}, \quad F : \mathbb{R}^n \to \mathbb{R}^m\ .$$

We can consider the condition number as the magnification number of the input relative error. Therefore the functions is considered ***numerically stable*** (or ***well-conditioned***) at the point a, if $c(f,a)$ is „small". Otherwise f is considered as ***numerically unstable*** (***ill-conditioned***). The condition number depends on the point a. A function can be well-conditioned at point a, while it is ill-conditioned at point b. The term „small" is relative. It depends on the problem, the computer and the required precision.

The condition number of matrices can be defined as the upper bound of a function condition number. Let us define the mapping $F : \mathbb{R}^n \to \mathbb{R}^n$ by the solution of the equation $Ay = x$ ($A \in \mathbb{R}^{n\times n}$, $\det(A) \neq 0$), that is, let $F(x) = A^{-1}x$. Then $F' \equiv A^{-1}$ and

$$c(F,a) = \frac{\|a\|\,\left\|A^{-1}\right\|}{\|A^{-1}a\|} = \frac{\|Ay\|\,\left\|A^{-1}\right\|}{\|y\|} \leq \|A\|\,\left\|A^{-1}\right\| \quad (Ay = a)\ .$$

The upper bound of the right side is called the ***condition number of the matrix*** A. This bound is sharp, since there exists a vector $a \in \mathbb{R}^n$ such that $c(F,a) = \|A\|\,\left\|A^{-1}\right\|$.

12.1.2. Forward and backward errors

Let us investigate the calculation of the function value $f(x)$. If we calculate the approximation $\hat{y}$ instead of the exact value $y = f(x)$, then the forward error $\Delta y = \hat{y} - y$. If for a value $x + \Delta x$ the equality $\hat{y} = f(x + \Delta x)$ holds, that is, $\hat{y}$ is the exact function value of the perturbed input data $\hat{x} = x + \Delta x$, then Δx is called the ***backward error***. The connection of the two concepts is shown on the Figure 12.1.

The continuous line shows exact value, while the dashed one indicates computed value. The analysis of the backward error is called the ***backward error analysis*** . If there exist more than one backward error, then the estimation of the smallest one

is the most important.

An algorithm for computing the value $y = f(x)$ is called ***backward stable***, if for any x it gives a computed value $\hat{y}$ with small backward error Δx. Again, the term „small" is relative to the problem environment.

The connection of the forward and backward errors is described by the approximate thumb rule

$$\frac{\delta \hat{y}}{|y|} \lessapprox c(f,x) \frac{\delta \hat{x}}{|x|} , \tag{12.1}$$

which means that

$$\text{relative forward error} \leq \text{condition number} \times \text{relative backward error.}$$

This inequality indicates that the computed solution of an ill-conditioned problem may have a big relative forward error. An algorithm is said to be ***forward stable*** if the forward error is small. A forward stable method is not necessarily backward stable. If the forward error and the condition number are small, then the algorithm is forward stable.

Example 12.2 Consider the function $f(x) = \log x$ the condition number of which is $c(f,x) = c(x) = 1/|\log x|$. For $x \approx 1$ the condition number $c(f,x)$ is big. Therefore the relative forward error is big for $x \approx 1$.

12.1.3. Rounding errors and floating point arithmetic

The classical error analysis investigates only the effects of the input data errors and assumes exact arithmetic operations. The digital computers however are representing the numbers with a finite number of digits, the arithmetic computations are carried out on the elements of a finite set F of such numbers and the results of operations belong to F. Hence the computer representation of the numbers may add further errors to the input data and the results of arithmetic operations may also be subject to further rounding. If the result of operation belongs to F, then we have the exact result. Otherwise we have three cases:

(i) rounding to representable (nonzero) number;

(ii) underflow (rounding to 0);

(iii) overflow (in case of results whose moduli too large).

The most of the scientific-engineering calculations are done in *floating point arithmetic* whose generally accepted model is the following:

Definition 12.1 *The set of floating point numbers is given by*

$$F(\beta,t,L,U) =$$
$$= \left\{ \pm m \times \beta^e \mid \tfrac{1}{\beta} \leq m < 1,\ m = 0.d_1 d_2 \dots d_t,\ L \leq e \leq U \right\} \cup \{0\} ,$$

where

- β is the base (or radix) of the number system,

- m is the mantissa in the number system with base β,

- *e is the exponent,*
- *t is the length of mantissa (the precision of arithmetic),*
- *L is the smallest exponent (underflow exponent),*
- *U is the biggest exponent (overflow exponent).*

The parameters of the three most often used number systems are indicated in the following table

Name	β	Machines
binary	2	most computer
decimal	10	most calculators
hexadecimal	16	IBM mainframe computers

The mantissa can be written in the form

$$m = 0.d_1 d_2 \dots d_t = \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t}. \tag{12.2}$$

We can observe that condition $1/\beta \le m < 1$ implies the inequality $1 \le d_1 \le \beta - 1$ for the first digit d_1. The remaining digits must satisfy $0 \le d_i \le \beta - 1$ $(i = 2, \dots, t)$. Such arithmetic systems are called ***normalized***. The zero digit and the dot is not represented. If $\beta = 2$, then the first digit is 1, which is also unrepresented. Using the representation (12.2) we can give the set $F = F(\beta, t, L, U)$ in the form

$$F = \left\{ \pm \left(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \right) \beta^e \mid \ L \le e \le U \right\} \cup \{0\} \ , \tag{12.3}$$

where $0 \le d_i \le \beta - 1$ $(i = 1, \dots, t)$ and $1 \le d_1$.

Example 12.3 The set $F\,(2, 3, -1, 2)$ contains 33 elements and its positive elements are given by

$$\frac{1}{4}, \frac{5}{16}, \frac{6}{16}, \frac{7}{16}, \frac{1}{2}, \frac{5}{8}, \frac{6}{8}, \frac{7}{8}, 1, \frac{10}{8}, \frac{12}{8}, \frac{14}{8}, 2, \frac{20}{8}, 3, \frac{28}{8} \quad .$$

The elements of F are not equally distributed on the real line. The distance of two consecutive numbers in $[1/\beta, 1] \cap F$ is β^{-t}. Since the elements of F are of the form $\pm m \times \beta^e$, the distance of two consecutive numbers in F is changing with the exponent. The maximum distance of two consecutive floating point numbers is β^{U-t}, while the minimum distance is β^{L-t}.

For the mantissa we have $m \in [1/\beta, 1 - 1/\beta^t]$, since

$$\frac{1}{\beta} \le m = \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_t}{\beta^t} \le \frac{\beta - 1}{\beta} + \frac{\beta - 1}{\beta^2} + \cdots + \frac{\beta - 1}{\beta^t} = 1 - \frac{1}{\beta^t}.$$

Using this observation we can easily prove the following result on the range of floating point numbers.

Theorem 12.2 *If $a \in F$, $a \neq 0$, then $M_L \le |a| \le M_U$, where*

$$M_L = \beta^{L-1}, \quad M_U = \beta^U (1 - \beta^{-t}).$$

Let $a, b \in F$ and denote $\square$ any of the four arithmetic operations $(+, -, *, /)$. The following cases are possible:

(1) $a \square b \in F$ (exact result),

(2) $|a \square b| > M_U$ (arithmetic overflow),

(3) $0 < |a \square b| < M_L$ (arithmetic underflow),

(4) $a \square b \notin F$, $M_L < |a \square b| < M_U$ (not representable result).

In the last two cases the *floating point arithmetic* is rounding the result $a \square b$ to the nearest floating point number in F. If two consecutive floating point numbers are equally distant from $a \square b$, then we generally round to the greater number. For example, in a five digit decimal arithmetic, the number 2.6457513 is rounded to the number 2.6458.

Let $G = [-M_U, M_U]$. It is clear that $F \subset G$. Let $x \in G$. The $fl\,(x)$ denotes an element of F nearest to x. The mapping $x \to fl\,(x)$ is called rounding. The quantity $|x - fl\,(x)|$ is called the rounding error. If $fl\,(x) = 1$, then the rounding error is at most $\beta^{1-t}/2$. The quantity $u = \beta^{1-t}/2$ is called the ***unit roundoff***. The quantity u is the relative error bound of $fl\,(x)$.

Theorem 12.3 *If $x \in G$, then*

$$fl(x) = x(1 + \varepsilon), \quad |\varepsilon| \leq u\ .$$

Proof Without loss of generality we can assume that $x > 0$. Let $m_1\beta^e$, $m_2\beta^e \in F$ be two consecutive numbers such that

$$m_1\beta^e \leq x \leq m_2\beta^e\ .$$

Either $1/\beta \leq m_1 < m_2 \leq 1 - \beta^{-t}$ or $1 - \beta^{-t} = m_1 < m_2 = 1$ holds. Since $m_2 - m_1 = \beta^{-t}$ holds in both cases, we have

$$|fl\,(x) - x| \leq \frac{|m_2 - m_1|}{2}\beta^e = \frac{\beta^{e-t}}{2}$$

either $fl\,(x) = m_1\beta^e$ or $fl\,(x) = m_2\beta^e$. It follows that

$$\frac{|fl\,(x) - x|}{|x|} \leq \frac{|fl\,(x) - x|}{m_1\beta^e} \leq \frac{\beta^{e-t}}{2m_1\beta^e} = \frac{\beta^{-t}}{2m_1} \leq \frac{1}{2}\beta^{1-t} = u\ .$$

Hence $fl\,(x) - x = \lambda x u$, where $|\lambda| \leq 1$. A simple arrangement yields

$$fl\,(x) = x(1 + \varepsilon) \quad (\varepsilon = \lambda u)$$

Since $|\varepsilon| \leq u$, we proved the claim. ∎

Thus we proved that the relative error of the rounding is bounded in floating point arithmetic and the bound is the unit roundoff u.

Another quantity used to measure the rounding errors is the so called the ***machine epsilon*** $\epsilon_M = 2u = \beta^{1-t}$ $(\epsilon_M = 2u)$. The number ϵ_M is the distance of 1 and its nearest neighbour greater than 1. The following algorithm determines ϵ_M in the case of binary base.

MACHINE-EPSILON

```
1  x ← 1
2  while 1 + x > 1
3        do x ← x/2
4  ϵ_M ← 2x
5  return ϵ_M
```

In the MATLAB system $\epsilon_M \approx 2.2204 \times 10^{-16}$.

For the results of floating point arithmetic operations we assume the following (standard model):

$$fl(a \square b) = (a \square b)(1+\varepsilon), \quad |\varepsilon| \le u \quad (a, b \in F) \ . \tag{12.4}$$

The IEEE arithmetic standard satisfies this assumption. It is an important consequence of the assumption that for $a \square b \neq 0$ the relative error of arithmetic operations satisfies

$$\frac{|fl(a \square b) - (a \square b)|}{|a \square b|} \le u.$$

Hence the relative error of the floating point arithmetic operations is small.

There exist computer floating point arithmetics that do not comply with the standard model (12.4). The usual reason for this is that the arithmetic lacks a guard digit in subtraction. For simplicity we investigate the subtraction $1 - 0.111$ in a three digit binary arithmetic. In the first step we equate the exponents:

$$\begin{array}{rcccccccc} & 2 & \times & 0 & . & 1 & 0 & 0 & \\ - & 2 & \times & 0 & . & 0 & 1 & 1 & 1 \end{array} .$$

If the computation is done with four digits, the result is the following

$$\begin{array}{rcccccccc} & 2^1 & \times & 0 & . & 1 & 0 & 0 & \\ - & 2^1 & \times & 0 & . & 0 & 1 & 1 & 1 \\ \hline & 2^1 & \times & 0 & . & 0 & 0 & 0 & 1 \end{array} ,$$

from which the normalized result is $2^{-2} \times 0.100$. Observe that the subtracted number is unnormalised. The temporary fourth digit of the mantissa is called a guard digit. Without a guard digit the computations are the following:

$$\begin{array}{rcccccccc} & 2^1 & \times & 0 & . & 1 & 0 & 0 & \\ - & 2^1 & \times & 0 & . & 0 & 1 & 1 & \\ \hline & 2^1 & \times & 0 & . & 0 & 0 & 1 & \end{array} .$$

Hence the normalized result is $2^{-1} \times 0.100$ with a relative error of 100%. Several CRAY computers and pocket calculators lack guard digits.

Without the guard digit the floating point arithmetic operations satisfy only the weaker conditions

$$fl(x \pm y) = x(1+\alpha) \pm y(1+\beta), \quad |\alpha|, |\beta| \le u, \tag{12.5}$$

$$fl(x \square y) = (x \square y)(1+\delta), \quad |\delta| \le u, \quad \square = *, / \ . \tag{12.6}$$

Assume that we have a guard digit and the arithmetic complies with standard model (12.4). Introduce the following notations:

$$|z| = [|z_1|, \ldots, |z_n|]^T \quad (z \in \mathbb{R}^n) \ , \tag{12.7}$$

$$|A| = [|a_{ij}|]_{i,j=1}^{m,n} \quad \left(A \in \mathbb{R}^{m \times n}\right), \tag{12.8}$$

$$A \leq B \Leftrightarrow a_{ij} \leq b_{ij} \quad \left(A, B \in \mathbb{R}^{m \times n}\right) \ . \tag{12.9}$$

The following results hold:

$$\left| fl\left(x^T y\right) - x^T y \right| \leq 1.01nu \left|x\right|^T |y| \quad (nu \leq 0.01) \ , \tag{12.10}$$

$$fl\left(\alpha A\right) = \alpha A + E \quad (|E| \leq u \left|\alpha A\right|) \ , \tag{12.11}$$

$$fl\left(A + B\right) = (A + B) + E \quad (|E| \leq u \left|A + B\right|) \ , \tag{12.12}$$

$$fl\left(AB\right) = AB + E \quad \left(|E| \leq nu \left|A\right| |B| + O\left(u^2\right)\right) \ , \tag{12.13}$$

where E denotes the error (matrix) of the actual operation.

The standard *floating point arithmetics* have many special properties. It is an important property that the addition is not associative because of the rounding.

Example 12.4 If $a = 1$, $b = c = 3 \times 10^{-16}$, then using MATLAB and AT386 type PC we obtain

$$1.000000000000000e + 000 = (a + b) + c \neq a + (b + c) = 1.000000000000001e + 000 \ .$$

We can have a similar result on Pentium1 machine with the choice $b = c = 1.15 \times 10^{-16}$.

The example also indicates that for different (numerical) processors may produce different computational results for the same calculations. The commutativity can also be lost in addition. Consider the computation of the sum $\sum_{i=1}^{n} x_i$. The usual algorithm is the recursive summation.

RECURSIVE-SUMMATION(n, x)

```
1  s ← 0
2  for i ← 1 to n
3       do s ← s + x_i
4  return s
```

Example 12.5 Compute the sum

$$s_n = 1 + \sum_{i=1}^{n} \frac{1}{i^2 + i}$$

for $n = 4999$. The recursive summation algorithm (and MATLAB) gives the result

$$1.999800000000002e + 000 \ .$$

If the summation is done in the reverse (increasing) order, then the result is

$$1.999800000000000e + 000 \ .$$

If the two values are compared with the exact result $s_n = 2 - 1/(n+1)$, then we can see that the second summation gives better result. In this case the sum of smaller numbers gives significant digits to the final result unlike in the first case.

The last example indicates that the summation of a large number of data varying in modulus and sign is a complicated task. The following algorithm of W. Kahan is one of the most interesting procedures to solve the problem.

COMPENSATED-SUMMATION(n, x)

```
1  s ← 0
2  e ← 0
3  for  i ← 1 to n
4       do t ← s
5          y ← x_i + e
6          s ← t + y
7          e ← (t − s) + y
8  return s
```

12.1.4. The floating point arithmetic standard

The ANSI/IEEE Standard 754-1985 of a binary ($\beta = 2$) *floating point arithmetic system* was published in 1985. The standard specifies the basic arithmetic operations, comparisons, rounding modes, the arithmetic exceptions and their handling, and conversion between the different arithmetic formats. The square root is included as a basic operation. The standard does not deal with the exponential and transcendent functions. The standard defines two main floating point formats:

Type	Size	Mantissa	e	u	$[M_L, M_U] \approx$
Single	32 bits	$23+1$ bits	8 bits	$2^{-24} \approx 5.96 \times 10^{-8}$	$10^{\pm 38}$
Double	64 bits	$52+1$ bits	11 bits	$2^{-53} \approx 1.11 \times 10^{-16}$	$10^{\pm 308}$

In both formats one bit is reserved as a sign bit. Since the floating point numbers are normalized and the first digit is always 1, this bit is not stored. This hidden bit is denoted by the „+1" in the table.

The arithmetic standard contains the handling of arithmetic exceptions.

Exception type	Example	Default result
Invalid operation	$0/0$, $0 \times \infty$, $\sqrt{-1}$	NaN (Not a Number)
Overflow	$\|x \quad y\| > M_U$	$\pm\infty$
Divide by zero	Finite nonzero/0	$\pm\infty$
Underflow	$0 < \|x \quad y\| < M_L$	Subnormal numbers
Inexact	$fl\,(x \quad y) \neq x \quad y$	Correctly rounded result

(The numbers of the form $\pm m \times \beta^{L-t}$, $0 < m < \beta^{t-1}$ are called ***subnormal numbers***.) The IEEE arithmetic is a closed system. Every arithmetic operations has a result, whether it is expected mathematically or not. The exceptional operations raise a signal and continue. The arithmetic standard conforms with the standard model (12.4).

The first hardware implementation of the IEEE standard was the Intel 8087 mathematical coprocessor. Since then it is generally accepted and used.

Remark. In the single precision we have about 7 significant digit precision in the decimal system. For double precision we have approximately 16 digit precision in decimals. There also exists an extended precision format of 80 bits, where $t = 63$ and the exponential has 15 bits.

Exercises

12.1-1 The measured values of two resistors are $R_1 = 110.2 \pm 0.3\Omega$ and $R_2 = 65.6 \pm 0.2\Omega$. We connect the two resistors parallel and obtain the circuit resistance $R_e = R_1R_2/(R_1 + R_2)$. Calculate the relative error bounds of the initial data and the approximate value of the resistance R_e. Evaluate the absolute and relative error bounds δR_e and $\delta R_e/R_e$, respectively in the following three ways:
(i) Estimate first δR_e using only the absolute error bounds of the input data, then estimate the relative error bound $\delta R_e/R_e$.
(ii) Estimate first the relative error bound $\delta R_e/R_e$ using only the relative error bounds of the input data, then estimate the absolute error bound δR_e.
(iii) Consider the circuit resistance as a two variable function $R_e = F(R_1, R_2)$.

12.1-2 Assume that $\sqrt{2}$ is calculated with the absolute error bound 10^{-8}. The following two expressions are theoretically equal:
(i) $1/\left(1+\sqrt{2}\right)^6$;
(ii) $99 - 70\sqrt{2}$.
Which expression can be calculated with less relative error and why?

12.1-3 Consider the arithmetic operations as two variable functions of the form $f(x, y) = x \square y$, where $\square \in \{+, -, *, /\}$.
(i) Derive the *error bounds* of the arithmetic operations from the error formula of two variable functions.
(ii) Derive the *condition numbers* of these functions. When are they ill-conditioned?
(iii) Derive *error bounds* for the power function assuming that both the base and the exponent have errors. What is the result if the exponent is exact?
(iv) Let $y = 16x^2$, $x \approx a$ and $y \approx b = 16a^2$. Determine the smallest and the greatest value of a as a function of x such that the *relative error bound* of b should be at most 0.01.

12.1-4 Assume that the number $C = \mathrm{EXP}(4\pi^2/\sqrt{83})$ $(= 76.1967868\ldots)$ is calculated in a 24 bit long mantissa and the exponential function is also calculated with 24 significant bits. Estimate the *absolute error* of the result. Estimate the *relative error* without using the actual value of C.

12.1-5 Consider the emphfloating point number set $F(\beta, t, L, U)$ and show that
(i) Every arithmetic operation can result *arithmetic overflow*;
(ii) Every arithmetic operation can result *arithmetic underflow*.

12.1-6 Show that the following expressions are *numerically unstable* for $x \approx 0$:

(i) $(1-\cos x)/\sin^2 x$;
(ii) $\sin(100\pi + x) - \sin(x)$;
(iii) $2 - \sin x - \cos x - e^{-x}$.
Calculate the values of the above expressions for $x = 10^{-3}, 10^{-5}, 10^{-7}$ and estimate the error. Manipulate the expressions into *numerically stable* ones and estimate the error as well.

12.1-7 How many elements does the set $F = F(\beta, t, L, U)$ have? How many *subnormal numbers* can we find?

12.1-8 If $x, y \geq 0$, then $(x+y)/2 \geq \sqrt{xy}$ and equality holds if and only if $x = y$. Is it true numerically? Check the inequality experimentally for various data (small and large numbers, numbers close to each other or different in magnitude).

12.2. Linear systems of equations

The general form of linear algebraic systems with n unknowns and m equations is given by

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1j}x_j + \cdots + a_{1n}x_n &= b_1 \\ &\vdots \\ a_{i1}x_1 + \cdots + a_{ij}x_j + \cdots + a_{in}x_n &= b_i \\ &\vdots \\ a_{m1}x_1 + \cdots + a_{mj}x_j + \cdots + a_{mn}x_n &= b_m \end{aligned} \tag{12.14}$$

This system can be written in the more compact form

$$Ax = b\,, \tag{12.15}$$

where

$$A = [a_{ij}]_{i,j=1}^{m,n} \in \mathbb{R}^{m\times n},\ x \in \mathbb{R}^n,\ b \in \mathbb{R}^m\,.$$

The systems is called *underdetermined* if $m < n$. For $m > n$, the systems is called *overdetermined*. Here we investigate only the case $m = n$, when the coefficient matrix A is square. We also assume that the inverse matrix A^{-1} exists (or equivalently $\det(A) \neq 0$). Under this assumption the linear system $Ax = b$ has exactly one solution: $x = A^{-1}b$.

12.2.1. Direct methods for solving linear systems

Triangular linear systems

Definition 12.4 *The matrix $A = [a_{ij}]_{i,j=1}^n$ is **upper triangular** if $a_{ij} = 0$ for all $i > j$. The matrix A is **lower triangular** if $a_{ij} = 0$ for all $i < j$.*

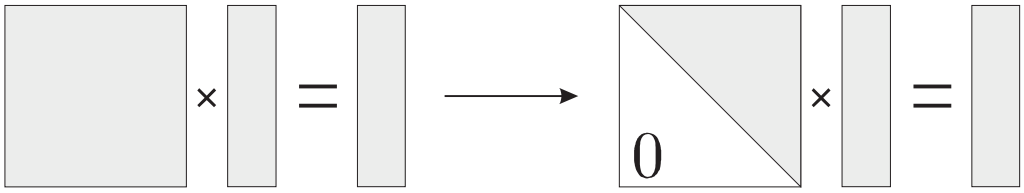


Figure 12.2. Gaussian elimination.

For example the general form of the upper triangular matrices is the following:

$$\begin{bmatrix} * & * & \cdots & \cdots & * \\ 0 & * & & & \vdots \\ \vdots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & * & * \\ 0 & \cdots & \cdots & 0 & * \end{bmatrix}.$$

We note that the diagonal matrices are both lower and upper triangular. It is easy to show that $\det(A) = a_{11}a_{22}\dots a_{nn}$ holds for the upper or lower triangular matrices. It is easy to solve linear systems with triangular coefficient matrices. Consider the following upper triangular linear system:

$$\begin{array}{rcccccc} a_{11}x_1+ & \cdots & +a_{1i}x_i+ & \cdots & +a_{1n}x_n & = & b_1 \\ & \ddots & \vdots & & \vdots & & \vdots \\ & & a_{ii}x_i+ & \cdots & +a_{in}x_n & = & b_i \\ & & & \ddots & \vdots & & \vdots \\ & & & & a_{nn}x_n & = & b_n \end{array}$$

This can be solved by the so called *back substitution* algorithm.

BACK-SUBSTITUTION(A, b, n)

1 $x_n \leftarrow b_n/a_{nn}$
2 **for** $i \leftarrow n-1$ **downto** 1
3 **do** $x_i \leftarrow (b_i - \sum_{j=i+1}^{n} a_{ij}x_j)/a_{ii}$
4 **return** x

The solution of lower triangular systems is similar.

The Gauss method The Gauss method or Gaussian elimination (GE) consists of two phases:
I. The linear system $Ax = b$ is transformed to an equivalent upper triangular system using elementary operations (see Figure 12.2).
II. The obtained upper triangular system is then solved by the back substitution algorithm.

The first phase is often called the elimination or forward phase. The second phase

of GE is called the backward phase. The elementary operations are of the following three types:

1. Add a multiple of one equation to another equation.
2. Interchange two equations.
3. Multiply an equation by a nonzero constant.

The elimination phase of GE is based on the following observation. Multiply equation k by $\gamma \neq 0$ and subtract it from equation i:

$$(a_{i1} - \gamma a_{k1})\, x_1 + \cdots + (a_{ij} - \gamma a_{kj})\, x_j + \cdots + (a_{in} - \gamma a_{kn})\, x_n = b_i - \gamma b_k.$$

If $a_{kj} \neq 0$, then by choosing $\gamma = a_{ij}/a_{kj}$, the coefficient of x_j becomes 0 in the new equivalent equation, which replaces equation i. Thus we can eliminate variable x_j (or coefficient a_{ij}) from equation i.

The Gauss method eliminates the coefficients (variables) under the main diagonal of A in a systematic way. First variable x_1 is eliminated from equations $i = 2, \ldots, n$ using equation 1, then x_2 is eliminated from equations $i = 3, \ldots, n$ using equation 2, and so on.

Assume that the unknowns are eliminated in the first $(k-1)$ columns under the main diagonal and the resulting linear system has the form

$$\begin{array}{cccccccccccc}
a_{11}x_1+ & \cdots & \cdots & +a_{1k}x_k & + & \cdots & + & a_{1n}x_n & = & b_1 \\
 & \ddots & & \vdots & & & & \vdots & & \vdots \\
 & & \ddots & \vdots & & & & \vdots & & \vdots \\
 & & & a_{kk}x_k & + & \cdots & + & a_{kn}x_n & = & b_k \\
 & & & \vdots & & & & \vdots & & \vdots \\
 & & & a_{ik}x_k & + & \cdots & + & a_{in}x_n & = & b_i \\
 & & & \vdots & & & & \vdots & & \vdots \\
 & & & a_{nk}x_k & + & \cdots & + & a_{nn}x_n & = & b_n
\end{array}$$

If $a_{kk} \neq 0$, then multiplying row k by γ and subtracting it from equation i we obtain

$$(a_{ik} - \gamma a_{kk})x_k + (a_{i,k+1} - \gamma a_{k,k+1})x_{k+1} + \cdots + (a_{in} - \gamma a_{kn})x_n = b_i - \gamma b_k\ .$$

Since $a_{ik} - \gamma a_{kk} = 0$ for $\gamma = a_{ik}/a_{kk}$, we eliminated the coefficient a_{ik} (variable x_k) from equation $i > k$. Repeating this process for $i = k+1, \ldots, n$ we can eliminate the coefficients under the main diagonal entry a_{kk}. Next we denote by $A\,[i, j]$ the element a_{ij} of matrix A and by $A\,[i, j : n]$ the vector $[a_{ij}, a_{i,j+1}, \ldots, a_{in}]$. The Gauss method has the following form (where the *pivoting* discussed later is also included):

GAUSS-METHOD(A, b)

```
1  ▷ Forward phase:
2  n ← rows[A]
3  for k ← 1 to n − 1
4      do { pivoting and interchange of rows and columns}
5          for i ← k + 1 to n
6              do γ_ik ← A[i, k] / A[k, k]
7                  A[i, k + 1 : n] ← A[i, k + 1 : n] − γ_ik ∗ A[k, k + 1 : n]
8                  b_i ← b_i − γ_ik b_k
9  ▷ Backward phase: see the back substitution algorithm.
10 return x
```

The algorithm overwrites the original matrix A and vector b. It does not write however the zero entries under the main diagonal since these elements are not necessary for the second phase of the algorithm. Hence the lower triangular part of matrix A can be used to store information for the LU decomposition of matrix A.

The above version of the Gauss method can be performed only if the elements a_{kk} occurring in the computation are not zero. For this and numerical stability reasons we use the Gaussian elimination with *pivoting*.

The Gauss method with pivoting If $a_{kk} = 0$, then we can interchange row k with another row, say i, so that the new entry (a_{ki}) at position (k, k) should be nonzero. If this is not possible, then all the coefficients $a_{kk}, a_{k+1,k}, \ldots, a_{nk}$ are zero and $\det(A) = 0$. In the latter case $Ax = b$ has no unique solution. The element a_{kk} is called the k^{th} ***pivot element***. We can always select new pivot elements by interchanging the rows. The selection of the pivot element has a great influence on the reliability of the computed results. The simple fact that we divide by the pivot element indicates this influence. We recall that $\delta(a/b)$ is proportional to $1/|b|^2$. It is considered advantageous if the pivot element is selected so that it has the greatest possible modulus. The process of selecting the pivot element is called *pivoting*. We mention the following two pivoting processes.

Partial pivoting: At the k^{th} step, interchange the rows of the matrix so the largest remaining element, say a_{ik}, in the k^{th} column is used as pivot. After the pivoting we have

$$|a_{kk}| = \max_{k \le i \le n} |a_{ik}| \ .$$

Complete pivoting: At the k^{th} step, interchange both the rows and columns of the matrix so that the largest element, say a_{ij}, in the remaining matrix is used as pivot After the pivoting we have

$$|a_{kk}| = \max_{k \le i,j \le n} |a_{ij}| \ .$$

Note that the interchange of two columns implies the interchange of the corresponding unknowns. The significance of pivoting is well illustrated by the following

Example 12.6 The exact solution of the linear system

$$\begin{array}{rcrcl} 10^{-17}x & + & y & = & 1 \\ x & + & y & = & 2 \end{array}$$

is $x = 1/(1-10^{-17})$ and $y = 1 - 10^{-17}/(1-10^{-17})$. The MATLAB program gives the result $x = 1$, $y = 1$ and this is the best available result in standard double precision arithmetic. Solving this system with the Gaussian elimination without pivoting (also in double precision) we obtain the catastrophic result $x = 0$ and $y = 1$. Using partial pivoting with the Gaussian elimination we obtain the best available numerical result $x = y = 1$.

Remark 12.5 *Theoretically we do not need pivoting in the following cases: 1. If A is symmetric and positive definite ($A \in \mathbb{R}^{n\times n}$ is positive definite $\Leftrightarrow x^T Ax > 0$, $\forall x \in \mathbb{R}^n$, $x \neq 0$). 2. If A is diagonally dominant in the following sense:*

$$|a_{ii}| > \sum_{j\neq i} |a_{ij}| \quad (1 \leq i \leq n) \ .$$

In case of symmetric and positive definite matrices we use the Cholesky method which is a special version of the Gauss-type methods.

During the Gaussian elimination we obtain a sequence of equivalent linear systems

$$A^{(0)}x = b^{(0)} \to A^{(1)}x = b^{(1)} \to \cdots \to A^{(n-1)}x = b^{(n-1)} \ ,$$

where

$$A^{(0)} = A, \quad A^{(k)} = \left[a_{ij}^{(k)}\right]_{i,j=1}^{n} .$$

Note that matrices $A^{(k)}$ are stored in the place of $A = A^{(0)}$. The last coefficient matrix of phase I has the form

$$A^{(n-1)} = \begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & \cdots & a_{1n}^{(0)} \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & \cdots & a_{nn}^{(n-1)} \end{bmatrix} ,$$

where $a_{kk}^{(k-1)}$ is the k^{th} pivot element. The ***growth factor of pivot elements*** is given by

$$\rho = \rho_n = \max_{1\leq k\leq n} \left| a_{kk}^{(k-1)} / a_{11}^{(0)} \right| \ .$$

Wilkinson proved that the error of the computed solution is proportional to the growth factor ρ and the bounds

$$\rho \leq \sqrt{n} \left(2 \cdot 3^{\frac{1}{2}} \cdots n^{\frac{1}{n-1}} \right)^{\frac{1}{2}} \sim c n^{\frac{1}{2}} n^{\frac{1}{4} log(n)}$$

and

$$\rho \leq 2^{n-1}$$

hold for complete and partial pivoting, respectively. Wilkinson conjectured that $\rho \leq n$ for complete pivoting. This has been proved by researchers for small values of n. Statistical investigations on random matrices ($n \leq 1024$) indicate that the average of ρ is $\Theta\left(n^{2/3}\right)$ for the partial pivoting and $\Theta\left(n^{1/2}\right)$ for the complete pivoting. Hence the case $\rho > n$ hardly occurs in the statistical sense.

We remark that Wilkinson constructed a linear system on which $\rho = 2^{n-1}$ for the partial pivoting. Hence Wilkinson's bound for ρ is sharp in the case of partial pivoting. There also exist examples of linear systems concerning discretisations of differential and integral equations, where ρ is increasing exponentially if Gaussian elimination is used with partial pivoting.

The growth factor ρ can be very large, if the Gaussian elimination is used without pivoting. For example, $\rho = \rho_4(A) = 1.23 \times 10^5$, if

$$A = \begin{bmatrix} 1.7846 & -0.2760 & -0.2760 & -0.2760 \\ -3.3848 & 0.7240 & -0.3492 & -0.2760 \\ -0.2760 & -0.2760 & 1.4311 & -0.2760 \\ -0.2760 & -0.2760 & -0.2760 & 0.7240 \end{bmatrix}.$$

Operations counts The Gauss method gives the solution of the linear system $Ax = b$ ($A \in \mathbb{R}^{n \times n}$) in a finite number of steps and arithmetic operations ($+, -, *, /$). The amount of necessary arithmetic operations is an important characteristic of the direct linear system solvers, since the CPU time is largely proportional to the number of arithmetic operations. It was also observed that the number of additive and multiplicative operations are nearly the same in the numerical algorithms of linear algebra. For measuring the cost of such algorithms C. B. Moler introduced the concept of flop.

Definition 12.6 ***One (old) flop*** *is the computational work necessary for the operation $s = s + x * y$ (1 addition + 1 multiplication).* ***One (new) flop*** *is the computational work necessary for any of the arithmetic operations $+, -, *, /$.*

The new flop can be used if the computational time of additive and multiplicative operations are approximately the same. Two new flops equals to one old flop. Here we use the notion of old flop.

For the Gauss method a simple counting gives the number of additive and multiplicative operations.

Theorem 12.7 *The computational cost of the Gauss method is $n^3/3 + \Theta(n^2)$ flops.*

V. V. Klyuyev and N. Kokovkin-Shcherbak proved that if only elementary row and column operations (multiplication of row or column by a number, interchange of rows or columns, addition of a multiple of row or column to another row or column) are allowed, then the linear system $Ax = b$ cannot be solved in less than $n^3/3 + \Omega(n^2)$ flops.

Using fast matrix inversion procedures we can solve the $n \times n$ linear system $Ax = b$ in $O(n^{2.808})$ flops. These theoretically interesting algorithms are not used in practice since they are considered as numerically unstable.

The LU-decomposition In many cases it is easier to solve a linear system if the coefficient matrix can be decomposed into the product of two triangular matrices.

Definition 12.8 *The matrix $A \in \mathbb{R}^{n\times n}$ has an LU-**decomposition**, if $A = LU$, where $L \in \mathbb{R}^{n\times n}$ is lower and $U \in \mathbb{R}^{n\times n}$ is upper triangular matrix.*

The LU-decomposition is not unique. If a nonsingular matrix has an LU-decomposition, then it has a particular LU-decomposition, where the main diagonal of a given component matrix consists of 1's. Such triangular matrices are called ***unit (upper or lower) triangular*** matrices. The LU decomposition is unique, if L is set to be lower unit triangular or U is set to be unit upper triangular.

The LU-decomposition of nonsingular matrices is closely related to the Gaussian elimination method. If $A = LU$, where L is unit lower triangular, then $l_{ik} = \gamma_{ik}$ $(i > k)$, where γ_{ik} is given by the Gauss algorithm. The matrix U is the upper triangular part of the matrix we obtain at the end of the forward phase. The matrix L can also be derived from this matrix, if the columns of the lower triangular part are divided by the corresponding main diagonal elements. We remind that the first phase of the Gaussian elimination does not annihilate the matrix elements under the main diagonal. It is clear that a nonsingular matrix has LU-decomposition if and only if $a_{kk}^{(k-1)} \neq 0$ holds for each pivot element for the Gauss method without pivoting.

Definition 12.9 *A matrix $P \in \mathbb{R}^{n\times n}$ whose every row and column has one and only one non-zero element, that element being 1, is called a **permutation matrix**.*

In case of partial pivoting we permute the rows of the coefficient matrix (multiply A by a permutation matrix on the left) so that $a_{kk}^{(k-1)} \neq 0$ $(k = 1, \ldots, n)$ holds for a nonsingular matrix. Hence we have

Theorem 12.10 *If $A \in \mathbb{R}^{n\times n}$ is nonsingular then there exists a permutation matrix P such that PA has an LU-decomposition.*

The the algorithm of LU-decomposition is essentially the Gaussian elimination method. If pivoting is used then the interchange of rows must also be executed on the elements under the main diagonal and the permutation matrix P must be recorded. A vector containing the actual order of the original matrix rows is obviously sufficient for this purpose.

The LU- and Cholesky-methods Let $A = LU$ and consider the equation $Ax = b$. Since $Ax = LUx = L(Ux) = b$, we can decompose $Ax = b$ into the equivalent linear system $Ly = b$ and $Ux = b$, where L is lower triangular and U is upper triangular.

LU-METHOD(A, b)

1 Determine the LU-decomposition $A = LU$.
2 Solve $Ly = b$.
3 Solve $Ux = y$.
4 **return** x

Remark. In case of partial pivoting we obtain the decomposition $\hat{A} = PA = LU$ and we set $\hat{b} = Pb$ instead of b.

In the first phase of the Gauss method we produce decomposition $A = LU$ and the equivalent linear system $Ux = L^{-1}b$ with upper triangular coefficient matrix. The latter is solved in the second phase. In the LU-method we decompose the first phase of the Gauss method into two steps. In the first step we obtain only the decomposition $A = LU$. In the second step we produce the vector $y = L^{-1}b$. The third step of the algorithm is identical with the second phase of the original Gauss method.

The LU-method is especially advantageous if we have to solve several linear systems with the same coefficient matrix:

$$Ax = b_1,\ Ax = b_2, \ldots,\ Ax = b_k.$$

In such a case we determine the LU-decomposition of matrix A only once, and then we solve the linear systems $Ly_i = b_i$, $Ux_i = y_i$ $(x_i, y_i, b_i \in \mathbf{R}^n,\ i = 1, \ldots, k)$. The computational cost of this process is $n^3/3 + kn^2 + \Theta(kn)$ flops.

The *inversion of a matrix* $A \in \mathbb{R}^{n \times n}$ can be done as follows:

1. Determine the LU-decomposition $A = LU$. .

2. Solve $Ly_i = e_i$, $Ux_i = y_i$ (e_i is the i^{th} unit vector $i = 1, \ldots, n$).

The inverse of A is given by $A^{-1} = [x_1, \ldots, x_n]$. The computational cost of the algorithm is $4n^3/3 + \Theta\left(n^2\right)$ flops.

The *LU*-method with pointers This implementation of the LU-method is known since the 60's. Vector P contains the indices of the rows. At the start we set $P[i] = i$ $(1 \leq i \leq n)$. When exchanging rows we exchange only those components of vector P that correspond to the rows.

LU-Method-with-Pointers(A, b)

```
1  n ← rows[A]
2  P ← [1, 2, ..., n]
```

```
3  for k ← 1 to n − 1
4      do compute index t such that |A[P[t], k]| = max_{k≤i≤n} |A[P[i], k]| .
5         if k < t
6            then exchange the components P[k] and P[t].
7         for i ← k + 1 to n
8             do A[P[i], k] ← A[P[i], k] / A[P[k], k]
9                A[P[i], k + 1 : n] ← A[P[i], k + 1 : n] − A[P[i], k] ∗ A[P[k], k + 1 : n]
10 for i ← 1 to n
11     do s ← 0
12        for j ← 1 to i − 1
13            do s ← s + A[P[i], j] ∗ x[j]
14        x[i] ← b[P[i]] − s
15 for i ← n downto 1
16     do s ← 0
17        for j ← i + 1 to n
18            s ← s + A[P[i], j] ∗ x[j]
19        x[i] ← (x[i] − s) / A[P[i], i]
20 return x
```

If $A \in \mathbb{R}^{n\times n}$ is *symmetric and positive definite*, then it can be decomposed in the form $A = LL^T$, where L is lower triangular matrix. The LL^T-decomposition is called the ***Cholesky-decomposition*** . In this case we can save approximately half of the storage place for A and half of the computational cost of the LU-decomposition (LL^T-decomposition). Let

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & l_{nn} \end{bmatrix}.$$

Observing that only the first k elements may be nonzero in the k^{th} column of L^T we obtain that

$$\begin{aligned} a_{kk} &= l_{k1}^2 + l_{k2}^2 + \cdots + l_{k,k-1}^2 + l_{kk}^2, \\ a_{ik} &= l_{i1}l_{k1} + l_{i2}l_{k2} + \cdots + l_{i,k-1}l_{k,k-1} + l_{ik}l_{kk} \quad (i = k+1, \ldots, n) \ . \end{aligned}$$

This gives the formulae

$$\begin{aligned} l_{kk} &= (a_{kk} - \sum_{j=1}^{k-1} l_{kj}^2)^{1/2}, \\ l_{ik} &= (a_{ik} - \sum_{j=1}^{k-1} l_{ij}l_{kj})/l_{kk} \quad (i = k+1, \ldots, n) \ . \end{aligned}$$

Using the notation $\sum_{j=i}^{k} s_j = 0$ $(k < i)$ we can formulate the Cholesky-method

as follows.

CHOLESKY-METHOD(A)

```
1  n ← rows[A]
2  for k ← 1  to n
3       do  a_kk ← (a_kk − Σ_{j=1}^{k−1} a_kj^2)^{1/2}
4           for i ← k + 1  to  n
5                do  a_ik ← (a_ik − Σ_{j=1}^{k−1} a_ij a_kj)/a_kk
6  return  A
```

The lower triangular part of A contains L. The computational cost of the algorithm is $n^3/6 + \Theta(n^2)$ flops and n square roots. The algorithm, which can be considered as a special case of the Gauss-methods, does not require pivoting, at least in principle.

The LU- and Cholesky-methods on banded matrices It often happens that linear systems have *banded coefficient matrices*.

Definition 12.11 *Matrix $A \in \mathbb{R}^{n\times n}$ is banded with lower bandwidth p and upper bandwidth q if*

$$a_{ij} = 0, \quad \text{if } i > j + p \text{ or } j > i + q \,.$$

The possibly non-zero elements a_{ij} $(i-p \le j \le i+q)$ form a band like structure. Schematically A has the form

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1,1+q} & 0 & \cdots & \cdots & 0 \\ a_{21} & a_{22} & & & & \ddots & & & \vdots \\ \vdots & & \ddots & & & & \ddots & & \vdots \\ a_{1+p,1} & & & \ddots & & & & \ddots & 0 \\ 0 & \ddots & & & \ddots & & & & a_{n-q,n} \\ \vdots & & \ddots & & & \ddots & & & \vdots \\ \vdots & & & \ddots & & & \ddots & & \vdots \\ \vdots & & & & \ddots & & & \ddots & a_{n-1,n} \\ 0 & \cdots & \cdots & \cdots & 0 & a_{n,n-p} & \cdots & a_{n,n-1} & a_{nn} \end{bmatrix} .$$

The *banded matrices* yield very efficient algorithms if p and q are significantly less than n. If a banded matrix A with lower bandwidth p and upper bandwidth q has an LU-decomposition, then both L and U are banded with lower bandwidth p and upper bandwidth q, respectively.

Next we give the LU-method for banded matrices in three parts.

THE-LU-DECOMPOSITION-OF-BANDED-MATRIX(A, n, p, q)

```
1  for  k ← 1  to  n − 1
2       do for i ← k + 1  to  min {k + p, n}
3             do a_ik ← a_ik/a_kk
4                for j ← k + 1  to  min {k + q, n}
5                   do a_ij ← a_ij − a_ik a_kj
6  return A
```

Entry a_{ij} is overwritten by l_{ij}, if $i > j$ and by u_{ij}, if $i \leq j$. The computational cost of is $c\,(p, q)$ flops, where

$$c\,(p,q) = \begin{cases} npq - \frac{1}{2}pq^2 - \frac{1}{6}p^3 + pn, & p \leq q \\ npq - \frac{1}{2}qp^2 - \frac{1}{6}q^3 + qn, & p > q \end{cases}$$

The following algorithm overwrites b by the solution of equation $Ly = b$.

SOLUTION-OF-BANDED-UNIT-LOWER-TRIANGULAR-SYSTEM(L, b, n, p)

```
1  for i ← 1  to  n
2       do  b_i ← b_i − Σ_{j=max{1,i−p}}^{i−1} l_ij b_j
3  return b
```

The total cost of the algorithm is $np - p^2/2$ flops. The next algorithm overwrites vector b by the solution of $Ux = b$.

SOLUTION-OF-BANDED-UPPER-TRIANGULAR-SYSTEM(U, b, n, q)

```
1  for i ← n downto 1
2       do  b_i ← ( b_i − Σ_{j=i+1}^{min{i+q,n}} u_ij b_j ) / u_ii
3  return b
```

The computational cost is $n\,(q+1) - q^2/2$ flops.

Assume that $A \in \mathbb{R}^{n\times n}$ is symmetric, positive definite and banded with lower bandwidth p. The banded version of the Cholesky-methods is given by

CHOLESKY-DECOMPOSITION-OF-BANDED-MATRICES(A, n, p)

```
1  for i ← 1  to  n
2       do for  j ← max {1, i − p}  to  i − 1
3             do a_ij ← ( a_ij − Σ_{k=max{1,i−p}}^{j−1} a_ik a_jk ) / a_jj
4          a_ii ← ( a_ii − Σ_{k=max{1,i−p}}^{i−1} a_ik^2 )^{1/2}
5  return A
```

The elements a_{ij} are overwritten by l_{ij} $(i \geq j)$. The total amount of work is given by $\left(np^2/2\right) - \left(p^3/3\right) + (3/2)\left(np - p^2\right)$ flops és n square roots.

Remark. If $A \in \mathbb{R}^{n\times n}$ has lower bandwidth p and upper bandwidth q and partial pivoting takes place, then the upper bandwidth of U increases up to $\hat{q} = p + q$.

12.2.2. Iterative methods for linear systems

There are several iterative methods for solving linear systems of algebraic equations. The best known iterative algorithms are the classical *Jacobi-*, the *Gauss-Seidel-* and the *relaxation methods*. The greatest advantage of these iterative algorithms is their easy implementation to large systems. At the same time they usually have slow convergence. However for parallel computers the *multisplitting* iterative algorithms seem to be efficient.

Consider the iteration

$$x_i = Gx_{i-1} + b \quad (i = 1, 2, \ldots)$$

where $G \in \mathbb{R}^{n\times n}$ és $x_0, b \in \mathbb{R}^n$. It is known that $\{x_i\}_{i=0}^{\infty}$ converges for all x_0, $b \in \mathbb{R}^n$ if and only if the spectral radius of G satisfies $\rho(G) < 1$ ($\rho(G) = \max|\lambda| \mid \lambda$ is an eigenvalue of G). In case of convergence $x_i \to x^* = (I - G)^{-1} b$, that is we obtain the solution of the equation $(I - G)x = b$. The speed of convergence depends on the spectral radius $\rho(G)$. Smaller the spectral radius$\rho(G)$, faster the convergence.

Consider now the linear system

$$Ax = b\,,$$

where $A \in \mathbb{R}^{n\times n}$ is nonsingular. The matrices $M_l, N_l, E_l \in \mathbf{R}^{n\times n}$ form a multisplitting of A if

(i) $A = M_i - N_i$, $i = 1, 2, \ldots, L$,
(ii) M_i is nonsingular, $i = 1, 2, \ldots, L$,
(iii) E_i is non-negative diagonal matrix, $i = 1, 2, \ldots, L$,
(iv) $\sum_{i=1}^{L} E_i = I$.

Let $x_0 \in \mathbb{R}^n$ be a given initial vector. The multisplitting iterative method is the following.

MULTISPLITTING-ITERATION$(x_0, b, L, M_l, N_l, E_l, l = 1, \ldots, L)$

```
1  i ← 0
2  while exit condition=FALSE
3        do i ← i + 1
4           for l ← 1 to L
5                do M_l y_l ← N_l x_{i-1} + b
6           x_i ← Σ_{l=1}^{L} E_l y_l
7  return x_i
```

It is easy to show that $y_l = M_l^{-1} N_l x_{i-1} + M_l^{-1} b$ and

$$\begin{aligned} x_i &= \sum_{l=1}^{L} E_l y_l = \sum_{l=1}^{L} E_l M_l^{-1} N_l x_{i-1} + \sum_{l=1}^{L} E_l M_l^{-1} b \\ &= Hx_{i-1} + c\,. \end{aligned}$$

Thus the condition of convergence is $\rho(H) < 1$. The *multisplitting iteration* is a true *parallel algorithm* because we can solve L linear systems parallel in each iteration

(synchronised parallelism). The bottleneck of the algorithm is the computation of iterate x_i.

The selection of matrices M_i and E_i is such that the solution of the linear system $M_i y = c$ should be cheap. Let $S_1, S_2, \ldots, S_L$ be a partition of $\{1, \ldots, n\}$, that is $S_i \neq \emptyset$, $S_i \cap S_j = \emptyset$ $(i \neq j)$ and $\cup_{i=1}^{L} S_i = \{1, \ldots, n\}$. Furthermore let $S_i \subseteq T_i \subseteq \{1, \ldots, n\}$ $(i = 1, \ldots, L)$ be such that $S_l \neq T_l$ for at least one l.

The ***non-overlapping block Jacobi splitting*** of A is given by

$$M_l = \left[M_{ij}^{(l)}\right]_{i,j=1}^{n}, \quad M_{ij}^{(l)} = \begin{cases} a_{ij}, & \text{if } i,j \in S_l \\ a_{ii}, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases},$$

$$N_l = M_l - A,$$

$$E_l = \left[E_{ij}^{(l)}\right]_{i,j=1}^{n}, \quad E_{ij}^{(l)} = \begin{cases} 1, & \text{if } i = j \in S_l \\ 0, & \text{otherwise} \end{cases}$$

for $l = 1, \ldots, L$.

Define now the simple splitting

$$A = M - N\ ,$$

where M is nonsingular,

$$M = [M_{ij}]_{i,j=1}^{n}, \quad M_{ij} = \begin{cases} a_{ij}, & \text{if } i,j \in S_l \text{ for some } l \in \{1, \ldots, n\} \\ 0, & \text{otherwise} \end{cases}.$$

It can be shown that

$$H = \sum_{l=1}^{L} E_l M_l^{-1} N_l = M^{-1} N$$

holds for the non-overlapping block Jacobi multisplitting.

The ***overlapping block Jacobi multisplitting*** of A is defined by

$$\widetilde{M}_l = \left[\widetilde{M}_{ij}^{(l)}\right]_{i,j=1}^{n}, \quad \widetilde{M}_{ij}^{(l)} = \begin{cases} a_{ij}, & \text{if } i,j \in T_l \\ a_{ii}, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases},$$

$$\widetilde{N}_l = \widetilde{M}_l - A,$$

$$\widetilde{e}_l = \left[\widetilde{e}_{ij}^{(l)}\right]_{i,j=1}^{n}, \quad E_{ii}^{(l)} = 0, \text{ if } i \notin T_l$$

for $l = 1, \ldots, L$.

A nonsingular matrix $A \in \mathbb{R}^{n \times n}$ is called an M-matrix, if $a_{ij} \leq 0$ $(i \neq j)$ and all the elements of A^{-1} are nonnegative.

Theorem 12.12 *Assume that* $A \in \mathbb{R}^{n\times n}$ *is nonsingular* M*-matrix,* $\{M_i, N_i, E_i\}_{i=1}^{L}$ *is a non-overlapping,* $\left\{\widetilde{M}_i, \widetilde{N}_i, E_i\right\}_{i=1}^{L}$ *is an overlapping block Jacobi multisplitting of* A*, where the weighting matrices* E_i *are the same. The we have*

$$\rho\left(\widetilde{H}\right) \leq \rho\left(H\right) < 1 \ ,$$

where $H = \sum_{l=1}^{L} E_l M_l^{-1} N_l$ *and* $\widetilde{H} = \sum_{l=1}^{L} E_l \widetilde{M}_l^{-1} \widetilde{N}_l$.

We can observe that both iteration procedures are convergent and the convergence of the overlapping multisplitting is not slower than that of the non-overlapping procedure. The theorem remains true if we use block Gauss-Seidel multisplittings instead of the block Jacobi multisplittings. In this case we replace the above defined matrices M_i and $\widetilde{M}_i$ with their lower triangular parts.

The multisplitting algorithm has multi-stage and asynchronous variants as well.

12.2.3. Error analysis of linear algebraic systems

We analyse the direct and inverse errors. We use the following notations and concepts. The exact (theoretical) solution of $Ax = b$ is denoted by x, while any approximate solution is denoted by $\hat{x}$. The direct error of the approximate solution is given by $\Delta x = \hat{x} - x$. The quantity $r = r\left(y\right) = Ay - b$ is called the ***residual error***. For the exact solution $r\left(x\right) = 0$, while for the approximate solution

$$r\left(\hat{x}\right) = A\hat{x} - b = A\left(\hat{x} - x\right) = A\Delta x \ .$$

We use various models to estimate the inverse error. In the most general case we assume that the computed solution $\hat{x}$ satisfies the linear system$\hat{A}\hat{x} = \hat{b}$, where $\hat{A} = A + \Delta A$ and $\hat{b} = b + \Delta b$. The quantities ΔA and Δb are called the inverse errors.

One has to distinguish between the sensitivity of the problem and the stability of the solution algorithm. By ***sensitivity of a problem*** we mean the sensitivity of the solution to changes in the input parameters (data). By the ***stability (or sensitivity) of an algorithm*** we mean the influence of computational errors on the computed solution. We measure the sensitivity of a problem or algorithm in various ways. One such characterization is the *condition number*„condition number", which compares the relative errors of the input and output values.

The following general principles are used when applying any algorithm:

- We use only stable or well-conditioned algorithms.
- We cannot solve an unstable (ill-posed or ill-conditioned) problem with a general purpose algorithm, in general.

Sensitivity analysis Assume that we solve the perturbed equation

$$A\hat{x} = b + \Delta b \tag{12.16}$$

instead of the original $Ax = b$. Let $\widehat{x} = x + \Delta x$ and investigate the difference of the two solutions.

Theorem 12.13 *If A is nonsingular and $b \neq 0$, then*

$$\frac{\|\Delta x\|}{\|x\|} \leq \operatorname{cond}(A) \frac{\|\Delta b\|}{\|b\|} = \operatorname{cond}(A) \frac{\|r(\hat{x})\|}{\|b\|}, \tag{12.17}$$

where $\operatorname{cond}(A) = \|A\| \left\|A^{-1}\right\|$ *is the condition number of* A.

Here we can see that the condition number of A may strongly influence the relative error of the perturbed solution $\widehat{x}$. A linear algebraic system is said to be *well-conditioned* if $\operatorname{cond}(A)$ is small, and *ill-conditioned*, if $\operatorname{cond}(A)$ is big. It is clear that the terms „small" and „big" are relative and the condition number depends on the norm chosen. We identify the applied norm if it is essential for some reason. For example $\operatorname{cond}_\infty(A) = \|A\|_\infty \left\|A^{-1}\right\|_\infty$. The next example gives possible geometric characterization of the condition number.

Example 12.7 The linear system

$$\begin{array}{rcrcl} 1000x_1 & + & 999x_2 & = & b_1 \\ 999x_1 & + & 998x_2 & = & b_2 \end{array}$$

is ill-conditioned ($\operatorname{cond}_\infty(A) = 3.99 \times 10^6$). The two lines, whose meshpoint defines the system, are almost parallel. Therefore if we perturb the right hand side, the new meshpoint of the two lines will be far from the previous meshpoint.

The inverse error is Δb in the sensitivity model under investigation. Theorem 12.13 gives an estimate of the direct error which conforms with the thumb rule. It follows that we can expect a small relative error of the perturbed solution $\widehat{x}$, if the condition number of A is small.

Example 12.8 Consider the linear system $Ax = b$ with

$$A = \begin{matrix} 1+\epsilon & 1 \\ 1 & 1 \end{matrix}, \quad b = \begin{matrix} 1 \\ 1 \end{matrix}, \quad x = \begin{matrix} 0 \\ 1 \end{matrix}.$$

Let $\hat{x} = \begin{matrix} 2 \\ -1 \end{matrix}$. Then $r = \begin{matrix} 2\epsilon \\ 0 \end{matrix}$ and $\|r\|_\infty / \|b\|_\infty = 2\epsilon$, but $\|\hat{x} - x\|_\infty / \|x\|_\infty = 2$.

Consider now the perturbed linear system

$$(A + \Delta A)\hat{x} = b \tag{12.18}$$

instead of $Ax = b$. It can be proved that for this perturbation model there exist more than one *inverse errors*„inverse error" among which $\Delta A = -r(\widehat{x})\widehat{x}^T / \widehat{x}^T\widehat{x}$ is the inverse error with minimal spectral norm, provided that $\widehat{x}, r(\widehat{x}) \neq 0$.

The following theorem establish that for small relative residual error the relative inverse error is also small.

Theorem 12.14 *Assume that $\hat{x} \neq 0$ is the approximate solution of $Ax = b$, $\det(A) \neq 0$ and $b \neq 0$. If $\|r(\hat{x})\|_2 / \|b\|_2 = \alpha < 1$, the the matrix $\Delta A = -r(\hat{x})\hat{x}^T / \hat{x}^T\hat{x}$ satisfies $(A + \Delta A)\hat{x} = b$ and $\|\Delta A\|_2 / \|A\|_2 \leq \alpha / (1 - \alpha)$.*

If the relative inverse error and the condition number of A are small, then the relative residual error is small.

Theorem 12.15 *If* $r(\hat{x}) = A\hat{x} - b$, $(A + \Delta A)\hat{x} = b$, $A \neq 0$, $b \neq 0$ *and* $\operatorname{cond}(A) \frac{\|\Delta A\|}{\|A\|} < 1$, *then*

$$\frac{\|r(\hat{x})\|}{\|b\|} \leq \frac{\operatorname{cond}(A) \frac{\|\Delta A\|}{\|A\|}}{1 - \operatorname{cond}(A) \frac{\|\Delta A\|}{\|A\|}} . \tag{12.19}$$

If A is ill-conditioned, then Theorem 12.15 is not true.

Example 12.9 Let $A = \begin{matrix} 1+\epsilon & 1 \\ 1 & 1-\epsilon \end{matrix}$, $\Delta A = \begin{matrix} 0 & 0 \\ 0 & \epsilon^2 \end{matrix}$ and $b = \begin{matrix} 1 \\ -1 \end{matrix}$, $(0 < \epsilon \ll 1)$. Then $\operatorname{cond}_\infty(A) = (2+\epsilon)^2/\epsilon^2 \approx 4/\epsilon^2$ and $\|\Delta A\|_\infty / \|A\|_\infty = \epsilon^2/(2+\epsilon) \approx \epsilon^2/2$. Let

$$\hat{x} = (A + \Delta A)^{-1} b = \frac{1}{\epsilon^3} \begin{matrix} 2 - \epsilon + \epsilon^2 \\ -2 - \epsilon \end{matrix} \approx \begin{matrix} 2/\epsilon^3 \\ -2/\epsilon^3 \end{matrix} .$$

Then $r(\hat{x}) = A\hat{x} - b = \begin{matrix} 0 \\ 2/\epsilon + 1 \end{matrix}$ and $\|r(\hat{x})\|_\infty / \|b\|_\infty = 2/\epsilon + 1$, which is not small.

In the most general case we solve the perturbed equation

$$(A + \Delta A)\hat{x} = b + \Delta b \tag{12.20}$$

instead of $Ax = b$. The following general result holds.

Theorem 12.16 *If* A *is nonsingular,* $\operatorname{cond}(A) \frac{\|\Delta A\|}{\|A\|} < 1$ *and* $b \neq 0$, *then*

$$\frac{\|\Delta x\|}{\|x\|} \leq \frac{\operatorname{cond}(A) \left(\frac{\|\Delta A\|}{\|A\|} + \frac{\|\Delta b\|}{\|b\|} \right)}{1 - \operatorname{cond}(A) \frac{\|\Delta A\|}{\|A\|}} . \tag{12.21}$$

This theorem implies the following „thumb rule".

Thumb rule. *Assume that* $Ax = b$. *If the entries of* A *and* b *are accurate to about* s *decimal places and* $\operatorname{cond}(A) \sim 10^t$, *where* $t < s$, *then the entries of the computed solution are accurate to about* $s - t$ *decimal places.*

The assumption $\operatorname{cond}(A) \|\Delta A\| / \|A\| < 1$ of Theorem 12.16 guarantees that that matrix $A + \Delta A$ is nonsingular. The inequality $\operatorname{cond}(A) \|\Delta A\| / \|A\| < 1$ is equivalent with the inequality $\|\Delta A\| < \frac{1}{\|A^{-1}\|}$ and the distance of A from the nearest singular matrix is just $1/\left\|A^{-1}\right\|$. Thus we can give a new characterization of the condition number:

$$\frac{1}{\operatorname{cond}(A)} = \min_{A + \Delta A \text{ is singular}} \frac{\|\Delta A\|}{\|A\|} . \tag{12.22}$$

Thus if a matrix is ill-conditioned, then it is close to a singular matrix. Earlier we defined the condition numbers of matrices as the condition number of the mapping $F(x) = A^{-1}x$.

Let us introduce the following definition.

Definition 12.17 *A linear system solver is said to be* ***weakly stable*** *on a matrix class H, if for all well-conditioned $A \in H$ and for all b, the computed solution $\widehat{x}$ of the linear system $Ax = b$ has small relative error $\|\hat{x} - x\| / \|x\|$.*

Putting together Theorems 12.13–12.16 we obtain the following.

Theorem 12.18 (Bunch). *A linear system solver is weakly stable on a matrix class H, if for all well-conditioned $A \in H$ and for all b, the computed solution $\widehat{x}$ of the linear system $Ax = b$ satisfies any of the following conditions:*
(1) $\|\hat{x} - x\| / \|x\|$ is small;
(2) $\|r(\hat{x})\| / \|b\|$ is small;
(3) There exists ΔA such that $(A + \Delta A)\hat{x} = b$ and $\|\Delta A\| / \|A\|$ are small.

The estimate of Theorem 12.16 can be used in practice if we know estimates of $\Delta b, \Delta A$ and $\mathrm{cond}(A)$. If no estimates are available, then we can only make a posteriori error estimates.

In the following we study the componentwise error estimates. We first give an estimate for the absolute error of the approximate solution using the components of the inverse error.

Theorem 12.19 (Bauer, Skeel). *Let $A \in \mathbb{R}^n$ be nonsingular and assume that the approximate solution $\widehat{x}$ of $Ax = b$ satisfies the linear system $(A + E)\widehat{x} = b + e$. If $S \in \mathbb{R}^{n\times n}$, $s \in \mathbb{R}^n$ and $\varepsilon > 0$ are such that $S \geq 0$, $s \geq 0$, $|E| \leq \varepsilon S$, $|e| \leq \varepsilon s$ and $\varepsilon \left\| \left|A^{-1}\right| S \right\|_\infty < 1$, then*

$$\left\|\widehat{x} - x\right\|_\infty \leq \frac{\varepsilon \left\| \left|A^{-1}\right| (S|x| + s) \right\|_\infty}{1 - \varepsilon \left\| |A^{-1}| S \right\|_\infty}. \tag{12.23}$$

If $e = 0$ $(s = 0)$, $S = |A|$ and

$$k_r(A) = \left\| \left|A^{-1}\right| |A| \right\|_\infty < 1, \tag{12.24}$$

then we obtain the estimate

$$\left\|\widehat{x} - x\right\|_\infty \leq \frac{\varepsilon k_r(A)}{1 - \varepsilon k_r(A)}. \tag{12.25}$$

The quantity $k_r(A)$ is said to be ***Skeel-norm*** , although it is not a norm in the earlier defined sense. The Skeel-norm satisfies the inequality

$$k_r(A) \leq \mathrm{cond}_\infty(A) = \|A\|_\infty \left\|A^{-1}\right\|_\infty. \tag{12.26}$$

Therefore the above estimate is not worse than the traditional one that uses the standard condition number.

The inverse error can be estimated componentwise by the following result of Oettli and Prager. Let $A, \delta A \in \mathbb{R}^{n\times n}$ and $b, \delta b \in \mathbb{R}^n$. Assume that $\delta A \geq 0$ and $\delta b \geq 0$. Furthermore let

$$D = \left\{ \Delta A \in \mathbb{R}^{n\times n} : |\Delta A| \leq \delta A \right\}, \quad G = \left\{ \Delta b \in \mathbb{R}^n : |\Delta b| \leq \delta b \right\}.$$

Theorem 12.20 (Oettli, Prager). *The computed solution $\hat{x}$ satisfies a perturbed equation $(A+\Delta A)\,\hat{x}=b+\Delta b$ with $\Delta A\in D$ and $\Delta b\in G$, if*

$$|r\,(\hat{x})|=|A\hat{x}-b|\leq \delta A\,|\hat{x}|+\delta b. \tag{12.27}$$

We do not need the condition number to apply this theorem. In practice the entries δA and δb are proportional to the machine epsilon.

Theorem 12.21 (Wilkinson). *The approximate solution $\widehat{x}$ of $Ax=b$ obtained by the Gauss method in floating point arithmetic satisfies the perturbed linear equation*

$$(A+\Delta A)\,\widehat{x}=b \tag{12.28}$$

with

$$\|\Delta A\|_{\infty}\leq 8n^{3}\rho_{n}\,\|A\|_{\infty}\,u+O(u^{2}), \tag{12.29}$$

where ρ_n denotes the groth factor of the pivot elements and u is the unit roundoff.

Since ρ_n is small in practice, the relative error

$$\frac{\|\Delta A\|_{\infty}}{\|A\|_{\infty}}\leq 8n^{3}\rho_{n}u+O(u^{2})$$

is also small. Therefore Theorem12.18 implies that the Gauss method is weakly stable both for full and partial pivoting.

Wilkinson's theorem implies that

$$\mathrm{cond}_{\infty}(A)\frac{\|\Delta A\|_{\infty}}{\|A\|_{\infty}}\leq 8n^{3}\rho_{n}\mathrm{cond}_{\infty}\,(A)\,u+O\left(u^{2}\right)\ .$$

For a small condition number we can assume that $1-\mathrm{cond}_{\infty}(A)\,\|\Delta A\|_{\infty}/\,\|A\|_{\infty}\approx 1$. Using Theorems 12.21 and 12.16 (case $\Delta b=0$) we obtain the following estimate of the direct error:

$$\frac{\|\Delta x\|_{\infty}}{\|x\|_{\infty}}\leq 8n^{3}\rho_{n}\mathrm{cond}_{\infty}\,(A)\,u. \tag{12.30}$$

The obtained result supports the thumb rule in the case of the Gauss method.

Example 12.10 Consider the following linear system whose coefficients can be represented exactly:

$$\begin{aligned}888445x_{1}+887112x_{2}&=1,\\ 887112x_{1}+885781x_{2}&=0.\end{aligned}$$

Here $\mathrm{cond}(A)_{\infty}$ is big, but $\mathrm{cond}_{\infty}(A)\|\Delta A\|_{\infty}/\|A\|_{\infty}$ is negligible. The exact solution of the problem is $x_1=885781$, $x_2=-887112$. The MATLAB gives the approximate solution $\hat{x}_1=885827.23$, $\hat{x}_2=-887158.30$ with the relative error

$$\frac{\|x-\hat{x}\|_{\infty}}{\|x\|_{\infty}}=5.22\times 10^{-5}.$$

Since $s\approx 16$ and $\mathrm{cond}\,(A)_{\infty}\approx 3.15\times 10^{12}$, the result essentially corresponds to the Wilkinson theorem or the thumb rule. The Wilkinson theorem gives the bound

$$\|\Delta A\|_{\infty}\leq 1.26\times 10^{-8}$$

for the inverse error. If we use the Oettli-Prager theorem with the choice $\delta A=\epsilon_{M}\,|A|$ and $\delta b=\epsilon_{M}\,|b|$, then we obtain the estimate $|r\,(\hat{x})|\leq \delta A\,|\hat{x}|+\delta b$. Since $\||\delta A|\|_{\infty}=3.94\times 10^{-10}$, this estimate is better than that of Wilkinson.

Scaling and preconditioning Several matrices that occur in applications are ill-conditioned if their order n is large. For example the famous Hilbert-matrix

$$H_n = \left[\frac{1}{i+j-1}\right]_{i,j=1}^{n} \tag{12.31}$$

has $\mathrm{cond}_2(H_n) \approx e^{3.5n}$, if $n \to \infty$. There exist $2n \times 2n$ matrices with integer entries that can be represented exactly in standard IEEE754 floating point arithmetic while their condition number is approximately 4×10^{32n}.

We have two main techniques to solve linear systems with large condition numbers. Either we use multiple precision arithmetic or decrease the condition number. There are two known forms of decreasing the condition number.

1. Scaling We replace the linear system $Ax = b$ with the equation

$$(RAC)\, y = (Rb)\,, \tag{12.32}$$

where R and C are diagonal matrices.

We apply the Gauss method to this scaled system and get the solution y. The quantity $x = Cy$ defines the requested solution. If the condition number of the matrix RAC is smaller then we expect a smaller error in y and consequently in x. Various strategies are given to choose the scaling matrices R and C. One of the best known strategies is the ***balancing*** which forces every column and row of RAC to have approximately the same norm. For example, if

$$\hat{D} = \mathrm{diag}\left(\frac{1}{\|a_1^T\|_2}, \ldots, \frac{1}{\|a_n^T\|_2}\right)$$

where a_i^T is the i^{th} row vector of A, the Euclidean norms of the rows of $\hat{D}A$ will be 1 and the estimate

$$\mathrm{cond}_2\left(\hat{D}A\right) \le \sqrt{n} \min_{D \in D_+} \mathrm{cond}_2(DA)$$

holds with $D_+ = \{\mathrm{diag}(d_1, \ldots, d_n) \mid d_1, \ldots, d_n > 0\}$. This means that $\hat{D}$ optimally scales the rows of A in an approximate sense.

The next example shows that the scaling may lead to bad results.

Example 12.11 Consider the matrix

$$A = \begin{bmatrix} \epsilon/2 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

for $0 < \epsilon \ll 1$. It is easy to show that $\mathrm{cond}_\infty(A) = 12$. Let

$$R = C = \begin{bmatrix} 2/\sqrt{\epsilon} & 0 & 0 \\ 0 & \sqrt{\epsilon}/2 & 0 \\ 0 & 0 & \sqrt{\epsilon}/2 \end{bmatrix}.$$

Then the scaled matrix

$$RAR = \begin{bmatrix} 2 & 1 & 1 \\ 1 & \epsilon/4 & \epsilon/4 \\ 1 & \epsilon/4 & \epsilon/2 \end{bmatrix},$$

has the condition number $\mathrm{cond}_\infty(RAR) = 32/\epsilon$, which a very large value for small ϵ.

2. Preconditioning The preconditioning is very close to scaling. We rewrite the linear system $Ax = b$ with the equivalent form

$$\widetilde{A}x = (MA)\,x = Mb = \tilde{b}, \tag{12.33}$$

where matrix M is such that $\mathrm{cond}\left(M^{-1}A\right)$ is smaller and $Mz = y$ is easily solvable.

The preconditioning is often used with iterative methods on linear systems with symmetric and positive definite matrices.

A posteriori error estimates The a posteriori estimate of the error of an approximate solution is necessary to get some information on the reliability of the obtained result. There are plenty of such estimates. Here we show three estimates whose computational cost is $\Theta\left(n^2\right)$ flops. This cost is acceptable when comparing to the cost of direct or iterative methods ($\Theta\left(n^3\right)$ or $\Theta\left(n^2\right)$ per iteration step).

The estimate of the direct error with the residual error

Theorem 12.22 (Auchmuty). *Let $\widehat{x}$ be the approximate solution of $Ax = b$. Then*

$$\|x - \hat{x}\|_2 = \frac{c\,\|r(\hat{x})\|_2^2}{\|A^T r(\hat{x})\|_2}\,,$$

where $c \geq 1$.

The error constant c depends on A and the direction of error vector $\hat{x} - x$. Furthermore

$$\frac{1}{2}\mathrm{cond}_2(A) \approx C_2(A) = \frac{1}{2}\left(\mathrm{cond}_2(A) + \frac{1}{\mathrm{cond}_2(A)}\right) \leq \mathrm{cond}_2(A)\ .$$

The error constant c takes the upper value $C_2(A)$ only in exceptional cases. The computational experiments indicate that the average value of c grows slowly with the order of A and it depends more strongly on n than the condition number of A. The following experimental estimate

$$\|x - \widehat{x}\|_2 \lessapprox 0.5 \dim(A)\,\|r(\widehat{x})\|_2^2 \,/\, \left\|A^T r(\widehat{x})\right\|_2 \tag{12.34}$$

seems to hold with a high degree of probability.

The LINPACK estimate of $\left\|A^{-1}\right\|$ The famous LINPACK program package uses the following process to estimate $\left\|A^{-1}\right\|$. We solve the linear systems $A^T y = d$ and $Aw = y$. Then the estimate of $\left\|A^{-1}\right\|$ is given by

$$\left\|A^{-1}\right\| \approx \frac{\|w\|}{\|y\|} \quad \left(\leq \left\|A^{-1}\right\|\right). \tag{12.35}$$

Since

$$\frac{\|w\|}{\|y\|} = \frac{\left\|A^{-1}\left(A^{-T}d\right)\right\|}{\|A^{-T}d\|}\,,$$

we can interpret the process as an application of the power method of the eigenvalue problem. The estimate can be used with the $1-, 2-$ and ∞-norms. The entries of vector d are ± 1 possibly with random signs.

If the linear system $Ax = b$ is solved by the LU-method, then the solution of further linear systems costs $\Theta(n^2)$ flops per system. Thus the total cost of the LINPACK estimate remains small. Having the estimate $\left\|A^{-1}\right\|$ we can easily estimate $\mathrm{cond}(A)$ and the error of the approximate solution (cf. Theorem 12.16 or the thumb rule). We remark that several similar processes are known in the literature.

The Oettli-Prager estimate of the inverse error We use the Oettli-Prager theorem in the following form. Let $r(\hat{x}) = A\hat{x} - b$ be the residual error, $E \in \mathbb{R}^{n\times n}$ and $f \in \mathbb{R}^n$ are given such that $E \geq 0$ and $f \geq 0$. Let

$$\omega = \max_i \frac{|r(\hat{x})_i|}{(E|\hat{x}| + f)_i} ,$$

where $0/0$ is set to 0-nak, $\rho/0$ is set to ∞, if $\rho \neq 0$. Symbol $(y)_i$ denotes the i^{th} component of the vector y. If $\omega \neq \infty$, then there exist a matrix ΔA and a vector Δb for which

$$|\Delta A| \leq \omega E, \quad |\Delta b| \leq \omega f$$

holds and

$$(A + \Delta A)\hat{x} = b + \Delta b .$$

Moreover ω is the smallest number for which ΔA and Δb exist with the above properties. The quantity ω measures the relative inverse error in terms of E and f. If for a given E, f and $\hat{x}$, the quantity ω is small, then the perturbed problem (and its solution) are close to the original problem (and its solution). In practice, the choice $E = |A|$ and $f = |b|$ is preferred

Iterative refinement Denote by $\hat{x}$ the approximate solution of $Ax = b$ and let $r(y) = Ay - b$ be the residual error at the point y. The precision of the approximate solution $\hat{x}$ can be improved with the following method.

ITERATIVE-REFINEMENT$(A, b, \hat{x}, tol)$

```
1  k ← 1
2  x_1 ← x̂
3  d̂ ← inf
4  while = ‖d̂‖ / ‖x_k‖ > tol
5      do r ← Ax_k − b
6         Compute the approximate solution d̂ of Ad = r with the LU-method.
7         x_{k+1} ← x_k − d̂
8         k ← k + 1
9  return x_k
```

There are other variants of this process. We can use other linear solvers instead of the LU-method.

Let η be the smallest bound of relative inverse error with

$$(A + \Delta A)\,\hat{x} = b + \Delta b, \quad |\Delta A| \le \eta\,|A|\,, \quad |\Delta b| \le \eta\,|b| \ .$$

Furthermore let

$$\sigma\,(A, x) = \max_k \left(|A|\,|x|\right)_k \Big/ \min_k \left(|A|\,|x|\right)_k, \quad \min_k \left(|A|\,|x|\right)_k > 0\ .$$

Theorem 12.23 (Skeel). *If $k_r\left(A^{-1}\right)\sigma\,(A, x) \le c_1 < 1/\epsilon_M$, then for sufficiently large k we have*

$$(A + \Delta A)\,x_k = b + \Delta b, \quad |\Delta A| \le 4\eta\epsilon_M\,|A|\,, \quad |\Delta b| \le 4\eta\epsilon_M\,|b| \ . \tag{12.36}$$

This result often holds after the first iteration, i.e. for $k = 2$. Jankowski and Wozniakowski investigated the iterative refinement for any method ϕ which produces an approximate solution $\widehat{x}$ with relative error less than 1. They showed that the iterative refinement improves the precision of the approximate solution even in single precision arithmetic and makes method ϕ to be weakly stable.

Exercises

12.2-1 Prove Theorem 12.7.

12.2-2 Consider the linear systems $Ax = b$ and $Bx = b$, where

$$A = \begin{bmatrix} 1 & 1/2 \\ 1/2 & 1/3 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & -1/2 \\ 1/2 & 1/3 \end{bmatrix}$$

and $b \in \mathbb{R}^2$. Which equation is more sensitive to the perturbation of b? What should be the *relative error* of b in the more sensitive equation in order to get the solutions of both equations with the same precision?

12.2-3 Let $\chi = 3/2^{29}$, $\zeta = 2^{14}$ and

$$A = \begin{bmatrix} \chi\zeta & -\zeta & \zeta \\ \zeta^{-1} & \zeta^{-1} & 0 \\ \zeta^{-1} & -\chi\zeta^{-1} & \zeta^{-1} \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1+\varepsilon \\ 1 \end{bmatrix}\ .$$

Solve the linear systems $Ax = b$ for $\varepsilon = 10^{-1}, 10^{-3}, 10^{-5}, 10^{-7}, 10^{-10}$. Explain the results.

12.2-4 Let A be a 10×10 matrix and choose the band matrix consisting of the main and the neighbouring two subdiagonals of A as a preconditioning matrix. How much does the *condition number* of A improves if (i) A is a random matrix; (ii) A is a Hilbert matrix?

12.2-5 Let

$$A = \begin{bmatrix} 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \\ 1/4 & 1/5 & 1/6 \end{bmatrix},$$

and assume that ε is the common error bound of every component of $b \in \mathbb{R}^3$. Give the sharpest possible error bounds for the solution $[x_1, x_2, x_3]^T$ of the equation $Ax = b$ and for the sum $(x_1 + x_2 + x_3)$.

12.2-6 Consider the linear system $Ax = b$ with the approximate solution $\hat{x}$.
(i) Give an error bound for $\hat{x}$, if $(A+E)\hat{x} = b$ holds exactly and both A and $A+E$ is nonsingular.
(ii) Let

$$A = \begin{bmatrix} 10 & 7 & 8 \\ 7 & 5 & 6 \\ 8 & 6 & 10 \end{bmatrix}, \quad b = \begin{bmatrix} 25 \\ 18 \\ 24 \end{bmatrix}$$

and consider the solution of $Ax = b$. Give (if possible) a relative error bound for the entries of A such that the integer part of every solution component remains constant within the range of this relative error bound.

12.3. Eigenvalue problems

The set of complex n-vectors will be denoted by $\mathbb{C}^n$. Similarly, $\mathbb{C}^{m\times n}$ denotes the set of *complex $m \times n$ matrices*.

Definition 12.24 *Let $A \in \mathbb{C}^{n\times n}$ be an arbitrary matrix. The number $\lambda \in \mathbb{C}$ is the* ***eigenvalue*** *of A if there is vector $x \in \mathbb{C}^n$ $(x \neq 0)$ such that*

$$Ax = \lambda x. \tag{12.37}$$

Vector x is called the (right) eigenvector of A that belongs to the eigenvalue λ.

Equation $Ax = \lambda x$ can be written in the equivalent form $(A - \lambda I)x = 0$, where I is the unit matrix of appropriate size. The latter homogeneous linear system has a nonzero solution x if and only if

$$\phi(\lambda) = \det(A - \lambda I) = \det\left(\begin{bmatrix} a_{11}-\lambda & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22}-\lambda & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn}-\lambda \end{bmatrix}\right) = 0\ . \tag{12.38}$$

Equation (12.38) is called the ***characteristic equation*** of matrix A. The roots of this equation are the eigenvalues of matrix A. Expanding $\det(A - \lambda I)$ we obtain a polynomial of degree n:

$$\phi(\lambda) = (-1)^n(\lambda^n - p_1\lambda^{n-1} - \ldots - p_{n-1}\lambda - p_n)\ .$$

This polynomial called the ***characteristic polynomial*** of A. It follows from the fundamental theorem of algebra that any matrix $A \in \mathbb{C}^{n\times n}$ has exactly n eigenvalues with multiplicities. The eigenvalues may be complex or real. Therefore one needs to use complex arithmetic for eigenvalue calculations. If the matrix is real and the computations are done in real arithmetic, the complex eigenvalues and eigenvectors can be determined only with special techniques.

If $x \neq 0$ is an eigenvector, $t \in \mathbb{C}$ $(t \neq 0)$, then tx is also eigenvector. The number of linearly independent eigenvectors that belong to an eigenvalue λ_k does not exceed

the multiplicity of λ_k in the characteristic equation (12.38). The eigenvectors that belong to different eigenvalues are linearly independent.

The following results give estimates for the size and location of the eigenvalues.

Theorem 12.25 *Let λ be any eigenvalue of matrix A. The upper estimate $|\lambda| \leq \|A\|$ holds in any induced matrix norm.*

Theorem 12.26 (Gersgorin). *Let $A \in \mathbb{C}^{n\times n}$,*

$$r_i = \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad (i = 1, \ldots, n)$$

and

$$D_i = \{z \in \mathbf{C} \,|\, |z - a_{ii}| \leq r_i\} \quad (i = 1, \ldots, n)\,.$$

Then for any eigenvalue λ of A we have $\lambda \in \cup_{i=1}^{n} D_i$.

For certain matrices the solution of the characteristic equation (12.38) is very easy. For example, if A is a triangular matrix, then its eigenvalues are entries of the main diagonal. In most cases however the computation of all eigenvalues and eigenvectors is a very difficult task. Those transformations of matrices that keeps the eigenvalues unchanged have practical significance for this problem. Later we see that the eigenvalue problem of transformed matrices is simpler.

Definition 12.27 *The matrices $A, B \in \mathbb{C}^{n\times n}$ are similar if there is a matrix T such that $B = T^{-1}AT$. The mapping $A \to T^{-1}AT$ is said to be* ***similarity transformation*** *of A.*

Theorem 12.28 *Assume that* $\det(T) \neq 0$. *Then the eigenvalues of A and $B = T^{-1}AT$ are the same. If x is the eigenvector of A, then $y = T^{-1}x$ is the eigenvector of B.*

Similar matrices have the same eigenvalues.

The difficulty of the eigenvalue problem also stems from the fact that the eigenvalues and eigenvectors are very sensitive (unstable) to changes in the matrix entries. The eigenvalues of A and the perturbed matrix $A+\delta A$ may differ from each other significantly. Besides the multiplicity of the eigenvalues may also change under perturbation. The following theorems and examples show the very sensitivity of the eigenvalue problem.

Theorem 12.29 (Ostrowski, Elsner). *For every eigenvalue λ_i of matrix $A \in \mathbb{C}^{n\times n}$ there exists an eigenvalue μ_k of the perturbed matrix $A + \delta A$ such that*

$$|\lambda_i - \mu_k| \leq (2n-1)\left(\|A\|_2 + \|A + \delta A\|_2\right)^{1-\frac{1}{n}} \|\delta A\|_2^{\frac{1}{n}}\,.$$

We can observe that the eigenvalues are changing continuously and the size of change is proportional to the n^{th} root of $\|\delta A\|_2$.

Example 12.12 Consider the following perturbed *Jordan matrix* of the size $r \times r$:

$$\begin{bmatrix} \mu & 1 & 0 & \dots & 0 \\ 0 & \mu & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & & \ddots & \mu & 1 \\ \epsilon & 0 & \dots & 0 & \mu \end{bmatrix}.$$

The characteristic equation is $(\lambda - \mu)^r = \epsilon$, which gives the r different eigenvalues

$$\lambda_s = \mu + \epsilon^{1/r} \left(\cos\left(2s\pi/r\right) + i \sin\left(2s\pi/r\right)\right) \quad (s = 0, \dots, r-1)$$

instead of the original eigenvalue μ with multiplicity r. The size of change is $\epsilon^{1/r}$, which corresponds to Theorem (12.29). If $|\mu| \approx 1$, $r = 16$ and $\epsilon = \epsilon_M \approx 2.2204 \times 10^{-16}$, then the perturbation size of the eigenvalues is ≈ 0.1051. This is a significant change relative to the input perturbation ϵ.

For special matrices and perturbations we may have much better perturbation bounds.

Theorem 12.30 (Bauer, Fike). *Assume that $A \in \mathbb{C}^{n \times n}$ is **diagonalisable** , that is a matrix X exists such that $X^{-1}AX = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$. Denote μ an eigenvalue of $A + \delta A$. Then*

$$\min_{1 \le i \le n} |\lambda_i - \mu| \le \operatorname{cond}_2(X) \, \|\delta A\|_2 . \tag{12.39}$$

This result is better than that of Ostrowski and Elsner. Nevertheless $\operatorname{cond}_2(X)$, which is generally unknown, can be very big.

The eigenvalues are continuous functions of the matrix entries. This is also true for the normalized eigenvectors if the eigenvalues are simple. The following example shows that this property does not hold for multiple eigenvalues.

Example 12.13 Let

$$A(t) = \begin{matrix} 1 + t\cos(2/t) & -t\sin(2/t) \\ -t\sin(2/t) & 1 - t\cos(2/t) \end{matrix} \qquad (t \neq 0).$$

The eigenvalues of $A(t)$ are $\lambda_1 = 1 + t$ and $\lambda_2 = 1 - t$. Vector $[\sin(1/t), \cos(1/t)]^T$ is the eigenvector belonging to λ_1. Vector $[\cos(1/t), -\sin(1/t)]^T$ is the eigenvector belonging to λ_2. If $t \to 0$, then

$$A(t) \to I = \begin{matrix} 1 & 0 \\ 0 & 1 \end{matrix}, \quad \lambda_1, \lambda_2 \to 1 ,$$

while the eigenvectors do not have limit.

We study the numerical solution of the eigenvalue problem in the next section. Unfortunately it is very difficult to estimate the goodness of numerical approximations. From the fact that $Ax - \lambda x = 0$ holds with a certain error we cannot conclude anything in general.

Example 12.14 Consider the matrix

$$A(\epsilon) = \begin{matrix} 1 & 1 \\ \epsilon & 1 \end{matrix} ,$$

where $\epsilon \approx 0$ is small. The eigenvalues of $A(\epsilon)$ are $1 \pm \sqrt{\epsilon}$, while the corresponding eigenvectors are $[1, \pm\sqrt{\epsilon}]^T$. Let $\mu = 1$ be an approximation of the eigenvalues and let $x = [1, 0]^T$ be the approximate eigenvector. Then

$$\|Ax - \mu x\|_2 = \begin{matrix} 0 \\ \epsilon \end{matrix}_2 = \epsilon .$$

If $\epsilon = 10^{-10}$, then the residual error under estimate the true error 10^{-5} by five order.

Remark 12.31 *We can define the* ***condition number of eigenvalues*** *for simple eigenvalues:*

$$\nu(\lambda_1) \approx \frac{\|x\|_2 \|y\|_2}{|x^H y|} ,$$

where x and y are the right and left eigenvectors, respectively. For multiple eigenvalues the condition number is not finite.

12.3.1. Iterative solutions of the eigenvalue problem

We investigate only the real eigenvalues and eigenvectors of real matrices. The methods under consideration can be extended to the complex case with appropriate modifications.

The power method This method is due to von Mieses. Assume that $A \in \mathbb{R}^{n \times n}$ has exactly n different real eigenvalues. Then the eigenvectors $x_1, \ldots, x_n$ belonging to the corresponding eigenvalues $\lambda_1, \ldots, \lambda_n$ are linearly independent. Assume that the eigenvalues satisfy the condition

$$|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$$

and let $v^{(0)} \in \mathbb{R}^n$ be a given vector. This vector is a unique linear combination of the eigenvectors, that is $v^{(0)} = \alpha_1 x_1 + \alpha_2 x_2 + \ldots + \alpha_n x_n$. Assume that $\alpha_1 \neq 0$ and compute the sequence $v^{(k)} = Av^{(k-1)} = A^k v^{(0)}$ $(k = 1, 2, \ldots)$. The initial assumptions imply that

$$\begin{aligned} v^{(k)} = Av^{(k-1)} &= A(\alpha_1 \lambda_1^{k-1} x_1 + \alpha_2 \lambda_2^{k-1} x_2 + \ldots + \alpha_n \lambda_n^{k-1} x_n) \\ &= \alpha_1 \lambda_1^k x_1 + \alpha_2 \lambda_2^k x_2 + \ldots + \alpha_n \lambda_n^k x_n \\ &= \lambda_1^k \left(\alpha_1 x_1 + \alpha_2 \left(\tfrac{\lambda_2}{\lambda_1}\right)^k x_2 + \ldots + \alpha_n \left(\tfrac{\lambda_n}{\lambda_1}\right)^k x_n \right) . \end{aligned}$$

Let $y \in \mathbb{R}^n$ be an arbitrary vector such that $y^T x_1 \neq 0$. Then

$$\frac{y^T A v^{(k)}}{y^T v^{(k)}} = \frac{y^T v^{(k+1)}}{y^T v^{(k)}} = \frac{\lambda_1^{k+1} \left(\alpha_1 y^T x_1 + \sum_{i=2}^n \alpha_i \left(\frac{\lambda_i}{\lambda_1}\right)^{k+1} y^T x_i \right)}{\lambda_1^k \left(\alpha_1 y^T x_1 + \sum_{i=2}^n \alpha_i \left(\frac{\lambda_i}{\lambda_1}\right)^k y^T x_i \right)} \to \lambda_1 .$$

Given the initial vector $v^{(0)} \in \mathbb{R}^n$, the power method has the following form.

POWER-METHOD$(A, v^{(0)})$

```
1  k ← 0
2  while exit condition = FALSE
3        do k ← k + 1
4           z^(k) ← A v^(k−1)
5           Select vector y such that y^T v^(k−1) ≠ 0
6           γ_k ← y^T z^(k) / y^T v^(k−1)
7           v^(k) ← z^(k) / ||z^(k)||_∞
8  return γ_k, v^(k)
```

It is clear that

$$v^{(k)} \to x_1, \quad \gamma_k \to \lambda_1 \ .$$

The convergence $v^{(k)} \to x_1$ here means that $\left(v^{(k)}, x_1\right)_{\measuredangle} \to 0$, that is the action line of $v^{(k)}$ tends to the action line of x_1. There are various strategies to select y. We can select $y = e_i$, where i is defined by $\left|v_i^{(k)}\right| = \left\|v^{(k)}\right\|_\infty$. If we select $y = v^{(k-1)}$, then $\gamma_k = v^{(k-1)T} A v^{(k-1)} / \left(v^{(k-1)T} v^{(k-1)}\right)$ will be identical with the *Rayleigh quotient* $\mathbf{R}\left(v^{(k-1)}\right)$. This choice gives an approximation of λ_1 that have the minimal residual norm (Example 12.14 shows that this choice is not necessarily the best option).

The speed of convergence depends on the quotient $|\lambda_2/\lambda_1|$. The method is very sensitive to the choice of the initial vector $v^{(0)}$. If $\alpha_1 = 0$, then the process does not converge to the dominant eigenvalue λ_1. For certain matrix classes the power method converges with probability 1 if the initial vector $v^{(0)}$ is randomly chosen. In case of complex eigenvalues or multiple λ_1 we have to use modifications of the algorithm. The speed of convergence can be accelerated if the method is applied to the shifted matrix $A - \sigma I$, where σ is an appropriately chosen number. The shifted matrix $A - \sigma I$ has the eigenvalues $\lambda_1 - \sigma, \lambda_2 - \sigma, \ldots, \lambda_n - \sigma$ and the corresponding convergence factor $|\lambda_2 - \sigma| \,/\, |\lambda_1 - \sigma|$. The latter quotient can be made smaller than $|\lambda_2/\lambda_1|$ with the proper selection of σ.

The usual exit condition of the power method is

$$\|E_k\|_2 = \frac{\|r_k\|_2}{\left\|v^{(k)}\right\|_2} = \frac{\left\|Av^{(k)} - \gamma_k v^{(k)}\right\|_2}{\left\|v^{(k)}\right\|_2} \le \epsilon \ .$$

If we simultaneously apply the power method to the transposed matrix A^T and $w_k = \left(A^T\right)^k w_0$, then the quantity

$$\nu(\lambda_1) \approx \frac{\left\|w^{(k)}\right\|_2 \left\|v^{(k)}\right\|_2}{\left|w_k^T v_k\right|}$$

gives an estimate for the condition number of λ_1 (see Remark 12.31). In such a case we use the exit condition

$$\nu(\lambda_1) \|E_k\|_2 \le \epsilon \ .$$

The power method is very useful for large sparse matrices. It is often used to

determine the largest and the smallest eigenvalue. We can approximate the smallest eigenvalue as follows. The eigenvalues of A^{-1} are $1/\lambda_1, \ldots, 1/\lambda_n$. The eigenvalue $1/\lambda_n$ will be the eigenvalue with the largest modulus. We can approximate this value by applying the power method to A^{-1}. This requires only a small modification of the algorithm. We replace line 4. with the following:

$$\text{Solve equation } Az^{(k)} = v^{(k-1)} \text{ for } z^{(k)}$$

The modified algorithm is called the ***inverse power method***. It is clear that $\gamma_k \to 1/\lambda_n$ and $v^{(k)} \rightharpoonup x_n$ hold under appropriate conditions. If we use the LU-method to solve $Az^{(k)} = v^{(k-1)}$, we can avoid the inversion of A.

If the inverse power method is applied to the shifted matrix $A - \mu I$, then the eigenvalues of $(A - \mu I)^{-1}$ are $(\lambda_i - \mu)^{-1}$. If μ approaches, say, to λ_t, then $\lambda_i - \mu \to \lambda_i - \lambda_t$. Hence the inequality

$$|\lambda_t - \mu|^{-1} > |\lambda_i - \mu|^{-1} \quad (i \neq t)$$

holds for the eigenvalues of the shifted matrix. The speed of convergence is determined by the quotient

$$q = |\lambda_t - \mu| \,/\, \{\max |\lambda_i - \mu|\} \ .$$

If μ is close enough to λ_t, then q is very small and the inverse power iteration converges very fast. This property can be exploited in the calculation of approximate eigenvectors if an approximate eigenvalue, say μ, is known. Assuming that $\det(A - \mu I) \neq 0$, we apply the inverse power method to the shifted matrix $A - \mu I$. In spite of the fact that matrix $A - \mu I$ is nearly singular and the linear equation $(A - \mu I)\, z^{(k)} = v^{(k)}$ cannot be solved with high precision, the algorithm gives very often good approximations of the eigenvectors.

Finally we note that in principle the von Mieses method can be modified to determine all eigenvalues and eigenvectors.

Orthogonalisation processes We need the following definition and theorem.

Definition 12.32 *The matrix $Q \in \mathbb{R}^{n\times n}$ is said to be* ***orthogonal*** *if $Q^T Q = I$.*

Theorem 12.33 (QR-decomposition). *Every matrix $A \in \mathbb{R}^{n\times m}$ having linearly independent column vectors can be decomposed in the product form $A = QR$, where Q is orthogonal and R is upper triangular matrix.*

We note that the QR-decomposition can be applied for solving linear systems of equations, similarly to the LU-decomposition. If the QR-decomposition of A is known, then the equation $Ax = QRx = b$ can be written in the equivalent form $Rx = Q^T b$. Thus we have to solve only an upper triangular linear system.

There are several methods to determine the QR-decomposition of a matrix. In practice the Givens-, the Householder- and the MGS-methods are used.

The MGS (Modified Gram-Schmidt) method is a stabilised, but algebraically equivalent version of the classical Gram-Schmidt orthogonalisation algorithm. The basic problem is the following: We seek for an *orthonormal basis* $\{q_j\}_{j=1}^m$ of the

subspace

$$\mathcal{L}\{a_1,\ldots,a_m\} = \left\{ \sum_{j=1}^{m} \lambda_j a_j \mid \lambda_j \in \mathbb{R},\ j=1,\ldots,m \right\} ,$$

where $a_1,\ldots,a_m \in \mathbb{R}^n$ $(m \le n)$ are linearly independent vectors. That is we determine the linearly independent vectors $q_1,\ldots,q_m$ such that

$$q_i^T q_j = 0 \ \ (i \ne j)\,, \|q_i\|_2 = 1 \ \ (i=1,\ldots,m)$$

and

$$\mathcal{L}\{a_1,\ldots,a_m\} = \mathcal{L}\{q_1,\ldots,q_m\} \ .$$

The basic idea of the *classical Gram-Schmidt-method* is the following:

Let $r_{11} = \|a_1\|_2$ and $q_1 = a_1/r_{11}$. Assume that vectors $q_1,\ldots,q_{k-1}$ are already computed and orthonormal. Assume that vector $\tilde{q}_k = a_k - \sum_{j=1}^{k-1} r_{jk} q_j$ is such that $\tilde{q}_k \perp q_i$, that is $\tilde{q}_k^T q_i = a_k^T q_i - \sum_{j=1}^{k-1} r_{jk} q_j^T q_i = 0$ holds for $i = 1,\ldots,k-1$. Since $q_1,\ldots,q_{k-1}$ are orthonormal, $q_j^T q_i = 0$ $(i \ne j)$ and $r_{ik} = a_k^T q_i \quad (i=1,\ldots,k-1)$. After normalisation we obtain $q_k = \tilde{q}_k / \|\tilde{q}_k\|_2$.

The algorithm is formalised as follows.

CGS-Orthogonalization$(m, a_1,\ldots,a_m)$

```
1  for k ← 1 to m
2      do for i ← 1 to k − 1
3             do r_ik ← a_k^T a_i
4                a_k ← a_k − r_ik a_i
5         r_kk ← ‖a_k‖_2
6         a_k ← a_k / r_kk
7  return a_1, ..., a_m
```

The algorithm overwrites vectors a_i by the orthonormal vectors q_i. The connection with the QR-decomposition follows from the relation $a_k = \sum_{j=1}^{k-1} r_{jk} q_j + r_{kk} q_k$. Since

$$\begin{aligned} a_1 &= q_1 r_{11}, \\ a_2 &= q_1 r_{12} + q_2 r_{22}, \\ a_3 &= q_1 r_{13} + q_2 r_{23} + q_3 r_{33}, \\ &\vdots \\ a_m &= q_1 r_{1m} + q_2 r_{2m} + \ldots + q_m r_{mm} \ , \end{aligned}$$

we can write that

$$A = [a_1,\ldots,a_m] = \underbrace{[q_1,\ldots,q_m]}_{Q} \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & \ldots & r_{1m} \\ 0 & r_{22} & r_{23} & \ldots & r_{2m} \\ 0 & 0 & r_{33} & \ldots & r_{3m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & r_{mm} \end{bmatrix}}_{R} = QR \ .$$

The *numerically stable MGS method* is given in the following form

MGS-ORTHOGONALISATION$(m, a_1, \ldots, a_m)$

```
1  for k ← 1 to m
2      do r_kk ← ‖a_k‖_2
3         a_k ← a_k / r_kk
4         for j ← k + 1 to m
5             do r_kj ← a_j^T a_k
6                a_j ← a_j − r_kj a_k
7  return a_1, ..., a_m
```

The algorithm overwrites vectors a_i by the orthonormal vectors q_i. The MGS method is more stable than the CGS algorithm. Björck proved that for $m = n$ the computed matrix $\hat{Q}$ satisfies

$$\hat{Q}^T\hat{Q} = I + E, \qquad \|E\|_2 \cong \operatorname{cond}(A)\, u\ ,$$

where u is the unit roundoff.

The QR-method Today the QR-method is the most important numerical algorithm to compute all eigenvalues of a general matrix. It can be shown that the QR-method is a generalisation of the power method. The basic idea of the method is the following: Starting from $A_1 = A$ we compute the sequence $A_{k+1} = Q_k^{-1}A_kQ_k = Q_k^TA_kQ_k$, where Q_k is orthogonal, A_{k+1} is orthogonally similar to A_k (A) and the lower triangular part of A_k tends to a diagonal matrix, whose entries will be the eigenvalues of A. Here Q_k is the orthogonal factor of the QR-decomposition $A_k = Q_kR_k$. Therefore $A_{k+1} = Q_k^T(Q_kR_k)Q_k = R_kQ_k$. The basic algorithm is given in the following form.

QR-METHOD(A)

```
1  k ← 1
2  A_1 ← A
3  while exit condition = FALSE
4        do Compute the QR-decomposition A_k = Q_k R_k
5           A_{k+1} ← R_k Q_k
6           k ← k + 1
7  return A_k
```

The following result holds.

Theorem 12.34 (Parlett). *If the matrix A is diagonalisable, $X^{-1}AX = diag(\lambda_1, \lambda_2, \ldots, \lambda_n)$, the eigenvalues satisfy*

$$|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n| > 0$$

and X has an LU-decomposition, then the lower triangular part of A_k converges to a diagonal matrix whose entries are the eigenvalues of A.

In general, matrices A_k do not necessarily converge to a given matrix. If A has p eigenvalues of the same modulus, the form of matrices A_k converge to the form

$$\begin{bmatrix} \times & & & & & & & & \times \\ 0 & \ddots & & & & & & & \\ & & \times & & & & & & \\ 0 & & 0 & * & \cdots & * & & & \\ & & & \vdots & & \vdots & & & \\ & & & * & \cdots & * & & & \\ 0 & & & & & 0 & \times & & \\ & & & & & & & \ddots & \\ 0 & & & & & & & 0 & \times \end{bmatrix}, \tag{12.40}$$

where the entries of the submatrix denoted by $*$ do not converge. However the eigenvalues of this submatrix will converge. This submatrix can be identified and properly handled. A real matrix may have real and complex eigenvalues. If there is a complex eigenvalues, than there is a corresponding conjugate eigenvalue as well. For pairs of complex conjugated eigenvalues p is at least 2. Hence the sequence A_k will show this phenomenon .

The QR-decomposition is very expensive. Its cost is $\Theta(n^3)$ flops for general $n \times n$ matrices. If A has upper Hessenberg form, the cost of QR-decomposition is $\Theta(n^2)$ flops.

Definition 12.35 *The matrix $A \in \mathbb{R}^{n\times n}$ has **upper Hessenberg form**, , if*

$$A = \begin{bmatrix} a_{11} & & & \cdots & & a_{1n} \\ a_{21} & & & & & \\ 0 & a_{32} & & & & \vdots \\ \vdots & 0 & \ddots & & & \\ & & \ddots & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & & \cdots & 0 & a_{n,n-1} & a_{nn} \end{bmatrix}.$$

The following theorem guarantees that if A has upper Hessenberg form, then every A_k of the QR-method has also upper Hessenberg form.

Theorem 12.36 *If A has upper Hessenberg form and $A = QR$, then RQ has also upper Hessenberg form.*

We can transform a matrix A to a similar matrix of upper Hessenberg form in many ways. One of the cheapest ways, that costs about $5/6n^3$ flops, is based on the Gauss elimination method. Considering the advantages of the upper Hessenberg form the efficient implementation of the QR-method requires first the similarity transformation of A to upper Hessenberg form.

The convergence of the QR-method, similarly to the power method, depends on the quotients $|\lambda_{i+1}/\lambda_i|$. The eigenvalues of the shifted matrix $A - \sigma I$ are $\lambda_1 -$

$\sigma, \lambda_2 - \sigma, \ldots, \lambda_n - \sigma$. The corresponding eigenvalue ratios are $|(\lambda_{i+1} - \sigma) / (\lambda_i - \sigma)|$. A proper selection of σ can fasten the convergence.

The usual form of the QR-method includes the transformation to upper Hessenberg form and the shifting.

SHIFTED-QR-METHOD(A)

1 $H_1 \leftarrow U^{-1}AU$ (H_1 is of upper Hessenberg form)
2 $k \leftarrow 1$
3 **while** *exit condition* = FALSE
4 **do** compute the QR-decomposition $H_k - \sigma_k I = Q_k R_k$
5 $H_{k+1} \leftarrow R_k Q_k + \sigma_k I$
6 $k \leftarrow k + 1$
7 **return** H_k

In practice the QR-method is used in shifted form. There are various strategies to select σ_i. The most often used selection is given by $\sigma_k = h_{nn}^{(k)}$ $\left(H_k = \left[h_{ij}^{(k)}\right]_{i,j=1}^{n}\right)$.

The eigenvectors of A can also be determined by the QR-method. For this we refer to the literature.

Exercises

12.3-1 Apply the *power method* to the matrix $A = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$ with the initial vector $v^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. What is the result of the 20th step?

12.3-2 Apply the *power method*, the *inverse power method* and the QR-method to the matrix

$$\begin{bmatrix} -4 & -3 & -7 \\ 2 & 3 & 2 \\ 4 & 2 & 7 \end{bmatrix}.$$

12.3-3 Apply the shifted QR-method to the matrix of the previous exercise with the choice $\sigma_i = \sigma$ (σ is fixed).

12.4. Numerical program libraries and software tools

We have plenty of devices and tools that support efficient coding and implementation of numerical algorithms. One aim of such developments is to free the programmers from writing the programs of frequently occurring problems. This is usually done by writing safe, reliable and standardised routines that can be downloaded from (public) program libraries. We just mention the LINPACK, EISPACK, LAPACK, VISUAL NUMERICS (former IMSL) and NAG libraries. Another way of developments is to produce software that work as a programming language and makes the programming very easy. Such software systems are the MATLAB and the SciLab.

12.4.1. Standard linear algebra subroutines

The main purpose of the BLAS (Basic Linear Algebra Subprograms) programs is the standardisation and efficient implementation the most frequent matrix-vector operations. Although the BLAS routines were published in FORTRAN they can be accessed in optimised machine code form as well. The BLAS routines have three levels:

- BLAS 1 (1979),
- BLAS 2 (1988),
- BLAS 3 (1989).

These levels corresponds to the computation cost of the implemented matrix operations. The BLAS routines are considered as the best implementations of the given matrix operations. The selection of the levels and individual BLAS routines strongly influence the efficiency of the program. A sparse version of BLAS also exists.

We note that the BLAS 3 routines were developed mainly for block parallel algorithms. The standard linear algebra packages LINPACK, EISPACK and LAPACK are built from BLAS routines. The parallel versions can be found in the SCALAPACK package. These programs can be found in the public NETLIB library:

`http:/www.netlib.org/index.html`

BLAS 1 routines Let $\alpha \in \mathbb{R}$, $x, y \in \mathbb{R}^n$. The BLAS 1 routines are the programs of the most important vector operations ($z = \alpha x$, $z = x + y$, $dot = x^T y$), the computation of $\|x\|_2$, the swapping of variables, rotations and the *saxpy* operation which is defined by

$$z = \alpha x + y \ .$$

The word *saxpy* means that „scalar alpha x plus y". The *saxpy* operation is implemented in the following way.

SAXPY(α, x, y)

```
1  n ← elements[x]
2  for i ← 1 to n
3       do z[i] = αx[i] + y[i]
4  return z
```

The *saxpy* is a software driven operation. The cost of BLAS 1 routines is $\Theta(n)$ flops.

BLAS 2 routines The matrix-vector operations of BLAS 2 requires $\Theta\left(n^2\right)$ flops. These operations are $y = \alpha Ax + \beta y$, $y = Ax$, $y = A^{-1}x$, $y = A^T x$, $A \leftarrow A + xy^T$ and their variants. Certain operations work only with triangular matrices. We analyse two operations in detail. The „outer or dyadic product" update

$$A \leftarrow A + xy^T \quad \left(A \in \mathbb{R}^{m \times n},\ x \in \mathbb{R}^m,\ y \in \mathbb{R}^n\right)$$

can be implemented in two ways.

The rowwise or „ij" variant:

OUTER-PRODUCT-UPDATE-VERSION „IJ" (A, x, y)

```
1  m ← rows[A]
2  for i ← 1 to m
3       do A[i, :] ← A[i, :] + x[i] y^T
4  return A
```

The notation „:" denotes all allowed indices. In our case this means the indices $1 \leq j \leq n$. Thus $A[i, :]$ denotes the i^{th} row of matrix A.

The columnwise or „*ji*" variant*:*

OUTER-PRODUCT-UPDATE-VERSION „JI"(A, x, y)

```
1  n ← columns[A]
2  for j ← 1 to n
3       do A[:, j] ← A[:, j] + y[j] x
4  return A
```

Here $A[:, j]$ denotes the j^{th} column of matrix A. Observe that both variants are based on the saxpy operation.

The *gaxpy* operation is defined by

$$z = y + Ax \quad \left(x \in \mathbb{R}^n,\ y \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}\right) .$$

The word *gaxpy* means that „general $A\,x$ plus y". The *gaxpy* operation is also software driven and implemented in the following way:

GAXPY(A, x, y)

```
1  n ← columns[A]
2  z ← y
3  for j ← 1 to n
4       do z ← z + x[j] A[:, j]
5  return z
```

Observe that the computation is done columnwise and the *gaxpy* operation is essentially a generalised *saxpy*.

BLAS 3 routines These routines are the implementations of $\Theta\left(n^3\right)$ matrix-matrix and matrix-vector operations such as the operations $C \leftarrow \alpha AB + \beta C$, $C \leftarrow \alpha AB^T + \beta C$, $B \leftarrow \alpha T^{-1} B$ (T is upper triangular) and their variants. BLAS 3 operations can be implemented in several forms. For example, the matrix product $C = AB$ can be implemented at least in three ways. Let $A \in \mathbb{R}^{m \times r}$, $B \in \mathbb{R}^{r \times n}$.

MATRIX-PRODUCT-DOT-VERSION(A, B)

```
1  m ← rows[A]
2  r ← columns[A]
3  n ← columns[B]
4  C[1 : m, 1 : n] ← 0
5  for i ← 1 to m
6      do for j ← 1 to n
7             do for k ← 1 to r
8                    do C[i, j] ← C[i, j] + A[i, k] B[k, j]
9  return C
```

This algorithm computes c_{ij} as the dot (inner) product of the i^{th} row of A and the j^{th} column of B. This corresponds to the original definition of matrix products.

Now let A, B and C be partitioned columnwise as follows

$$\begin{aligned} A &= [a_1, \ldots, a_r] \quad (a_i \in \mathbb{R}^m), \\ B &= [b_1, \ldots, b_n] \quad (b_i \in \mathbb{R}^r), \\ C &= [c_1, \ldots, c_n] \quad (c_i \in \mathbb{R}^m). \end{aligned}$$

Then we can write c_j as the linear combination of the columns of A, that is

$$c_j = \sum_{k=1}^{r} b_{kj} a_k \quad (j = 1, \ldots, n).$$

Hence the product can be implemented with *saxpy* operations.

MATRIX-PRODUCT-GAXPY-VARIANT(A, B)

```
1  m ← rows[A]
2  r ← columns[A]
3  n ← columns[B]
4  C[1 : m, 1 : n] ← 0
5  for j ← 1 to n
6      do for k ← 1 to r
7             do for i ← 1 to m
8                    do C[i, j] ← C[i, j] + A[i, k] B[k, j]
9  return C
```

The following equivalent form of the „*jki*"-algorithm shows that it is indeed a *gaxpy* based process.

MATRIX-PRODUCT-WITH-GAXPY-CALL(A, B)

```
1  m ← rows[A]
2  n ← columns[B]
3  C[1 : m, 1 : n] ← 0
4  for j ← 1 to n
5      do C[:, j] = gaxpy(A, B[:, j], C[:, j])
6  return C
```

Consider now the partitions $A = [a_1, \ldots, a_r]$ $(a_i \in R^m)$ and

$$B = \begin{bmatrix} b_1^T \\ \vdots \\ b_r^T \end{bmatrix} \quad (b_i \in R^n)\,.$$

Then $C = AB = \sum_{k=1}^{r} a_k b_k^T$.

MATRIX-PRODUCT-OUTER-PRODUCT-VARIANT(A, B)

```
1  m ← rows[A]
2  r ← columns[A]
3  n ← columns[B]
4  C[1 : m, 1 : n] = 0
5  for k ← 1 to r
6      do for j ← 1 to n
7             do for i ← 1 to m
8                    C[i, j] ← C[i, j] + A[i, k] B[k, j]
9  return C
```

The inner loop realizes a *saxpy* operation: it gives the multiple of a_k to the j^{th} column of matrix C.

12.4.2. Mathematical software

These are those programming tools that help easy programming in concise (possibly mathematical) form within an integrated program development system. Such systems were developed primarily for solving mathematical problems. By now they have been extended so that they can be applied in many other fields. For example, Nokia uses MATLAB in the testing and quality control of mobile phones. We give a short review on MATLAB in the next section. We also mention the widely used MAPLE, DERIVE and MATEMATICA systems.

The MATLAB system The MATLAB software was named after the expression MATrix LABoratory. The name indicates that the matrix operations are very easy to make. The initial versions of MATLAB had only one data type: the complex matrix. In the later versions high dimension arrays, cells, records and objects also appeared. The MATLAB can be learned quite easily and even a beginner can write programs for relatively complicated problems.

The coding of matrix operations is similar to their standard mathematical form. For example if A and B are two matrices of the same size, then their sum is given by the command $C = A + B$. As a programming language the MATLAB contains only four control structures known from other programming languages:

– the simple statement $Z =$*expression*,

– the if statement of the form

if *expression, commands* {**else**/**elseif** *commands*} **end**,

– the for loop of the form

for *the values of the loop variable, commands* **end**

– the while loop of the form

while *expression, commands* **end**.

The MATLAB has an extremely large number of built in functions that help efficient programming. We mention the following ones as a sample.

– $\max(A)$ selects the maximum element in every column of A,

– $[v, s] = \text{eig}(A)$ returns the approximate eigenvalues and eigenvectors of A,

– The command $A\backslash b$ returns the numerical solution of the linear system $Ax = b$.

The entrywise operations and partitioning of matrices can be done very efficiently in MATLAB. For example, the statement

$$A([2,3],:) = 1./A([3,2],:)$$

exchange the second and third rows of A while it takes the reciprocal of each element.

The above examples only illustrate the possibilities and easy programming of MATLAB. These examples require much more programming effort in other languages, say e.g. in PASCAL. The built in functions of MATLAB can be easily supplemented by other programs.

The higher number versions of MATLAB include more and more functions and special libraries (tool boxes) to solve special problems such as optimisation, statistics and so on.

There is a built in automatic technique to store and handle sparse matrices that makes the MATLAB competitive in solving large computational problems. The recent versions of MATLAB offer very rich graphic capabilities as well. There is an extra interval arithmetic package that can be downloaded from the WEB site

`http:/www.ti3.tu-harburg.de\%7Erump\intlab`

There is a possibility to build certain C and FORTRAN programs into MATLAB. Finally we mention that the system has an extremely well written help system.

Problems

12-1 Without overflow

Write a MATLAB program that computes the norm $\|x\|_2 = \left(\sum_{i=1}^n x_i^2\right)^{1/2}$ without *overflow* in all cases when the result does not make overflow. It is also required that the error of the final result can not be greater than that of the original formula.

12-2 Estimate

Equation $x^3 - 3.330000x^2 + 3.686300x - 1.356531 = 0$ has the solution $x_1 = 1.01$. The perturbed equation $x^3 - 3.3300x^2 + 3.6863x - 1.3565 = 0$ has the solutions $y_1,\ y_2,\ y_3$. Give an estimate for the perturbation $\min_i |x_1 - y_i|$.

12-3 Double word length

Consider an arithmetic system that has double word length such that every number represented with $2t$ digits are stored in two t digit word. Assume that the computer can only add numbers with t digits. Furthermore assume that the machine can

recognise overflow.
(i) Find an algorithm that add two positive numbers of $2t$ digit length.
(ii) If the representation of numbers requires the sign digit for all numbers, then modify algorithm (i) so that it can add negative and positive numbers both of the same sign. We can assume that the sum does not overflow.

12-4 Auchmuty theorem
Write a MATLAB program for the Auchmuty error estimate (see Theorem 12.22) and perform the following numerical testing.
(i) Solve the linear systems $Ax = b_i$, where $A \in \mathbb{R}^{n\times n}$ is a given matrix, $b_i = Ay_i$, $y_i \in \mathbb{R}^n$ $(i = 1, \ldots, N)$ are random vectors such that $\|y_i\|_\infty \leq \beta$. Compare the true errors $\|\widetilde{x}_i - y_i\|$, $(i = 1, \ldots, N)$ and the estimated errors $EST_i = \|r(\widetilde{x}_i)\|_2^2 / \left\|A^T r(\widetilde{x}_i)\right\|_2$, where $\widetilde{x}_i$ is the approximate solution of $Ax = b_i$. What is the minimum, maximum and average of numbers c_i? Use graphic for the presentation of the results. Suggested values are $n \leq 200$, $\beta = 200$ and $N = 40$.
(ii) Analyse the effect of condition number and size.
(iii) Repeat problems (i) and (ii) using LINPACK and BLAS.

12-5 Hilbert matrix
Consider the linear system $Ax = b$, where $b = [1, 1, 1, 1]^T$ and A is the fourth order Hilbert matrix, that is $a_{i,j} = 1/(i + j)$. A is ill-conditioned. The inverse of A is approximated by

$$B = \begin{bmatrix} 202 & -1212 & 2121 & -1131 \\ -1212 & 8181 & -15271 & 8484 \\ 2121 & -15271 & 29694 & -16968 \\ -1131 & 8484 & -16968 & 9898 \end{bmatrix}.$$

Thus an x_0 approximation of the true solution x is given by $x_0 = Bb$. Although the true solution is also integer x_0 is not an acceptable approximation. Apply the *iterative refinement* with B instead of A^{-1} to find an acceptable integer solution.

12-6 Consistent norm
Let $\|A\|$ be a *consistent norm* and consider the linear system $Ax = b$
(i) Prove that if $A + \Delta A$ is singular, then $\text{cond}(A) \geq \|A\| / \|\Delta A\|$.
(ii) Show that for the 2-norm equality holds in (i), if $\Delta A = -bx^T/(b^t x)$ and $\left\|A^{-1}\right\|_2 \|b\|_2 = \left\|A^{-1}b\right\|_2$.
(iii) Using the result of (i) give a lower bound to $\text{cond}_\infty(A)$, if

$$A = \begin{bmatrix} 1 & -1 & 1 \\ -1 & \varepsilon & \varepsilon \\ 1 & \varepsilon & \varepsilon \end{bmatrix}.$$

12-7 Cholesky-method

Use the Cholesky-method to solve the linear system $Ax = b$, where

$$A = \begin{bmatrix} 5.5 & 0 & 0 & 0 & 0 & 3.5 \\ 0 & 5.5 & 0 & 0 & 0 & 1.5 \\ 0 & 0 & 6.25 & 0 & 3.75 & 0 \\ 0 & 0 & 0 & 5.5 & 0 & 0.5 \\ 0 & 0 & 3.75 & 0 & 6.25 & 0 \\ 3.5 & 1.5 & 0 & 0.5 & 0 & 5.5 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Also give the exact *Cholesky-decomposition* $A = LL^T$ and the true solution of $Ax = b$. The approximate Cholesky-factor $\widetilde{L}$ satisfies the relation $\tilde{L}\tilde{L}^T = A + F$. It can proved that in a *floating point arithmetic* with t-digit mantissa and base β the entries of F satisfy the inequality $|f_{i,j}| \leq e_{i,j}$, where

$$E = \beta^{1-t} \begin{bmatrix} 11 & 0 & 0 & 0 & 0 & 3.5 \\ 0 & 11 & 0 & 0 & 0 & 1.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 11 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 3.5 & 1.5 & 0 & 0.5 & 0 & 11 \end{bmatrix}.$$

Give a bound for the relative error of the approximate solution $\tilde{x}$, if $\beta = 16$ and $t = 14$ (IBM3033).

12-8 Bauer-Fike theorem

Let

$$A = \begin{bmatrix} 10 & 10 & & & & \\ & 9 & 10 & & & \\ & & 8 & 10 & & \\ & & & \ddots & \ddots & \\ & & & & 2 & 10 \\ \varepsilon & & & & & 1 \end{bmatrix}$$

(i) Analyze the perturbation of the eigenvalues for $\varepsilon = 10^{-5}, 10^{-6}, 10^{-7}, 0$.
(ii) Compare the estimate of Bauer-Fike theorem to the matrix $A = A(0)$.

12-9 Eigenvalues

Using the MATLAB eig routine compute the eigenvalues of$B = AA^T$ for various (random) matrices $A \in \mathbb{R}^{n \times n}$ and order n. Also compute the eigenvalues of the perturbed matrices $B + R_i$, where R_i are random matrices with entries from the interval $\left[-10^{-5}, 10^{-5}\right]$ $(i = 1, \ldots, N)$. What is the maximum perturbation of the eigenvalues? How precise is the Bauer-Fike estimate? Suggested values are $N = 10$ and $5 \leq n \leq 200$. How do the results depend on the condition number and the order n? Display the maximum perturbations and the Bauer-Fike estimates graphically.

Chapter notes

The a posteriori error estimates of linear algebraic systems are not completely reliable. Demmel, Diament és Malajovich [58] showed that for the $\Theta\left(n^2\right)$ number

estimators there are always cases when the estimate is unreliable (the error of the estimate exceeds a given order). The first appearance of the iterative improvement is due to Fox, Goodwin, Turing and Wilkinson (1946). The experiences show that the decrease of the residual error is not monotone.

Young [262], Hageman and Young [99] give an excellent survey of the theory and application of iterative methods. Barett, Berry et al. [21] give a software oriented survey of the subject. Frommer [72] concentrates on the parallel computations.

The convergence of the QR-method is a delicate matter. It is analyzed in great depth and much better results than Theorem 12.34 exist in the literature. There are QR-like methods that involve double shifting. Batterson [22] showed that there exists a 3×3 Hessenberg matrix with complex eigenvalues such that convergence cannot be achieved even with multiple shifting.

Several other methods are known for solving the eigenvalue problems (see, e.g. [255], [252]). The LR-method is one of the best known ones. It is very effective on positive definite Hermitian matrices. The LR-method computes the Cholesky-decomposition $A_k = LL^*$ and sets $A_{k+1} = L^*L$.

Bibliography

[1] M. Agrawal, N. Kayal, N. Saxena. Primes is in p. *Annals of Mathematics*, 160(2):781–793, 2004. 350

[2] M. Agrawal, N. Kayal, N. Saxena. PRIMES is in P. http://www.cse.iitk.ac.in/users/manindra/, 2002. 350

[3] R. Ahlswede, B. Balkenhol, L. Khachatrian. Some properties of fix-free codes. In *Proceedings of the 1st International Seminarium on Coding Theory and Combinatorics*, 1996 (Thahkadzor, Armenia), pp. 20–23. 167

[4] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986. 129

[5] A. V. Aho, J. D. Ullman. *The Theory of Parsing, Translation and Compiling Vol. I*. Prentice-Hall, 1972. 78, 129

[6] A. V. Aho, J. D. Ullman. *The Theory of Parsing, Translation and Compiling Vol. II*. Prentice-Hall, 1973. 78, 129

[7] M. Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, 1998, pp. 10–19. 273

[8] M. Ajtai, J. Komlós, E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983. 213

[9] A. G. Akritas. *Elements of Computer Algebra with Applications*. John Wiley & Sons, 1989. 331

[10] S. Albers, H. Bals. Dynamic TCP acknowledgement, penalizing long delays. In *Proceedings of the 25th ACM-SIAM Symposium on Discrete Algorithms*, pp. 47–55, 2003. 427

[11] V. Arvind, P. Kurur. Graph isomorphism is in SPP. In *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science*, 743–750. pages. IEEE Computer Society Press, 2002. 392

[12] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. *Journal of the ACM*, 44:486–504, 1997. 428

[13] A. Asteroth, C. Christel. *Theoretische Informatik*. Pearson Studium, 2002. 79

[14] J-P. Aubin. *Mathematical Methods of Game and Economic Theory*. North-Holland, 1979. 476

[15] B. Awerbuch, Y. Azar S. Plotkin. Throughput-competitive online routing. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pp. 32–40, 1993. 428

[16] Y. Azar. On-line load balancing. Lecture Notes in Computer Science, Vol. 1442. Springer-Verlag, pp. 178–195, 1998. 428

[17] L. Babai. Trading group theory for randomness. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, 421–429. pages. ACM Press, 1985. 363

[18] L. Babai, S. Moran. Arthur-Merlin games: A randomized proof system, and a hierarchy of complexity classes. *Journal of Computer and Systems Sciences*, 36(2):254–276, 1988. 363

[19] B. S. Baker, J. S. Schwartz. Shelf algorithms for two dimensional packing problems. *SIAM Journal on Computing*, 12:508–525, 1983. 428

[20] B. Balkenhol, S. Kurtz. Universal data compression based on the Burrows-Wheeler transform: theory and practice. *IEEE Transactions on Computers*, 49(10):1043–1053–953, 2000. 167

[21] R. Barett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozzo, C. Romine, H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994. 551

[22] S. Batterson. Convergence of the shifted QR algorithm on 3×3 normal matrices. *Numerische Mathematik*, 58:341–352, 1990. 551

[23] T. C. Bell, I. H. Witten, J. G. Cleary. Modeling for text compression. *Communications of the ACM*, 21:557–591, 1989. 167

[24] T. C. Bell, I. H. Witten, J. G. Cleary. *Text Compression*. Prentice Hall, 1990. 167, 168

[25] E. Bender, R. Canfield. The asymptotic number of labeled graphs with given degree sequences. *Combinatorial Theory Series A*, 24:296–307, 1978. 213

[26] A. Beygelzimer, L. A. Hemaspaandra, C. Homan, J. Rothe. One-way functions in worst-case cryptography: Algebraic and security properties are on the house. *SIGACT News*, 30(4):25–40, 1999. 363

[27] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999. 363

[28] B. Borchert, L. A. Hemaspaandra, J. Rothe. Restrictive acceptance suffices for equivalence problems. *London Mathematical Society Journal of Computation and Mathematics*, 86:86–95, 2000. 392

[29] A. Borodin R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998. 427

[30] J. G. Brookshear. *Theory of Computation – Formal Langauges, Automata, and Complexity*. The Benjamin/Cummings Publishing Company, 1989. 79

[31] L. E. J. Brouwer. Über Abbildung von Manningfaltigkeiten. *Mathematische Annalen*, 82(3–4):286–292, 1921. 476

[32] T. Brueggemann, W. Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1–3):303–313, 2004. 375, 391, 392

[33] B. Buchberger. Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal, 1965. PhD dissertation, Leopold-Franzens-Universität, Innsbruck. 330

[34] M. Burrows, D. J. Wheeler. A block-sorting lossless data compression algorithm. Research Report 124, http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html, 1994. 167

[35] Calgary. The Calgary/Canterbury Text Compression. ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression. corpus, 2004. 168

[36] Canterbury. The Canterbury Corpus. http://corpus.canterbury.ac.nz, 2004. 168

[37] J. Carroll, D. Long. *Theory of Finite Automata*. Prentice Hall, 1989. 79

[38] J. L. Carter, M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. 383, 392

[39] B. F. Caviness. Computer algebra: past and future. *Journal of Symbolic Computations*, 2:217–263, 1986. 331

[40] Y. Cho, S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9(1):91–103, 1980. 428

[41] S. M. Christensen. Resources for computer algebra. *Computers in Physics*, 8:308–315, 1994. 331

[42] M. Chrobak, L. Larmore. An optimal algorithm for k-servers on trees. *SIAM Journal on Computing*, 20:144–148, 1991. 427

[43] M. Chrobak, L. Larmore. The server problem and on-line games. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 7, pp. 11-64. American Mathematical Society, 1992. 427

[44] M. Chrobak, H. J. Karloff, T. Payne, S. Vishwanathan. New results on the server problem. *SIAM Journal on Discrete Mathematics*, 4:172–181, 1991. 427

[45] E. Coffman. *Computer and Job Shop Scheduling*. John Wiley & Sons, 1976. 428

[46] A. M. Cohen, L. van Gasten, S. Lunel (editors). *Computer Algebra for Industry 2, Problem Solving in Practice*. John Wiley & Sons, 1995. 331

[47] A. M. Cohen (editor). *Computer Algebra for Industry: Problem Solving in Practice*. John Wiley & Sons, 1993. 331

[48] S. Cook. The complexity of theorem proving procedures. 151–158. pages. ACM Press, 1971. 391

[49] K. D. Cooper, L. Torczon. *Engineering a Compiler*. Morgan Kaufman Publisher, 2004. 130

[50] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology*, 10(4):233–260, 1997. 363

[51] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 2007 (8. corrected printing of 2. edition). 167, 330

[52] T. M. Cover, J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, 1991. 168

[53] J. Csirik, G. Woeginger. On-line packing and covering problems. Lecture Notes in Computer Science, Vol. 1442, pp. 147–177. Springer-Verlag, 1998. 428

[54] J. Csirik, G. J. Woeginger. Shelf algorithms for on-line strip packing. *Information Processing Letters*, 63:171–175, 1997. 428

[55] I. Csiszár, J. Körner. *Coding Theorems for Discrete Memoryless Systems*. Akadémiai Kiadó, 1981. 168

[56] E. Dantsin, A. Goerdt, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, U. Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-sat based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002. 375, 391, 392

[57] J. Davenport, Y. Siret, E. Tournier. *Computer Algebra: Systems and Algorithms for Algebraic Computation*. Academic Press, 2000. 331

[58] J. Demmel, D. Malajovich. On the complexity of computing error bounds. *Foundations of Computational Mathematics*, 1:101–125, 2001. 550

[59] R. Dobrushin, S. Ortyukov. Lower bound for the redundancy of self-correcting arrangements of unreliable functional elements. *Problems of Information Transmission* (translated from Russian), 13(1):59–65, 1977. 213

[60] R. Dobrushin, S. Ortyukov. Upper bound for the redundancy of self-correcting arrangements of unreliable elements. *Problems of Information Transmission* (translated from Russian), 13(3):201–208, 1977. 213

[61] D. R. Dooly, S. A. Goldman, S. D. Scott. On-line analysis of the TCP acknowledgement delay problem. *Journal of the ACM*, 48:243–273, 2001. 427, 428

[62] Gy. Dósa, Y. He. Better online algorithms for scheduling with machine cost. *SIAM Journal on Computing*, 33(5):1035–1051, 2004. 428

[63] D.-Z. Du, K-I. Ko. *Problem Solving in Automata, Languages, and Complexity*. John Wiley & Sons, 2001. 79

[64] M. Effros, K. Viswesvariah, S. Kulkarni, S. Verdú. Universal lossless source coding with the Burrows-Wheeler transform. *IEEE Transactions on Information Theory*, 48(5):1061–1081, 2002. 167

[65] S. N. Elaydi. *An Introduction to Difference Equations*. Springer-Verlag, 1999 (2. edition). 501

[66] S. Fenner, L. Fortnow, S. Kurtz. Gap-definable counting classes. *Journal of Computer and System Sciences*, 48(1):116–148, 1994. 386, 392

[67] A. Fiat, Y. Rabani, Y. Ravid. Competitive k-server algorithms. *Journal of Computer and System Sciences*, 48:410–428, 1994. 427

[68] A. Fiat, G. Woeginger (editors). *Online Algorithms. The State of Art*. Springer-Verlag, 1998. 427

[69] C. N. Fischer, R. LeBlanc (editors). *Crafting a Compiler*. The Benjamin/Cummings Publishing Company, 1988. 129

[70] R. Fleischer, M. Wahl. On-line scheduling revisited. *Journal of Scheduling*, 3(6):343–353, 2000. 428

[71] F. Forgó, J. Szép, F. Szidarovszky. *Introduction to the Theory of Games: Concepts, Methods and Applications*. Kluwer Academic Publishers, 1999. 476, 477

[72] A. Frommer. *Lösung linearer Gleichungssysteme auf Parallelrechnern.* Vieweg Verlag, 1990. 551

[73] I. Gaál. *Diophantine Equations and Power Integral Bases: New Computational Methods.* Birkhäuser Boston, 2002. 273

[74] R. Gallager. *Low-density Parity-check Codes.* The MIT Press, 1963. 213

[75] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979. 369, 370, 373, 391

[76] J. Gathen, von zur. *Modern Computer Algebra.* Cambridge University Press, 2003. 330, 331

[77] J. Gathen, von zur, J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999. 273

[78] F. Gécseg, I. Peák. *Algebraic Theory of Automata.* Akadémiai Kiadó, 1972. 78

[79] P. Gács. Reliable cellular automata with self-organization. *Journal of Statistical Physics*, 103(1–2):45–267, 2001. See also www.arXiv.org/abs/math.PR/0003117 and The Proceedings of the 1997 Symposium on the Theory of Computing. 214

[80] P. Gács, A. Gál. Lower bounds for the complexity of reliable Boolean circuits with noisy gates. *IEEE Transactions on Information Theory*, 40(2):579–583, 1994. 213

[81] P. Gács, J. Reif. A simple three-dimensional real-time reliable cellular array. *Journal of Computer and System Sciences*, 36(2):125–147, 1988. 214

[82] K. O. Geddes, S. Czapor, G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, 1992. 330, 331

[83] D. Giammarresi, R. Montalbano. Deterministic generalized automata. *Theoretical Computer Science*, 215(1–2):191–208, 1999. 78

[84] O. Goldreich, S. Micali, A. Wigderson. Proofs that yield nothing but their validity or all languages in NP. *Journal of the ACM*, 38(3):691–729, 1991. 359

[85] O. Goldreich. Randomness, interactive proofs, and zero-knowledge – a survey. In R. Herken (editor), *The Universal Turing Machine: A Half-Century Survey*, 377–405. pages. Oxford University Press, 1988. 363

[86] O. Goldreich. *Foundations of Cryptography.* Cambridge University Press, 2001. 363

[87] S. Goldwasser. Interactive proof systems. In J. Hartmanis (editor), *Computational Complexity Theory, AMS Short Course Lecture Notes: Introductory Survey Lectures. Proceedings of Symposia in Applied Mathematics* 38. kötete, 108–128. pages. American Mathematical Society, 1989. 363

[88] S. Goldwasser, S. Micali, C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989. 363

[89] S. Goldwasser, M. Sipser. Private coins versus public coins in interactive proof systems. In S. Micali (editor), *Randomness and Computation*, *Advances in Computing Research* 5. kötete, 73–90. pages. JAI Press, 1989. A preliminary version appeared in *Proc. 18th Ann. ACM Symp. on Theory of Computing*, 1986, pp. 59–68. 358

[90] G. Gonnet, D. Gruntz, L. Bernardin. Computer algebra systems. In A. Ralston, E. D. Reilly, D. Hemmendinger (editors), *Encyclopedia of Computer Science*, 287–301. pages. Nature Publishing Group, 4. edition, 2000. 331

[91] R. L. Graham. Bounds for certain multiprocessor anomalies. *The Bell System Technical Journal*, 45:1563–1581, 1966. 428

[92] R. L. Graham, D. E. Knuth, O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 1994 (second edition). 501

[93] D. H. Greene, D. E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkhäuser, 1990 (3. edition). 501

[94] D. Gries. *Compiler Construction for Digital Computers.* John Wiley & Sons, 1971. 129

[95] R. Grossman. *Symbolic Computation: Applications to Scientific Computing.* Frontiers in Applied Mathematics. Vol. 5 SIAM, 1989. 331

[96] D. Grune, H. Bal, C. J. H. Jacobs, K. Langendoen. *Modern Compiler Design.* John Wiley & Sons, 2000. 130

[97] J. Håstad. Solving simultaneous modular equations of low degree. *SIAM Journal on Computing*, 17(2):336–341, 1988. Special issue on cryptography. 363

[98] G. Hadley. *Nonlinear and Dynamic Programming.* Addison-Wesley, 1964. 477

[99] L. Hageman, D. Young. *Applied Iterative Methods*. Academic Press, 1981. 551

[100] T. S. Han, K. Kobayashi. *Mathematics of Information and Coding*. American Mathematical Society, 2002. 168

[101] D. Hankerson, G. A. Harris, P. D. Johnson. *Introduction to Information Theory and Data Compression*. CRC Press, 2003 (2. edition). 168

[102] D. Harper, C. Wooff, D. Hodginson. *A Guide to Computer Algebra Systems*. John Wiley & Sons, 1991. 331

[103] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, 1978. 78

[104] A. C. Hearn. *Future Directions for Research in Symbolic Computation. SIAM Reports on Issues in the Mathematical Sciences*. SIAM, 1990. 331

[105] L. A. Hemaspaandra, J. Rothe, A. Saxena. Enforcing and defying associativity, commutativity, totality, and strong noninvertibility for one-way functions in complexity theory. In M. Coppo et al. (editor), *ICTCS 2005*, *Lecture Notes in Computer Science* 117. kötete, 265–279. pages. Springer verlag, 2005. 356, 363

[106] L. A. Hemaspaandra, M. Ogihara. *The Complexity Theory Companion*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2002. 391

[107] L. A. Hemaspaandra, K. Pasanen, J. Rothe. If P $\neq$ NP then some strongly noninvertible functions are invertible. *Theoretical Computer Science*, 362(1–3):54–62, 2006. 356, 363

[108] L. A. Hemaspaandra, J. Rothe. Creating strong, total, commutative, associative one-way functions from any one-way function in complexity theory. *Journal of Computer and Systems Sciences*, 58(3):648–659, 1999. 356, 363

[109] C. Hoffman (editor). *Group-Theoretic Algorithms and Graph Isomorphism*, *Lecture Notes in Computer Science* 136. kötete. Springer-Verlag, 1982. 392

[110] C. Homan. Tight lower bounds on the ambiguity in strong, total, associative, one-way functions. *Journal of Computer and System Sciences*, 68(3):657–674, 2004. 363

[111] J. E. Hopcroft, R. Motwani, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001 (in German: *Einführung in Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Pearson Studium, 2002). 2. edition. 79

[112] J. E. Hopcroft, J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979. 78

[113] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. 166

[114] T. W. Hungerford. *Abstract Algebra: An Introduction*. Saunders College Publishers, 1990. 273

[115] R. W. Hunter. *Compilers, Their Design and Construction using Pascal*. John Wiley & Sons, 1985. 129

[116] Cs. Imreh. Online strip packing with modifiable boxes. *Operations Research Letters*, 66:79–86, 2001. 428

[117] Cs. Imreh, J. Noga. Scheduling with machine cost. In *Proceedings of APPROX'99*, Lecture Notes in Computer Science, Vol. 1540, pp. 168–176, 1999. 428

[118] A. Iványi. Performance bounds for simple bin packing algorithms. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 5:77–82, 1984. 428

[119] K. Iwama, S. Tamaki. Improved upper bounds for 3-SAT. In J. Munro (editor), *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 328–329. pages. Society for Industrial and Applied Mathematics, Society for Industrial and Applied Mathematics. 391, 392

[120] D. S. Johnson. *Near-Optimal Bin Packing Algorithms*. PhD thesis, MIT Department of Mathematics, 1973. 428

[121] D. S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974. 428

[122] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, R. L. Graham. Worst-case performance-bounds for simple one-dimensional bin packing algorithms. *SIAM Journal on Computing*, 3:299–325, 1974. 428

[123] S. Kakutani. A generalization of Brouwer's fixed point theorem. *Duke Mathematical Journal*, 8:457–459, 1941. 476

[124] B. Kaliski, M. Robshaw. The secure use of RSA. *CryptoBytes*, 1(3):7–13, 1995. 363

[125] S. Karamardian. The nonlinear complementarity problems with applications. I, II. *Journal of Optimization Theory and Applications*, 4:87–98 and 167–181, 1969. 477

[126] Z. Karian, A. Starrett. Use of symbolic computation in probability and statistics. In Z. Karian (editor), *Symbolic Computation in Undergraduate Mathematics Education*, Number 24 in *Notes of Mathematical Association of America*. Mathematical Association of America, 1992. 331

[127] A. R. Karlin, C. Kenyon, D. Randall. Dynamic TCP acknowledgement and other stories about $e/(e-1)$. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, 502–509. pages, 2001. 427

[128] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller, J. W. Thatcher (editors), *Complexity of Computer Computations*, 85–103. pages. Plenum Press, 1972. 391

[129] J. Köbler, U. Schöning, S. Toda, J. Torán. Turing machines with few accepting computations and low sets for PP. *Journal of Computer and System Sciences*, 44(2):272–286, 1992. 362, 386, 392

[130] J. Köbler, U. Schöning, J. Torán. Graph isomorphism is low for PP. *Computational Complexity*, 2:301–330, 1992. 386, 388, 392

[131] J. Köbler, U. Schöning, J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhäuser, 1993. 363, 392

[132] D. Kelley. *Automata and Formal Languages*. Prentice Hall, 1995. 79

[133] J. Kim, V. Vu. Generating random regular graphs. In *Proceedings of the Thirty Fifth ACM Symposium on Theory of Computing*, pages 213–222, 2003. 213

[134] D. E. Knuth. *Fundamental Algorithms. The Art of Computer Programming*. Vol. 1. Addison-Wesley, 1968 (third updated edition). 331, 501

[135] D. E. Knuth. *Seminumerical Algorithms. The Art of Computer Programming*. Vol. 2. Addison-Wesley, 1969 (3. corrected edition). 331

[136] D. E. Knuth. *Sorting and Searching. The Art of Computer Programming*. Vol. 3. Addison-Wesley, 1973 (3. corrected edition). 331

[137] E. Koutsoupias, C. Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42:971–983, 1995. 427

[138] A. Kovács. Computer algebra: Impact and perspectives. *Nieuw Archief voor Wiskunde*, 17(1):29–55, 1999. 331

[139] D. C. Kozen. *Automata and Computability*. Springer-Verlag, 1995. 79

[140] R. Krichevsky, V. Trofimov. The performance of universal encoding. *IEEE Transactions on Information Theory*, 27:199–207, 1981. 167

[141] Z. Kása. *Combinatorică cu aplicaţii* (*Combinatorics with Applications*). Presa Universitară Clujeană, 2003. 501

[142] H. W. Kuhn, A. Tucker (editors). *Contributions to the Theory of Games. II*. Princeton University Press, 1953. 476

[143] S. Kurtz, B. Balkenhol. Space efficient linear time computation of the Burrows and Wheeler transformation. in I. Althöfer, N. Cai, G. Dueck, L. Khachatrian, M. Pinsker, A. Sárközy, I. Wegener, Z. Zhang (editors) *Numbers, Information and Complexity*. Kluwer Academic Publishers, 2000, pp. 375–383. 167

[144] A. V. Kuznetsov. Information storage in a memory assembled from unreliablecomponents. *Problems of Information Transmission* (translated from Russian), 9(3):254–264, 1973. 213

[145] R. Ladner, N. A. Lynch, A. Selman. A comparison of polynomial time reducibilities. *Theoretical Computer Science*, 1(2):103–124, 1975. 379, 391

[146] G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28:135–149, 1984. 167

[147] M. V. Lawson. *Finite Automata*. Chapman & Hall/CRC, 2004. 79

[148] A. Lenstra, H. Lenstra. *The Development of the Number Field Sieve*, *Lecture Notes in Mathematics* 1554. kötete. Springer-Verlag, 1993. 354

[149] A. K. Lenstra, H. W. Lenstra, Jr., L. Lovász. Factoring polynomials with integer coefficients. *Mathematische Annalen*, 261:513–534, 1982. 273, 327

[150] S. Leonardi. On-line network routing. Lecture Notes in Computer Science, Vol. 1442, pp. 242–267. Springer-Verlag, 1998. 428

[151] R. Lidl, H. Niederreiter. *Introduction to Finite Fields and Their Applications.* Cambridge University Press, 1986. 273

[152] P. Linz. *An Introduction to Formal Languages and Automata.* Jones and Barlett Publishers, 2001. 79

[153] M. Lothaire. *Algebraic Combinatorics on Words.* Cambridge University Press, 2002. 78

[154] K. Louden. Compilers and interpreters. In A. B. Tucker (editor), *Handbook of Computer Science*, 99/1–99/30. pages. Chapman & Hall/CRC, 2004. 130

[155] L. Lovász. *Combinatorial Problems and Exercises.* Akadémiai Kiadó/North Holland Publ. House, 1979. 501

[156] O. B. Lupanov. On a method of circuit synthesis. *Izvestia VUZ (Radiofizika)*, pages 120–140, 1958. 213

[157] R. Mak. *Writing Compilers and Interpreters.* Addison-Wesley, 1991. 130

[158] M. Manasse, L. McGeoch, D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990. 427

[159] O. Mangasarian, H. Stone. Two-person zero-sum games and quadratic programming. *Journal of Mathematical Analysis and its Applications*, 9:348–355, 1964. 476

[160] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill Book Co., 1974. 78

[161] B. Martos. *Nonlinear Programming Theory and Methods.* Akadémiai Kiadó, 1975. 476

[162] R. Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–132, 1979. 388

[163] A. Meduna. *Automata and Languages: Theory and Applications.* Springer-Verlag, 2000. 79

[164] A. Meyer, L. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th IEEE Symposium on Switching and Automata Theory*, pages 129–129. 1972. 392

[165] D. Micciancio, S. Goldwasser. *Complexity of Lattice Problems: A Cryptographic Perspective.* Vol. 671.,The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 2002. 363

[166] R. E. Mickens. *Difference Equations. Theory and Applications.* Van Nostrand Reinhold, 1990. 501

[167] M. E. Mignotte. *Mathematics for Computer Algebra.* Springer, 1992. 331

[168] G. L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and Systems Sciences*, 13(3):300–317, 1976. 351

[169] H. Mills. Equilibrium points of finite games. *SIAM Journal of Applied Mathematics*, 8:397–402, 1976. 476

[170] B. E. Mishra. *Algorithmic Algebra.* Springer, 1993. 331

[171] R. N. Moll, M. A. Arbib, A. J. Kfoury. *An Introduction to Formal Language Theory.* Springer-Verlag, 1988. 79

[172] B. Monien, E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985. 375, 392

[173] J. Moore. Protocol failures in cryptosystems. In G. Simmons (editor), *Contemporary Cryptology: The Science of Information Integrity*, 541–558. pages. IEEE Computer Society Press, 1992. 363

[174] R. Motwani, P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995. 213

[175] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufman Publisher, 1997. 130

[176] J. Nash. Noncooperative games. *Annals of Mathematics*, 54:286–295, 1951. 476

[177] M. Nelson, J. L. Gailly. *The Data Compression Book.* M&T Books, 1996. 168

[178] J. Neumann, O. Morgenstern. *Theory of Games and Economical Behaviour.* Princeton University Press, 1947 (2. edition). 477

[179] J. Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. Princeton University Press, 1956. 213

[180] H. Nikaido, K. Isoda. Note on noncooperative games. *Pacific Journal of Mathematics*, 5:807–815, 1955. 476

[181] A. Odlyzko. *Applications of Symbolic Mathematics to Mathematics*. Kluwer Academic Publishers, 1985. 331

[182] K. Okuguchi. *Expectation and Stability of Oligopoly Models*. Springer, 1976. 477

[183] K. Okuguchi, F. Szidarovszky. *The Theory of Oligopoly with Multi-Product Firms*. Springer, 1999 (2. kiadás). 477

[184] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. 363, 391

[185] R. Pasco. *Source Coding Algorithms for Fast Data Compression*. PhD thesis, Stanford University, 1976. 167

[186] R. Paturi, P. Pudlák, M. Saks, F. Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science*, 628–637. pages. IEEE Computer Society Press, 1998. 391, 392

[187] R. Pavelle, M. Rothstein. Computer algebra. *Scientific American*, 245(12):102–113, 1981. 331

[188] M. Pinsker. On the complexity of a concentrator. *International Teletraffic Congr.*, 7:318/1–318/4, 1973. 213, 214

[189] N. Pippenger, G. Staomulis, J. N. Tsitsiklis. On a lower bound for the redundancy of reliable networks with noisy gates. *IEEE Transactions on Information Theory*, 37(3):639–643, 1991. 213

[190] N. Pippenger. Analysis of error correction by majority voting. In S. Micali (editor), *Randomness in Computation*. JAI Press, 1989. 213

[191] N. Pippenger. On networks of noisy gates. In *Proceeding of the 26th IEE FOCS Symposium*, pages 30–38, 1985. 213

[192] T. Pittman. *The Art of Compiler Design, Theory and Practice*. Prentice Hall, 1992. 129

[193] J. M. Pollard. Theorems on factorization and primality testing. *Proceedings of the Cambridge Philosophical Society*, 76:521–528, 1974. 354

[194] M. Rabi, A. Sherman. An observation on associative one-way functions in complexity theory. *Information Processing Letters*, 64(5):239–244, 1997. 363

[195] M. O. Rabin. Probabilistic algorithms for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980. 351

[196] R. Rao, J. Rothe, O. Watanabe. Upward separation for FewP and related classes. *Information Processing Letters*, 52(4):175–180, 1994 (Corrigendum appears in the same journal,74(1–2):89, 2000). 392

[197] R. Reischuk, B. Schmelz, B.. Reliable computation with noisy circuits and decision trees—a general $n \log n$ lower bound. In *Proceedings of the 32-nd IEEE FOCS Symposium*, pages 602–611, 1991. 213

[198] J. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198–203, 1976. 167

[199] J. Ritt. *Integration in Finite Terms*. Columbia University Press, 1948. 331

[200] R. L. Rivest, A. Shamir, L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. 349

[201] J. Robinson. An iterative method of solving a game. *Annals of Mathematics*, 154:296–301, 1951. 476

[202] J. Rosen. Existence and uniqueness of equilibrium points for concave n-person games. *Econometrica*, 33:520–534, 1965. 477

[203] J. Rothe. Some facets of complexity theory and cryptography: A five-lecture tutorial. *ACM Computing Surveys*, 34(4):504–549, 2002. 353, 363

[204] J. Rothe. A promise class at least as hard as the polynomial hierarchy. *Journal of Computing and Information*, 1(1):92–107, 1995. 392

[205] J. Rothe. *Complexity Theory and Cryptology. An Introduction to Cryptocomplexity*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2005. 354, 363, 391

[206] G. Rozenberg, A. Salomaa. *Handbook of Formal Languages, Volumes I–III*. Springer-Verlag, 1997. 78

[207] A. Salomaa. *Theory of Automata*. Pergamon Press, 1969. 78

[208] A. Salomaa. *Formal Languages.* Academic Press, 1987 (second updated edition). 78

[209] A. Salomaa. *Public-Key Cryptography.* EATCS Monographs on Theoretical Computer Science. Vol. 23., Springer-Verlag., 1996 (2. edition). 363

[210] D. Salomon. *Data Compression.* Springer-Verlag, 2004 (3. edition). 168

[211] K. Sayood. *Introduction to Data Compression.* Morgan Kaufman Publisher, 2000 (2. edition). 168

[212] U. Schöning. A low and a high hierarchy within NP. *Journal of Computer and System Sciences*, 27:14–28, 1983. 379, 382, 392

[213] U. Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37:312–323, 1987. 392

[214] U. Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, 410–414. pages. IEEE Computer Society Press, 1999. 375, 391, 392

[215] U. Schöning. *Algorithmik.* Spektrum Akademischer Verlag, 2001. 392

[216] R. Sedgewick, P. Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, 1996. 501

[217] A. Selman. Polynomial time enumeration reducibility. *SIAM Journal on Computing*, 7(4):440–457, 1978. 391

[218] J. Sgall. On-line scheduling. Lecture Notes in Computer Science, Vol. 1442, pp. 196–231. Springer-Verlag, 1998. 428

[219] A. Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, 1992. 358

[220] A. Shamir. RSA for paranoids. *CryptoBytes*, 1(3):1–4, 1995. 363

[221] C. Shannon. The synthesis of two-terminal switching circuits. *The Bell Systems Technical Journal*, 28:59–98, 1949. 213

[222] H. N. Shapiro. Note on a computation method in the theory of games. *Communications on Pure and Applied Mathematics*, 11:587–593, 1958. 476

[223] D. B. Shmoys, J. Wein, D. P. Williamson. Scheduling parallel machines online. *SIAM Journal on Computing*, 24:1313–1331, 1995. 428

[224] I. E. Shparlinski. *Finite Fields: Theory and Computation The Meeting Point of Number Theory, Computer Science, Coding Theory, and Cryptography.* Kluwer Academic Publishers, 1999. 273

[225] M. Simon. *Automata Theory.* World Scientific Publishing Company, 1999. 79

[226] D. A. Simovoci, R. L. Tenney. *Theory of Formal Languages with Applications.* World Scientific Publishing Company, 1999. 79

[227] S. Singh. *The Code Book. The Secret History of Codes and Code Breaking.* Fourth Estate, 1999. 363

[228] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997. 78

[229] D. Sleator R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985. 428

[230] N. P. Smart. *The Algorithmic Resolution of Diophantine Equations. London Mathematical Society Student Text,* Vol. 41. Cambridge University Press, 1998. 273

[231] R. Solovay, V. Strassen. A fast Monte Carlo test for primality. *SIAM Journal on Computing*, 6:84–85, 1977. Erratum appears in the same journal, 7(1):118, 1978. 351

[232] D. Spielman. Highly fault-tolerant parallel computation. In *Proceedings of the 37th IEEE Foundations of Computer Science Symposium*, pp. 154–163, 1996. 213

[233] D. Spielman. Linear-time encodable and decodable error-correcting codes. In *Proceedings of the 27th ACM STOC Symposium*, pp. 387–397, 1995 (further IEEE Transactions on Information Theory 42(6):1723–1732). 213

[234] D. Stinson. *Cryptography: Theory and Practice.* CRC Press, 2002 (2. edition). 354, 363

[235] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1977. 392

[236] H. Straubing. *Finite Automata, Formal Logic, and Circuit Complexity.* Birkhäuser, 1994. 79

[237] T. A. Sudkamp. *Languages and Machines.* Addison-Wesley, 1997. 79

[238] F. Szidarovszky, C. Chiarella. Dynamic oligopolies, stability and bifurcation. *Cubo Mathemática Educational*, 3(2):267–284, 2001. 477

[239] F. Szidarovszky, S. Yakowitz. *Principles and Procedures of Numerical Analysis*. Plenum Press, 1998. 476, 477

[240] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–308, 1955. 476

[241] D. S. Taubman, M. W. Marcelin. *JPEG 2000 – Image Compression, Fundamentals, Standards and Practice*. Society for Industrial and Applied Mathematics, 1983. 167

[242] M. G. Taylor. Reliable information storage in memories designed from unreliable components. *The Bell Systems Technical Journal*, 47(10):2299–2337, 1968. 213

[243] J-P. Tremblay, P. G. Sorenson. *Compiler Writing*. McGraw-Hill Book Co., 1985. 129

[244] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, ser. 2*, 2:230–265, 1936 (Correction, *ibid*, vol. 43, pp. 544–546, 1937). 365

[245] J. J. Uhl. MATHEMATICA and Me. *Notices of AMS*, 35:1345–1345, 1988. 331

[246] L. G. Valiant. The relative complexity of checking and evaluating. *Information Processing Letters*, 5(1):20–23, 1976. 392

[247] A. van Vliet. An improved lower bound for on-line bin packing algorithms. *Information Processing Letters*, 43:277–284, 1992. 428

[248] N. J. Vilenkin. *Combinatorial Mathematics for Recreation*. Mir, 1972. 501

[249] K. Wagner, G. Wechsung. *Computational Complexity*. D. Reidel Publishing Company, 1986 (and Kluwer Academic Publishers, 2001). 391

[250] W. Waite, G. Goos. *Compiler Construction*. Springer-Verlag, 1984. 129

[251] G. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34:30–44, 1991. 167

[252] D. Watkins. Bulge exchanges in algorithms of QR type. *SIAM Journal on Matrix Analysis and Application*, 19(4):1074–1096, 1998. 551

[253] G. Wechsung. *Vorlesungen zur Komplexitätstheorie*. Vol. 32. B. G. Teubner Verlagsgesellschaft, 2000. 391

[254] D. Welsh. *Codes and Cryptography*. Oxford University Press, 1988. 166

[255] J. Wilkinson. Convergence of the LR, QR, and related algorithms. *The Computer Journal*, 8(1):77–84, 1965. 551

[256] F. M. J. Willems, Y. M. Shtarkov, T. J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Transactions on Information Theory*, 47:653–664, 1995. 167

[257] F. M. J. Willems, Y. M. Shtarkov, T. J. Tjalkens. The context-tree weighting method: basic properties. *IEEE Information Theory Society Newsletter*, 1:1 and 20–27, 1997. 167

[258] F. Winkler. *Polynomial Algorithms in Computer Algebra*. Springer-Verlag, 1990. 331

[259] I. H. Witten, R. M. Neal, J. G. Cleary. Arithmetic coding for sequential data compression. *Communications of the ACM*, 30:520–540, 1987. 167

[260] A. C. C. Yao. New algorithms for bin packing. *Journal of the ACM*, 27:207–227, 1980. 428

[261] N. Young. On-line file caching. *Algorithmica*, 33:371–383, 2002. 427

[262] D. M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, 1971. 551

[263] J. Ziv, A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977. 167

[264] J. Ziv, A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978. 167

[265] S. I. Zuhovitsky, R. A. Polyak, M. E. Primak. Concave n-person games (numerical methods). *Ékonomika i Matematicheskie Methody*, 7:888–900, 1971 (in Russian). 477

[266] A. van Vliet. *Lower and upper bounds for on-line bin packing and scheduling heuristics*. PhD thesis, Erasmus University, Rotterdam, 1995. 428

[267] A. Vestjens. *On-line machine scheduling*. PhD thesis, Eindhoven University of Technology, 1997. 428

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Subject Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, "2-3-4-tree" is indexed as if were "two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: ex for exercise, fig for figure, pr for problem and fn for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

A, Â

abelian monoid, *see* monoid
absolute error, 511*ex*
accept, 100, 118
action table, 118, 124
actual
  symbol, 99, 104
aggregation function, *438*
ALARM algorithm, *404*
alarming algorithm, *404*
algebra, 217–274, *271*
algebraic
  element, 316
  extension, 314, 317, 321, 325
  number field, 317, 319
algebraic number field, 272
algorithms of compilers, 80
alphabet, *13*
ambiguous CF grammar, *70*
analyser, 82
  lexical, 81, 82
  semantic, 81, 83, 92
  syntactic, 81, 82, 92
arithmetic coding, *141*
ARITHMETIC-DECODER, *144*
ARITHMETIC-ENCODER, *143*
arithmetic overflow, 511*ex*
ARITHMETIC-PRECISION-ENCODER, *146*
arithmetic underflow, 511*ex*
ascending chain of ideals, *305*, 306
assembly language program, 83
associate polynomials, *223*
associative, 218
asymptotically C-competitive, *396*
asymptotic competitive ratio, *396*
attack
  chosen-plaintext, *338*
  ciphertext-only, *338*
  encryption-key, *338*
  known-plaintext, *338*
attribute
  grammar, 92
augmented grammar, 110
automata
  equivalence, 34
  minimization, 45
  nondeterministic pushdown, *60*
  pushdown, *60*
AUTOMATON-MINIMIZATION, *46*
automaton
  deterministic
    finite, 85
  finite, *26*
  with empty input, 41
automorhism
  of a graph, *342*
automorphism, *234*
average case analysis, *395*

B

BACK-SUBSTITUTION, *513*
BACKTRACKING-SAT, *374*
backward error, *504*
backward error analysis, *504*
backward induction, *433*
BACKWARD-INDUCTION, *435*
backward stable, *505*
BAL, *see* BALANCE
BALANCE, *398*
balancing, *530*
banded matrix, 521
basis
  of a lattice, *249*, 251, 269, 270
    reduced, 256–259, 269
    weakly reduced, *255*, 256
  of a vector space, *220*, 231, 272
  of the ideal, *301*
  orthogonal, 257
  orthonormal, 248, 251
BERLEKAMP-DETERMINISTIC, *246*

Berlekamp-Randomised, *247*
Berlekamp subalgebra, *243*, 245–247
  absolute, *243*, 248
Berlekamp-Zassenhaus, *267*
best reply mapping, *437*
bimatrix games, *444*
binary trees
  counting, 492
binomial formula
  generalisation of the, 490
bottom-up parsing, 94
box packing problem, *416*
Burrows-Wheeler-transform, *155*
BWT-Decoder, *157*
BWT-Encoder, *156*

C
canonical
  parser, 120
  parsing table, 119
  set
    kernel, 128
    LALR(1), 122
    LR(0), 128
    LR(1), 116
    merged, 122
Cantor-Zassenhaus algorithm, *241*
  in characteristic 2, 271
Cantor-Zassenhaus-Odd, *242*
C-competitive, *396*
centrally symmetric set, 251
CGS-Orthogonalization, *540*
characteristic, *219*, 230
characteristic equation, *479*, *534*, 536
characteristic polynomial, 271, 272, *534*
character stream, 81
check, 104
Chinese remainder theorem
  for polynomials, *229*, 241, 245, 272
Cholesky-decomposition, *520*, 550*pr*
Cholesky-Decomposition-of-Banded-Matrices, *522*
Cholesky-Method, *521*
Chomsky hierarchy, 18
Chomsky-Normal-Form, *74*
Chomsky normal form, *73*
chrominance, *160*
cipher
  affin linear block, *337*
  block, *337*
  Caesar, *335*
  Hill, *337*
  permutation, *337*
  shift, *334*
  stream, *337*
  substitution, *337*
  transposition, *337*
  Vigenère, *335*
ciphertext, *334*
ciphertext space, *334*
C-(k,h)-competitive, *406*
classical Cournot model, *465*
classical error analysis, 503
Classical-Euclidean, *282*, 330
classical Gram-Schmidt-method, 540
closure, 113, 128
Closure-Item, *114*
Closure-Set-of-Items, *114*
code
  generator, 81, 83
  handler, 82
  optimiser, 81, 83
Coeff-Build, *298*
commutative, *218*
Compensated-Summation, *510*
competitive analysis, *396*
competitive ratio, *396*
competive analysis, 395–428
⋆Compiler⋆, 80, 81, *84*
compilers, 80
complete right transversal, *341*
complexity theory, 364–392
complex numbers, 219
computation tree, 367
computer algebra,
computer algebra systems
  general purpose, 328
  special purpose, 329
concatenation, *13*
condition number, *504*, 511*ex*, 533*ex*, 549*pr*
condition number of eigenvalues, *537*
condition number of the matrix, *504*
configuration, *62*
configuration of the servers, *397*
conflict
  reduce-reduce, 123
  shift-reduce, 123
  shift-shift, 123
congruence
  of polynomials, *224*
consistent norm, 549*pr*
constant subfield, *310*
content, *288*
context dependent
  grammar, 92
context free
  grammar, 92
context-free language
  pumping lemma, 71
context tree, *148*
context-tree weighting algorithm, *149*
Continuous-Equilibrium, *462*
continuous games, *437*
convex set, 251
cost of polynomial operations, *224*
Create-Canonical-Sets, *116*
Create-Rec-Desc, *106*
cryptanalysis, *333*
cryptographic algorithms, *333*
cryptography, *333*
cryptology, 332–363, *333*
cryptosystem, *334*
  symmetric, *334*
cycle free grammar, 93

D
data acknowledgement problem, *403*
data representation
  in computer algebra, 278
DC algorithm, *401*
DCT algorithm, *161*

degree, *222*
derivation, *15*, *310*
  leftmost, 94
    direct, 94
  rightmost, 109
  rules of, 310
derivation tree, *69*
  result of, *70*
derivative
  of a polynomial, 237
determinant, 248, 250, 263
  of a lattice, *251*
deterministic
  finite
    automaton, 85
deterministic finite automaton, *29*
deterministic polynomial time, 350
deterministic pushdown automata, *61*
DFA-EQUIVALENCE, *34*
diagonalisable, *536*
diagonally strictly concave games, *458*
Dickson's lemma, 304, 309*ex*
differential algebra, 309
differential extension, 318
differential extension field, *311*, 316
differential field, *310*
  extensions of, 311
differential operator, *310*
differential subfield, *311*
differentiation, *310*
Diffie-Hellman protocol, 332
Diffie-Hellman protocol, 348*ex*
dimension, 219, *220*
directive, *91*
direct sum, *218*, 220
discrete cosine transform, *161*
Discrete Memoryless Source, *132*
discrete set, 248
discriminant, 300*ex*
distinct degree factorisation, *see* factorisation
DISTINCT-DEGREE-FACTORISATION, *239*
distributive, 217
divisibility
  of polynomials, *223*
division with remainder
  of polynomials, *222*, 224
DIVISION-WITH-REMAINDER
  multivariate, *303*
DMS, *see* Discrete Memoryless Source
DOUBLE-COVERAGE algorithm, *400*
dynamic semantics, 92

E, É
edge
  of finite automaton, *see* transition
  of pushdown automata, *see* transition
eigenvalue, *534*, 550*pr*
elementary
  extension, *316*
  extensions, 316
  functions, 316
ELIMINATE-EPSILON-MOVES, *42*
endomorphism
  Frobenius, 232
entropy, *133*
EQUILIBRIUM, *432*
equivalence relation, 224
equivalent expressions, *50*
equivalent states, *45*
error, 100, 118
  lexical, 82
  repair, 81
  report, 104
  semantic, 83
  syntactic, 82, 99, 120
error bound, *502*, 511*ex*
Euclidean algorithm
  extended, 228
  for polynomials, *227*
EUCLIDEAN-DIVISION-UNIVARIATE-POLYNOMIALS, 282, *300*
Euler function, 340
EXP algorithm, *409*
exponential
  elements, 316
exponential element, *316*
expression
  regular, 85
expression swell
  intermediate, 266
EXTENDED-EUCLIDEAN, *282*, 300*ex*, 330
EXTENDED-EUCLIDEAN-NORMALISED, *285*, 300*ex*
extended grammar, *22*

F
factorisation
  of a polynomial, *237*, 242, 259
    distinct degree, *239*, 248
    square-free, *237*, *238*
Fan's inequality, *439*
fast exponentiation, 238, 240, 242
fast Fourier transform, 224
Fermat's theorem
  little, *231*
Fibonacci number, 486
field, *219*, 230, *339*, *340*
  finite, *230*, 234
  of characteristic zero, *219*
  of elementary functions, *316*, 317, 319
  of transcendental elementary functions, *316*
field extension, 235
field of constants, *310*
file caching problem, *405*
FILL-IN-LR(1)-TABLE, *119*
final state
  of finite automaton, *27*
  of pushdown automata, *61*
  parsing, 101
finite
  automaton
    deterministic, 85
finite automata, 26
  minimization, 45
finite automaton
  complete deterministic, *29*
  deterministic, *29*
  nondeterministic, *27*
  with empty input, 41
FINITE-FIELD-CONSTRUCTION, *240*
finite games, *430*
FIRST, *97*

First$_k$, *95*
First-Fit algorithm, *413*
floating point arithmetic, 505, 507, 509, 550*pr*
floating point arithmetic system, 510
floating point number set, 511*ex*
Follow, *98*
Follow$_k$, 97
forward stable, *505*
From-Cfg-to-Pushdown-Automaton, *66*
function, 355
function mapping
  associative, *355*
fundamental solution, *479*

G
games, *429*
games representable by finite trees, *433*
Gauss, *252*
Gauss' algorithm
  for lattices, 252–256
Gauss-elimination, 327
Gauss lemma
  on primitive polynomials, 260
Gauss-Method, *515*
Gaxpy, *545*
general solution, *478*
generating function, 486
generating functions
  counting binary trees, 492
  operations, 487
generating set
  of vector spaces, *220*
goto table, 118, 124
grammar, *15*
  attribute, 92
  augmented, 110
  context dependent, 92
  context free, 92
  context-free, *18*, 60
  context-sensitive, *18*
  cycle free, 93
  generative, 15
  LALR(1), 124
  left-linear, *78*
  linear, *78*
  LL(k), 95
  LR(1), 112
  LR(k), 110
  normal form, 20
  of type 0,1,2,3, 18
  O-ATG, 81
  operator-, *78*
  phrase structure, *18*
  reduced, 93, 104
  regular, *18*, 85
  two-level, 92
  unambiguous, 93
Gram matrix, 250, 251, 255, 257, 259
Gram-Schmidt orthogonalisation, 254, *255*, 256, 257, 259
graph automorphism, *342*
graph isomorphism, *342*
greatest common divisor
  of polynomials, *227*
Greibach-Normal-Form, *75*
Greibach normal form, *74*
group, 219, *339*
  Abelian, 217–219
  cyclic, 221, 231
  multiplicative, 231
growth factor of pivot elements, *516*
Gröbner basis, 300, *305*, *306*, 307, 330*pr*
  minimal, *307*
  reduced, *307*
Gröbner-basis, 327

H
Hadamard inequality, 255, 257, 264, 268
handle, 93
handler
  code, 82
  list, 81
  source, 81
harmonic algorithm, *413*
harmonic numbers, 489
Hensel's lemma, *264*, 269
Hensel lifting, 264, 267, 268, 270
Hensel-Lifting, *265*
Hermite's method, 323
Hermite-reduction, 311, *313*, 319, 323, 326*ex*, 330*pr*
Hilbert's basis, 304
homomorphism, *218*, 227, 229, 232
honest-verifier perfect zero-knowledge, 359
Horowitz's method, 312
Huffman-algorithm, *139*
hyperexponential element, *316*

I, Í
ideal, *230*, 271
identical permutation, *341*
identity element, 217, 218
  multiplicative, 219
  of a ring, *218*
I-divergence, *137*
ill-conditioned, *504*
image, 271
Inaccessible-States, 30
initial condition, *478*
initial state
  of finite automaton, *27*
  of pushdown automata, *61*
  parser, 100
input alphabet
  of finite automaton, *27*
  of pushdown automaton, *61*
integers, 218, 276, 278
integral
  logarithmic part of, *312*
  rational part of, *312*
Integrate-Logarithmic-Part, *315*
integration
  by parts, *310*
  of elementary functions, 317
  of rational functions, 310
intermediate expression swell, *280*, 307
interpreter, 80
interval scheduling algorithm, *425*
inverse
  additive, *218*
  multiplicative, 219
inverse element for x, *339*

inverse power method, *539*, 543*ex*
inversion of a matrix, 519
invertible element of a ring, *340*
IRREDUCIBILITY-TEST, *240*
irreducible polynomial, *see* polynomial
isomorphism, *218*, 227, 231, 234
  of vector spaces, *220*
iterative refinement, 549*pr*
ITERATIVE-REFINEMENT, *532*

J
JPEG, *160*

K
key space, *334*
keyword, *89*
Kleene's theorem, 50
Kraft's inequality, *135*
Krichevsky-Trofimov estimator, *147*
Kuhn-Tucker conditions, 440

L
Lagrange's theorem, 221
LALR(1)
  canonical set, 122
  grammar, 124
  parser, 81, 121
  parsing
    table, 124
LANDLORD, *406*
language
  complement, 14
  context-free, 18, 69
  context-sensitive, 18
  iteration, 14
  mirror, 14
  of type 0,1,2,3, 18
  phrase-structure, 18
  power, 14
  regular, 18, 26
  star operation, 14
language generated, *16*
languages
  difference, 14
  intersection, 14
  multiplication, 14
  specifying them, 15
  union, 14
lattice, *248*, 269
  full, 249, 250
lattice point, *249*
lattice reduction, 248, 268
lattice vector, 248, *249*
  shortest, 248, 252–254, 257–259
Laurent-series, 323
Lazard-Rioboo-Trager-formula, 314
lazy evaluation, 279
leading coefficient, *281*
leftmost
  derivation, 94
  direct
    derivation, 94
leftmost derivation, *70*
left to right
  parsing, 93
Leibniz rule, *229*
Leibniz-rule, 310
LERC, *387*
lex, 85
LEX-ANALYSE, *88*
LEX-ANALYSE-LANGUAGE, 127
lexical
  analyser, 81, 82
  error, 82
linear combination, 249
LINEAR-HOMOGENEOUS, *484*
linear independence, *220*, 249
linear mapping, *220*, 229, 249, 271
LINEAR-NONHOMOGENEOUS, *498*
Liouville's Principle, 317, *318*, 321, 324
list, 81
list-handler, 81
LIST on-line scheduling model, *419*
LL(1) parser, 81
LL(1)-PARSER, *101*
LL(1)-TABLE-FILL-IN, *100*
LL(k)
  grammar, 95
LLL algorithm
  for factoring polynomials, *268*, 272
LLL-POLYNOMIAL-FACTORISATION, *270*
load, *419*
load balancing routing model, *408*
logarithmic
  derivative, *310*, 316
  element, 316, 319
  extension, *311*, 321
  function, 317
logarithmic integral, 315
lookahead, 90
  operator, *90*
LOVÁSZ-REDUCTION, *256*
LR(0)
  canonical set, 128
  item, 128
LR(1)
  canonical set, 116
    kernel, 128
  grammar, 112
  item, 113
    core, 113
    lookahead symbol, 113
    valid, 113
  parser, 117
  parsing
    table, 118, 119
LR(1)-PARSER, *120*
LR(k)
  grammar, 110
  parsing, 110
lucky prime, *297*
LU-decomposition, *518*
LU-METHOD, *518*
LU-METHOD-WITH-POINTERS, *519*
luminance, *160*

M
M-A algorithm, *389*
machine epsilon, *507*
MACHINE-EPSILON, *508*
mathematical expert systems, 329
matrix

definite, 248
positive definite, 250
MATRIX-EQUILIBRIUM, *450*
matrix games, *447*
MATRIX-PRODUCT-DOT-VERSION, *546*
MATRIX-PRODUCT-GAXPY-VARIANT, *546*
MATRIX-PRODUCT-OUTER-PRODUCT-VARIANT, *547*
MATRIX-PRODUCT-WITH-GAXPY-CALL, *546*
merged canonical set, 122
message, *334*
method of variation of constants, 485
MGS-ORTHOGONALISATION, *541*
Mignotte's theorem, 262, 269
MILLER-RABIN, *351*
minimal polynomial, *226*, 231, 235, 271, 272
Minkowski's Convex Body Theorem, 251, 259
mirror image, *14*
mixed extensions of finite games, *442*
modelling of a source, *141*
MODULAR-GCD-BIGPRIME, *297*
MODULAR-GCD-SMALLPRIMES, *298*
monoid, *339*
commutative, *339*
monomial, *301*
element, *318*
order, 301
monomial ideal, *304*
MOVE-TO-FRONT, *159*
move-to-front code, *158*
MULTISPLITTING-ITERATION, *523*
multivariate polynomial
leading coefficient, *302*
leading monomial, *302*
leading term, *302*
multidegree, *302*

N
N algorithm, *389*
Nash-equilibrium, *429*
neutral element of a group, *339*
NFA-DFA, *33*
NFS$_r$ algorithm, *417*
Noetherian ring, *305*
Noiseless Coding Theorem, *135*
NFA-FROM-REGULAR-GRAMMAR, *40*
nondeterministic finite automaton, *27*
nondeterministic pushdown automaton, *60*
nondeterministic Turing machine with k tapes, *366*
nondeterministic Turing reducibility, *381*
nonlinear complementarity problem, *473*
non-overlapping block Jacobi splitting, *524*
NONPRODUCTIVE-STATES, 30
norm
of a polynomial, *261*
normal basis, *237*
normal form, *281*
Chomsky, *73*
Greibach, *74*
normalized, *506*
NP, 248
NP-completeness, 371
N-PRE-ISO, *381*
numerically stable, *504*, 512*ex*
numerically stable MGS method, 541
numerically unstable, *504*, 511*ex*

O, Ó
O-ATG grammar, 81
off-line algorithm, *395*
offset, 153
OLIGOPOLY-EQUILIBRIUM, *471*
oligopoly game, *465*
One (new), *517*
One (old) flop, *517*
one element of a group, *340*
one-to-one map, 220
on-line algorithm, *395*
on-line LPT, *426*
operation of the CLASSICAL-EUCLIDEAN algorithm, 282*fig*
operation of the PRIMITIVE-EUCLIDEAN algorithm, 288*fig*
operations
on languages, 14
on regular languages, 40
oracle set, *379*
orbit of an element, *341*
order
allowable, 301
monomial, 301, *302*, 308*ex*
of a group element, *221*
order of a group, *339*
orthogonal, *539*
orthogonal vectors, *248*
OUTER-PRODUCT-UPDATE-VERSION „IJ", *545*
OUTER-PRODUCT-UPDATE-VERSION „JI", *545*
overflow, 548*pr*
overlapping block Jacobi multisplitting, *524*

P
packing pattern, *415*
parallelepiped, 251
parser, 82, 92
bottom-up, 94
canonical, 120
final state, 101, 120
initial state, 100
LALR(1), 81
left to right, 93
LL(1), 81, 99
LR(k), 110
start configuration, 120
state, 99, 119
top-down, 93
parsing, 92
table, 100, 118, 119, 124
partial fraction decomposition, 319
particular solution, *478*
payoff, *429*
payoff function, *429*
payoff matrixes, *432*
payoff vector, *429*
perfect secrecy, *339*
permutation, *341*
identical, 341
permutation group, *341*

permutation matrix, *518*
phrase, 93
  simple, 93
pivot element, *515*
plaintext, *334*
plaintext space, *334*
players, *429*
pointwise stabiliser, *341*
polynomial, *221*
  derived, *228*
  irreducible, *223*, 233–235, 240–242, 260, 272
  multivariate, 279, 302
  primitive, *260*
  representation, 279
  square-free, 237, 272
polynomial equations
  equivalence, 308
  finitely many solutions, 308
  number of finite solutions, 308
  solvability, 308
polynomial hierarchy, *381*
pop, 100
power method, 543*ex*
POWER-METHOD, *538*
power series, 279
prefix, *14*
prefix code, *133*
prime field, *219*, 230, 245
Prime Number Theorem, 264
primitive
  part, *288*
  polynomial, *287*
primitive element, *231*, 234
PRIMITIVE-EUCLIDEAN, *288*, 330*pr*
principal ideal domain, *230*
prisoner's dilemma, *430*
PRISONER-ENUMERATION, *432*
production, *15*
program
  assembly language, 83
  source, 81
    language, 80, 81
  target, 81
    language, 80, 81
propagation, 128
proper subword, *14*
pseudo-division, *287*
pseudo-quotient, *287*
pseudo-remainder, *287*
public-key, 349
public-key cryptography, 333
public-key cryptosystem, 333
pumping lemma
  for context-free languages, 71
  for regular languages, 47
pushdown automata, 60
  deterministic, *61*
FROM-PUSHDOWN-AUTOMATON-TO-CF-GRAMMAR, *69*

Q
QR-method, 541, 543*ex*
QR-METHOD, *541*
quadratic sieve, 354
quantisation, *162*, *163*
quotient, *281*, *303*

R
random
  polynomial, 235
randomised on-line algorithm, *396*
RANDOM-SAT, *375*
rank
  of a lattice, *248*
  of a matrix, 248
rational numbers, 219, 278
Rayleigh quotient, 538
read, 113, 128
READ-ITEM, *115*
READ-SET-OF-ITEMS, *115*
real numbers, 219
REC-DESC-STAT, *106*
REC-DESC-STAT1, *107*
REC-DESC-STAT2, *107*
RECURRENCE, *500*
recurrence equation, 478
  linear, 479, 484, 491, 497
  linear homogeneous, 479, 484
  linear nonhomogeneous, 484, 497, 498
  solving with generating functions, 490
recursive-descent method, 104
RECURSIVE-SUMMATION, *509*
reduced grammar, 93, 104
reduce-reduce conflict, 123
redundancy, *151*
regular
  expression, 50, 85
  grammar, 85
  language, 26
  operation, *14*
regular expression, *50*
REGULAR-GRAMMAR-FROM-DFA, *38*
relative error, *502*, 511*ex*, 533*ex*
relative error bound, 507, 511*ex*
relatively prime
  polynomials, *223*, 229
release time, *420*
reliable,
remainder, *281*, *303*
residual error, *525*
residue classes, 218, 219
residue class ring, *224*
residue theorem, 498
resultant, *290*, 320, 324
  Sylvester form, 291
resultant method, 289
retrieval cost, *405*
reversal, *14*
right coset, *341*
rightmost
  derivation, 109
  substitution, 109
ring, *217*, 232, 271, *339*
  commutative, *340*
  Euclidean, 223
Risch algorithm, *319*, 326
  exponential case, 323
  logarithmic case, 319
Risch differential equation, *325*
Risch integration algorithm, 315, 317
root
  of a polynomial, *223*, 233, 234

Rothstein-Trager integration algorithm, *313*
Rothstein-Trager method, 320, 324
RSA protocol, 349*fig*
Run-Length-Code, *164*
run-length coding, *163*
run-time semantics, 92

S

saddle points, *432*
SAXPY, *544*
scalar product, 248
scanner, 81, 82
secret-key agreement problem, 332
semantic
  analyser, 81, 83, 92
  error, 83
semantics, 92
  dynamic, 92
  run-time, 92
  static, 92
semigroup, 217
sensitivity of a problem, *525*
sentence, 92, 93
sentential form, 92
series of symbols, 82, 85
shadow, 136
Shannon-Fano-algorithm, *138*
Shannon-Fano-Elias-code, *138*
SHELF algorithms, *416*
shift cipher, *334*
shifted QR-method, 543*ex*
SHIFTED-QR-METHOD, *543*
shift-reduce conflict, 123
shift-shift conflict, 123
similarity transformation, *535*
simple
  phrase, 93
simplification of expressions, *308*
simultaneous strategy vector, *429*
size, *405*
Skeel-norm, *528*
sliding window, 153
SOLUTION-OF-BANDED-UNIT-LOWER-TRIANGULAR-SYSTEM, *522*
SOLUTION-OF-BANDED-UPPER-TRIANGULAR-SYSTEM, *522*
source
  handler, 81
  language program, 80, 81
  program, 81
S-polynomial, *305*
spontaneously generated, 128
SQUARE-AND-MULTIPLY, *347*, 348*ex*
square-free factorisation, *see* factorisation
SQUARE-FREE-FACTORISATION, *238*
square-free polynomial, *see* polynomial
square lattice, 250
stabiliser, *341*
  pointwise, 341
stability (or sensitivity) of an algorithm, *525*
stack alphabet
  of pushdown automaton, *61*
standard word, 89
start symbol, *15*
  of pushdown automata, *61*
state
  inaccessible, 29
  nonproductive, 30
  of finite automaton, *27*
  of pushdown automaton, *61*
  parser, 99
static
  semantics, 92
strategies, *429*
strategy set, *429*
strip packing problem, *416*
strong generator of a group, *341*
strong nondeterministic Turing reducibility, *381*
strong NPOTM, *391*
subalgebra, 243
subdeterminant, 257
subfield, 233
subgroup, 248
  additive, 220, 230
  multiplicative, 221
subgroup of a group, *339*
subnormal number, *511*, 512*ex*
subring, 243
subspace, *220*, 243, 250
  invariant, 271
substitution
  rightmost, 109
subword, *14*
suffix, *14*
Sylvester matrix, 263
symbol
  actual, 99, 104
  table, 81, *91*
symbolic
  computation, 275
  integration, 309
symmetric matrix games, *450*
syntactic
  analyser, 81, 82
  analysis, 92
  error, 82, 99, 120
syntax, 92
syntax tree, *70*
synthesis, 83
system
  monoalphabetic, 337
  polyalphabetic, 337

T

table
  parsing, 100, 118, 119, 124
target
  language program, 80, 81
  program, 81
terminal symbols, *15*
THE-LU-DECOMPOSITION-OF-BANDED-MATRIX, *522*
the mathematical model of routing, *408*
throughput routing model, *408*
thumb rule, *527*
TIME on-line scheduling model, *420*
top-down parsing, 93
tower of stabilisers, *341*
Towers of Hanoi, 485*ex*

trace
  in a finite field, 271
  of a linear mapping, 271, 272
transcendent
  element, *316*
transcendental
  element, 319
  elementary extension, *316*
transition
  of finite automaton, *27*
  of pushdown automata, *61*
triangular lattice, 250, 259
Turing reducibility, *381*
two-level grammar, 92

U, Ú
UDC, *see* uniquely decipherable code
unambiguous grammar, 93
unique factorisation, 223
uniquely decipherable code, *133*
unit, *223*
unit (upper or lower) triangular, *518*
ELIMINATE-UNIT-PRODUCTIONS, *19*, *20*
unit roundoff, *507*
UPDATE-CONTEXT-TREE, *150*
upper bound, 364
upper Hessenberg form, *542*

V
valid
  LR(1)-item, 113
value of matrix games, *448*
variables, *15*
variety, *301*
vector packing problem, *416*
vector space, *219*, 227, 230, 233, 248, 271
viable prefix, 112
Vigenère square, *335*
volume, 251
von Mieses method, 539

W
weak competitive ratio, *396*
weakly C-competitive, *396*
weakly stable, *528*
WEAK-REDUCTION, *255*, 256
well-conditioned, *504*
white space, *86*
word, *13*
  power, *13*
work function, *400*
WORK-FUNCTION algorithm, *400*
WRITE-PROGRAM, *105*

Y
yacc, 110
YCbCr-transform, *160*

Z
zero, *218*, 219
zero divisor, 271
zero element of a group, *340*
zero-sum games, *432*
Ziv-Lempel-coding, *153*
Z-transform method, 497

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of their birth and death.

A, Á
Abramov, Sergey Alexandrovich, 327
Adleman, Leonard, 349
Agrawal, Manindra, 350, 552
Ahlswede, Rudolf, 167, 552
Aho, Alfred V., 78, 129, 552
Ajtai, Miklós, 248, 273, 552
Akritas, A. G., 552
Albers, Susanne, 427, 552
Alberti, Leon Battista (1404–1472), 335
Althöfer, Ingo, 2, 10, 557
Arbib, M. A., 558
Arvind, V., 392, 552
Aspnes, James, 428, 552
Assing, Sigurd, 392
Asteroth, Alexander, 79, 552
Aubin, Jean-Pierre, 476, 552
Auchmuty, Giles, 531, 549
Awerbuch, Baruch, 428, 552
Azar, Yossi, 428, 552

B
Babai, László, 357, 363, 552
Baier, Christel, 79, 552
Baker, S. Brenda, 417, 428, 552
Bal, Henri E., 130, 555
Balkenhol, Bernhard, 167, 552, 557
Balogh, Ádám, 10, 428
Bals, Helge, 427, 552
Banach, Stephan (1892–1945), 437
Barett, R., 551, 553
Batterson, Steve, 551, 553
Bauer, F. L., 528, 536, 550
Belényesi, Viktor, 2, 10
Bell, T. C., 553
Benczúr, A. András, 10
Bender, E., 213, 553
Berlekamp, Elwyn Ralph, 241, 242, 246, 266, 327
Bernardin, Laurent, 555
Berry, M., 551, 553
Beygelzimer, A., 553
Björck, Ake, 541
Bodon, Ferenc, 9
Boneh, D., 553
Borchert, B., 392, 553
Borodin, Allan, 553
Bronstein, Manuel, 327
Brookshear, J. G., 553
Brouwer, Luitzer Egbertus Jan (1881–1966), 437, 476, 553
Brueggeman, T., 553
Buchberger, Bruno, 276, 305–307, 330, 553
Bunch, James R., 528
Burrows, Michael, 132, 553

C
Cai, Ning, 557
Canfield, R., 213, 553
Cantor, David G., 241, 246
Carroll, J., 553
Carter, J. L., 392, 553
Cauchy, Augustin-Louis (1789–1857), 456
Caviness, Bob Forrester, 331, 553
Chan, T. F., 553
Chiarella, Carl, 477, 560
Cho, Yookun, 428, 553
Cholesky, André-Louis (1875–1918), 516, 518, 520–522, 550, 551
Chomsky, Noam, 13, 18
Christenswn, S. M., 553
Chrobak, Marek, 400, 427, 553
Cleary, J. G., 553, 561
Cocke, J., 129
Coffman, Ed G., Jr., 428, 553
Cohen, Arjeh M., 331, 553
Collins, Georges Edwin, 327
Cook, Stephen Arthur, 554
Cooper, Keith D., 130, 554
Coppo, M., 556
Cormen, Thomas H., 167, 554
Cournot, Antoine Augustin (1801–1877), 465, 477
Cover, Thomas M., 554
Cramer, Gabriel (1704–1752), 280
Czapor, S. R., 555

CS
Csirik, János, 10, 418, 428, 554

Csiszár, Imre, 168, 554
Csörnyei, Zoltán, 2, 9

D
Dantsin, E., 391, 554
Davenport, J. H., 554
Demers, Alan, 428, 556
Demetrovics, János, 2, 10
Demmel, J., 553, 554
De Remer, F. L., 129
Diament, B., 550
Dickson, Leonard Eugene, 304, 309
Diffie, Whitfield, 332
Dobrushin, Roland Lvovitsch (1929–1995), 213, 554
Donato, J., 553
Dongarra, Jack, 553
Dooly, R. Dan, 403, 427, 554
Dósa, György, 554
Dömösi, Pál, 2, 10
Du, Ding-Zhu, 554
Dueck, Gunter, 557

E, É
Earley, J., 129
Effros, Michelle, 554
Eijkhout, V., 553
Elaydi, Saber N., 501, 554
Elek, István, 9
Elias, Peter, 138
Elsner, Ludwig, 535, 536
El-Yaniv, Ran, 553
Englert, Burkhard, 2, 10
Erdélyi, Gábor, 392
Euclid of Alexandria (i.e. 365–300), 282
Euler, Leonhard (1707–1783), 221, 340

F
Fan, Ky, 438, 439, 464, 476
Fano, Robert M., 137
Farkas, Gábor, 9
Fenner, S., 554
Fermat, Pierre, de (1601–1655), 231
Fiat, Amos, 400, 427, 428, 552, 554
Fibonacci, Leonardo Pisano (1170–1250), 478, 480, 486, 487, 489
Fischer, C. N., 129, 554
Flajolet, Philippe, 501, 560
Fleischer, Rudolf, 428, 554
Forgó, Ferenc, 476, 554
Fortnow, L., 554
Fox, L., 551
Fridli, Sándor, 10
Frobenius, Ferdinand Georg (1849–1917), 232, 328
Frommer, Andreas, 551, 554
Fülöp,Zoltán, 10

G
Gaál, István, 554
Gács, Péter, 2, 9, 214, 555
Gailly, J. L., 558
Gál, Anna, 2, 10, 555
Gál, István, 273
Galántai, Aurél, 2, 9
Gallager, Robert G., 213, 554
Garey, Michael R., 391, 428, 554, 556
Gathen, Joachim von zur, 273, 555
Gauss, Johann Carl Friedrich (1777–1855), 252, 279, 287, 296, 327, 513, 515, 517–519, 521, 523, 542
Gécseg, Ferenc, 78, 555
Geddes, Keith Oliver, 331, 555
Gerhard, Jürgen, 273, 555
Gersgorin, S. A., 535
Giammarresi, Dora, 78, 555
Givens, Wallace J., 539
Goerdt, A., 554
Goldman, A. Sally, 403, 427, 554
Goldreich, Oded, 363, 555
Goldwasser, Shaffi, 555, 558
Gonda, János, 2, 10
Gonnet, Haas Gaston Henry, 331, 555
Goodwin, E. T., 551
Goos, Gerhard, 129, 561
Graham, Ronald Lewis, 419, 428, 501, 555, 556
Gram, Jorgen Pedersen (1850–1916), 250, 254, 539, 541
Greene, Daniel H., 501, 555
Gries, David, 129, 555
Grossman, R., 555
Gröbner, Wolfgang Anton Maria, 276, 300, 305–307, 309, 327
Grune, Dick, 130, 555
Gruntz, Dominik, 555

GY
Gyires, Tibor, 2, 10

H
Hadamard, Jacques Salomon (1865–1963), 255
Hadley, George F., 477, 555
Hageman, L. A., 551, 555
Han, Te Sun, 555
Hankerson, D., 555
Harper, D., 555
Harris G. A., 555
Harrison, Michael A., 78, 556
Hartmanis, Juris, 555
He, Yong, 554
Hearn, Anthony Clern, 331, 556
Hellman, Martin E., 332
Hemaspaandra, Lane A., 391, 553, 556
Hemmendinger, David, 555
Hensel, Kurt Wilhelm Sebastian (1861–1913), 264, 327
Herken, R., 555
Hermite, Charles (1822–1901), 259, 310–312, 323, 330
Hessenberg, Gerhard, 542, 551
Hilbert, David, 530, 533, 549
Hilbert, David (1862–1943), 304, 305, 328
Hironaka, Heisuke, 330
Hodginson, D., 555
Hoffman, C., 392, 556
Homan, C., 553, 556
Hopcroft, John E., 78, 556
Horowitz, Ellis, 312
Horváth, Zoltán, 9
Householder, Alston Scott, 539

Hörmann, Anikó, 2, 10
Huang, Ming-Deh, 273
Huffman, David A. (1925–1999), 139, 166, 556
Hungerford, Thomas W., 273, 556
Hunter, Robin, 129, 556
Håstad, Johan, 555

I, Í
IeC Đemel, J., 550
Illés, Tibor, 9
Imreh, Csanád, 2, 9, 556
Isoda, K., 558
Iványi, Anna, 2, 10
Iványi, Antal, 2, 10, 11, 428, 556
Ivanyos, Gábor, 2, 9, 10
Iwama, K., 391, 556

J
Jacobi, Carl Gustav Jacob (1804–1851), 440, 523, 524
Jacobs, Ceriel J. H., 130, 555
Jankowski, T., 533
Járai, Antal, 2, 9, 10
Jeney, András, 2, 9
Johnson, David S., 391, 428, 556
Johnson, P. D., 555
Jordan, Camille, 536

K
Kahan, W., 510
Kakutani, Shizou, 437, 476, 556
Kaliski, B. Jr., 556
Kaltofen, Erich L., 273
Kannan, Ravindran, 554
Karamardian, S., 556
Karian, Z. A., 556
Karlin, Anna R., 427, 557
Karloff, J. Howard, 427, 553
Karp, Richard M., 557
Kása, Zoltán, 2, 9, 501, 557
Kasami, T., 129
Kátai, Imre, 9
Katsányi, István, 2
Kayal, Neeraj, 350, 552
Kelley, D., 557
Kenyon, Claire, 427, 557
Kern, W., 553
Kfoury, A. J., 558
Khachatrian, Levon (1954–2002), 167, 552, 557
Kim, J. H., 213, 557
Kiss, Attila, 2, 10
Kleene, Stephen C., 50
Kleinberg, J., 554
Klyuyev, v. v., 517
Knuth, Donald Ervin, 129, 331, 501, 555, 557
Ko, Ker-I, 554
Kobayashi, Kingo, 555
Kokovkin-Shcherbak, N., 517
Komlós, János, 552
Koutsoupias, Elias, 400, 427, 557
Kovács, Attila, 2, 9, 331, 557
Kowalski, Dariusz, 2
Kozen, D. C., 557
Köbler, J., 363, 392, 557
Körner, János, 168, 554
Krichevsky, R. E., 147, 557
Kuhn, Harold W., 440, 476, 557
Kulkarni, S. R., 554
Kung, H. T., 224
Kurtz, Stefan, 552, 554, 557
Kurur, P., 392, 552
Kutta, Wilhelm Martin (1867–1944), 457
Kuznetsov, A. V., 557

L
Labahn, G., 555
Ladner, Richard E., 391, 557
Lagrange, Joseph Louis (1736–1813), 221, 455
Lakatos, László, 9
Landau, Edmund Georg Hermann (1877–1938), 289, 297, 298, 300
Langdon, G. G., Jr., 557
Langendoen, Koen G., 130, 555
Larmore, Lawrence, 400, 427, 553
Laurent, Pierre Alphonse (1813–1854), 297, 309, 323
Lawson. M. V., 557
Lazard, Daniel, 314
LeBlanc, R. J., 129, 554
Leibniz, Gottfried Wilhelm (1646–1716), 229, 310
Leiserson, Charles E., 167, 554
Lempel, Abraham, 132, 167, 561
Lenstra, Arjen Klaas, 248, 268, 273, 327, 328, 557
Lenstra, Hendrik Willem, Jr., 248, 268, 273, 327, 557
Leonardi, Stefano, 427, 557
Leopold, Claudia, 2, 10
Lidl, Rudolf, 273, 557
Lie, Marius Sophus, 327
Linz, P., 557
Liouville, Joseph, 276, 315, 318, 321, 324
Locher, Kornél, 2, 10
Long, Darrell, 553
Lothaire, M., 78, 557
Louden, Kenneth C., 130, 558
Lovász, László, 8, 248, 256, 268, 273, 327, 501, 557, 558
Lukács, András, 9
Lunel, Sjoerd Verduyn, 553
Lupanov, Oleg Borisovitsch, 213, 558
Lynch, Nancy Ann, 391, 557

M
Mak, Ronald, 129, 558
Malajovich, Diement G., 550, 554
Malewicz, Grzegorz, 2, 10
Manasse, Mark, 397, 427, 558
Mangasarian, Olvi L., 558
Manna, Zohar, 78, 558
Marcelin, M. W., 560
Martos, Béla, 476, 558
Mathon, R., 388, 558
Mayer, János, 2, 10
McGeoch, Lyle, 397, 427, 558
Meduna, A., 558
Meskó, Attila, 10
Meyer, A., 392, 558
Micali, Silvio, 555, 559

Micciancio, D., 558
Mickens, Ronald Elbert, 501, 558
Mignotte, Maurice, 261, 289, 297, 298, 300, 331, 558
Miklós, István, 2, 10
Miller, Gary L., 351, 558
Miller, Raymond E., 557
Mills, H., 558
Minkowski, Hermann (1864–1909), 251, 259
Mishra, Bhubaneswar, 331, 558
Moler, Cleve B., 517
Moll, R. N., 558
Monien, B., 558
Montalbano, Rosa, 78, 555
Moore, J., 558
Moran, S., 552
Morgenstern, Oscar (1902–1976), 477, 558
Motwani, Rajeev, 78, 213, 556, 558
Muchnick, Steven S., 130, 558
Munro, J., 556

N
Nagy, Marianna, 9
Nash, John F., Jr., 429, 476, 558
Neal, R. M., 561
Nelson, Mark, 558
Neumann, John, von (1903–1957), 454, 477, 558
Niederreiter, Harald, 273, 557
Nikaido, Hukukane, 476, 558
Noether, Amalia Emmy (1882–1935), 305
Noga, John, 556

O, Ó
Odlyzko, Andrew Michael, 331, 558
Oettli, W., 529, 532
Ogihara, M., 556
Okuguchi, Koji, 477, 558
Orosz, László, 10
Ortyukov, S. I., 213, 554
Ostrowski, Alexander R., 535, 536

P
Pan, Victor Y., 273
Papadimitriou, Christos H., 363, 391, 400, 427, 554, 557, 559
Parlett, Beresford, 541
Pasanen, Kari, 556
Pasco, R., 167, 559
Pataricza, András, 10
Patashnik, Oren, 501, 555
Paturi, R., 391, 559
Pavelle, R., 559
Payne, Tom, 427, 553
Peák, István (1938–1989), 78, 555
Pethő, Attila, 2, 10
Pinsker, Mark S. (1925–2003), 557, 559
Pintér, Miklós Péter, 10
Pippenger, Nicholas, 559
Pittman, Thomas, 129, 559
Plotkin, Serge, 428, 552
Pollard, John Michael, 328, 559
Polyak, Roman A., 561
Pomerance, Karl, 328
Porta, Giovanni (1675–1755), 335
Pozzo, R., 553
Prager, W., 529, 532
Primak, M. E., 561
Pudlák, P., 559

R
Rabani, Yuval, 400, 427, 554
Rabi, M., 559
Rabin, Michael O., 351, 559
Rackoff, C., 555
Raghavan, P., 213, 554, 558
Ralston, Anthony, 555
Randall, Dana, 427, 557
Rao, R., 559
Ravid, Yiftach, 400, 427, 554
Rayleigh, John William Strutt, 538
Recski, András, 2, 10
Reif, John, 555
Reilly, Edwin D., 555
Reischuk, Rüdiger, 559
Rioboo, Renaud, 314
Risch, Robert, 310, 315, 317–319, 325, 327
Rissanen, J. J., 167, 559
Ritt, Joseph Fels, 331, 559
Rivest, Ronald Lewis, 167, 349, 554, 559
Robinson, Julia, 476, 559
Robshaw, M., 556
Romine, C., 553
Rónyai, Lajos, 2, 9, 10
Rosen, J. B., 477, 559
Rothe, Jörg, 2, 9, 353, 354, 391, 553, 556, 559
Rothstein, Michael, 313, 320, 324, 559
Rozenberg, Grzegorz, 78, 559
Runge, Carl David Tolmé (1856–1927), 457

S
Sahni, Sartaj, 428, 553
Saks, M., 559
Sali, Attila, 2, 10
Salomaa, Arto, 78, 559
Salomon D., 559
Sárközy, András, 557
Saxena, Amitabh, 556
Saxena, Nitin, 350, 552
Sayood, K., 559
Schmelz, B., 559
Schmidt, Erhard (1876–1959), 254, 539, 541
Schneider, Csaba, 10
Schönhage, Arnold, 224, 273
Schöning, Uwe, 391, 392, 554, 557, 559, 560
Schwartz, Jacob Theodore, 456
Schwarz, S. Jerald, 417, 428, 552
Schwarz, Stefan, 2, 10
Scott, D. Stephen, 403, 427, 554
Sedgewick, Robert, 501, 560
Seidel, Philipp Ludwig, von (1821–1896), 523
Selman, Alan L., 391, 557, 560
Sethi, Ravi, 129, 552
Sgall, Jirí, 428, 560
Shamir, Adi, 349, 560
Shannon, Claude Elwood (1916–2001), 137, 213, 560
Shapiro, Harold N., 560
Sherman, A., 559
Shmoys, David B., 428, 560
Shoup, Victor J., 273
Shparlinski, Igor E., 273, 560
Shtarkov, Yuri M., 142, 167, 561

Shvartsman, Alexander Allister, 2, 10
Sidló, Csaba, 9
Sieveking, Malte, 224
Sima, Dezső, 2
Simon, M., 560
Simons, G., 558
Simovoci, Dan A., 560
Singh, Simon, 560
Sipser, M., 555
Sipser, Michael, 78, 213, 560
Siret, Y., 554
Skeel, Robert D., 528, 533
Sleator, Daniel, 397, 427, 558, 560
Smart, Nigel P., 273, 560
Smith, Henry John Stephen, 328
Solovay, R., 560
Sorenson, Paul G., 129, 561
Spakowski, Holger, 392
Speckenmeyer, 558
Spielman, Daniel A., 213, 560
Stamoulis, George D., 559
Stein, Clifford, 167, 554
Sterrett, A., 556
Stinson, Douglas, 354, 560
Stockmeyer, L., 392, 560
Stone, H., 558
Storjohann, Arne, 273
Stoyan, Dietrich, 392
Stoyan, Robert, 392
Strassen, Volker, 224, 273, 328, 560
Straubing, H., 560
Sudkamp, Thomas A., 560
Swinnerton-Dyer, Peter, 267
Sylvester, James Joseph (1814–1897), 263, 290–292

SZ
Szántai, Tamás, 2, 10
Szeidl, László, 9
Szemerédi, Endre, 552
Szép, Jenő (1920–2004), 476, 554
Szidarovszky, Ferenc, 2, 9, 476, 477, 554, 558, 560
Szirmay-Kalos, László, 2, 10
Sztrik, János, 2

T
Tamaki, S., 391, 556
Tamm, Ulrich, 2, 9
Tarjan, Robert Endre, 560
Tarski, Alfred (1902–1983), 437, 476, 560
Taubman, D. S., 560
Taylor, Brook, 503
Taylor, M. G., 213, 561
Tejfel, Máté, 9
Telek, Miklós, 9
Tenney, R. L. , 560
Terlaky, Tamás, 9
Thatcher, J. W., 557
Thomas, J. A., 554
Tjalkens, Tjalling J., 142, 167, 561
Toda, S., 557
Torán, J., 392, 557
Torczon, Linda, 130, 554
Tournier, E., 554
Trager, Barry Marshall, 313, 314, 320, 324
Tremblay, Jean-Paul, 129, 561
Trithemius, Johannes (1492–1516), 335
Trofimov, V. K., 147, 557
Tsitsiklis, John N., 559
Tucker, Albert W., 440, 476, 557
Tucker, Allen B., 558
Turing, Alan (1912–1954), 551, 561

U, Ú
Uhl, J. J., 561
Ullman, Jeffrey David, 78, 79, 129, 428, 552, 556

V
van der Waerden, Bartel Leendert, 327
van Gastel, Leendert, 553
van Vliet, André, 428, 561
Varga, László, 2
Verdú, Sergio, 554
Vestjens, Arjen, 426, 428, 561
Vida, János, 2
Vigenère, Blaise, de (1523–1596), 335
Vilenkin, Naum Yakovlevich (1920–1992), 501, 561
Vishwanathan, Sundar, 427, 553
Visweswariah, K., 554
Vizvári, Béla, 2, 9, 10
von Mieses, Richard (1853–1953), 537, 539
von Neumann, John (1903–1957), 558
von zur Gathen, Joachim, 331, 555
Vorst, H., van der, 553
Vörös, Veronika, 2, 10
Vu, V. H., 213, 557

W
Waarts, Orli, 428, 552
Wagner, Klaus W., 391, 561
Wahl, Michaela, 428, 554
Waite, William M., 129, 561
Wallace, G. K., 561
Watanabe O., 559
Watkins, D. S., 561
Wechsung, Gerd, 391, 561
Wegener, Ingo, 557
Wegman, M. N., 392, 553
Wein, Joel, 428, 560
Welsh, Dominic, 561
Wheeler, David J., 132, 553
Wigderson, Avi, 555
Wilkinson, James H., 516, 517, 529, 551, 561
Willems, Frans M. J., 142, 167, 561
Williamson, David P., 428, 560
Winkler, Franz, 331, 561
Witten, I. H., 553, 561
Woeginger, J. Gerhard, 418, 428, 554
Wooff, C., 555
Wozniakowski, T., 533

Y
Yakowitz, Sidney, 477, 560
Yao, C. C. Andrew, 561
Young, David M., 555, 561
Young, L. A., 551
Young, Neal, 405, 427, 561
Younger, D. H., 129

Yun, David Y. Y., 273

Z
Zane, F., 559
Zassenhaus, Hans Julius (1912–1991), 241, 246, 266, 327
Zehendner, Eberhard, 2, 10
Zhang, Zhen, 557
Ziv, Jacob, 132, 167, 561
Zuhovitsky, S. I., 477, 561

The book *Algorithms of Informatics* consists of two volumes: *Volume 1. Foundations* and *Volume 2. Applications.*

The first volume contains twelve chapters. These chapters are divided into three parts. The chapters of the first part are connected with automata: *Automata and Formal Languages* (written by Zoltán Kása, Babeş-Bolyai University of Cluj-Napoca), *Compilers* (Zoltán Csörnyei, Eötvös Loránd University), *Compression and Decompression* (Ulrich Tamm, Chemnitz University of Technology Commitment), and *Reliable Computations* (Péter Gács, Boston University).

The second part contains the following chapters having algebraic character: *Algebra* (due to Gábor Ivanyos, and Lajos Rónyai, Budapest University of Technology and Economics), *Computer Algebra* (Antal Járai, Attila Kovács, Eötvös Loránd University), *Cryptology* and *Complexity Theory* (Jörg Rothe, Heinrich Heine University).

The chapters of the third part have numeric character: *Competitive Analysis* (Csanád Imreh, University of Szeged), *Game Theory* (Ferenc Szidarovszky, The University of Arizona) and *Scientific Computations* (Aurél Galántai, András Jeney, University of Miskolc).

The chapters of the first volume were validated by Gábor Ivanyos, Lajos Rónyai, András Recski, and Tamás Szántai (Budapest University of Technology and Economics), Sándor Fridli, János Gonda, and Béla Vizvári (Eötvös Loránd University), Pál Dömösi, and Attila Pethő (University of Debrecen), Zoltán Fülöp (University of Szeged), Anna Gál (University of Texas), János Mayer (University of Zürich).

The first volume contains verbal description, pseudocode and analysis of about 100 algorithms, and also about 100 figures and 110 examples illustrating how the algorithms work. Each section ends with exercises and each chapter ends with problems. We have included about 140 exercises and 30 problems.

We have supplied an extensive bibliography, in the section *Chapter notes* of each chapter. The web site of the book contains the maintained PDF version of the bibliography in which the names of the authors, journals and publishers are usually active links to the corresponding web site (the living elements of the printed bibliography are underlined).

http://elek.inf.elte.hu/EnglishBooks

3600 HUF

ISBN 978-963-87596-1-0