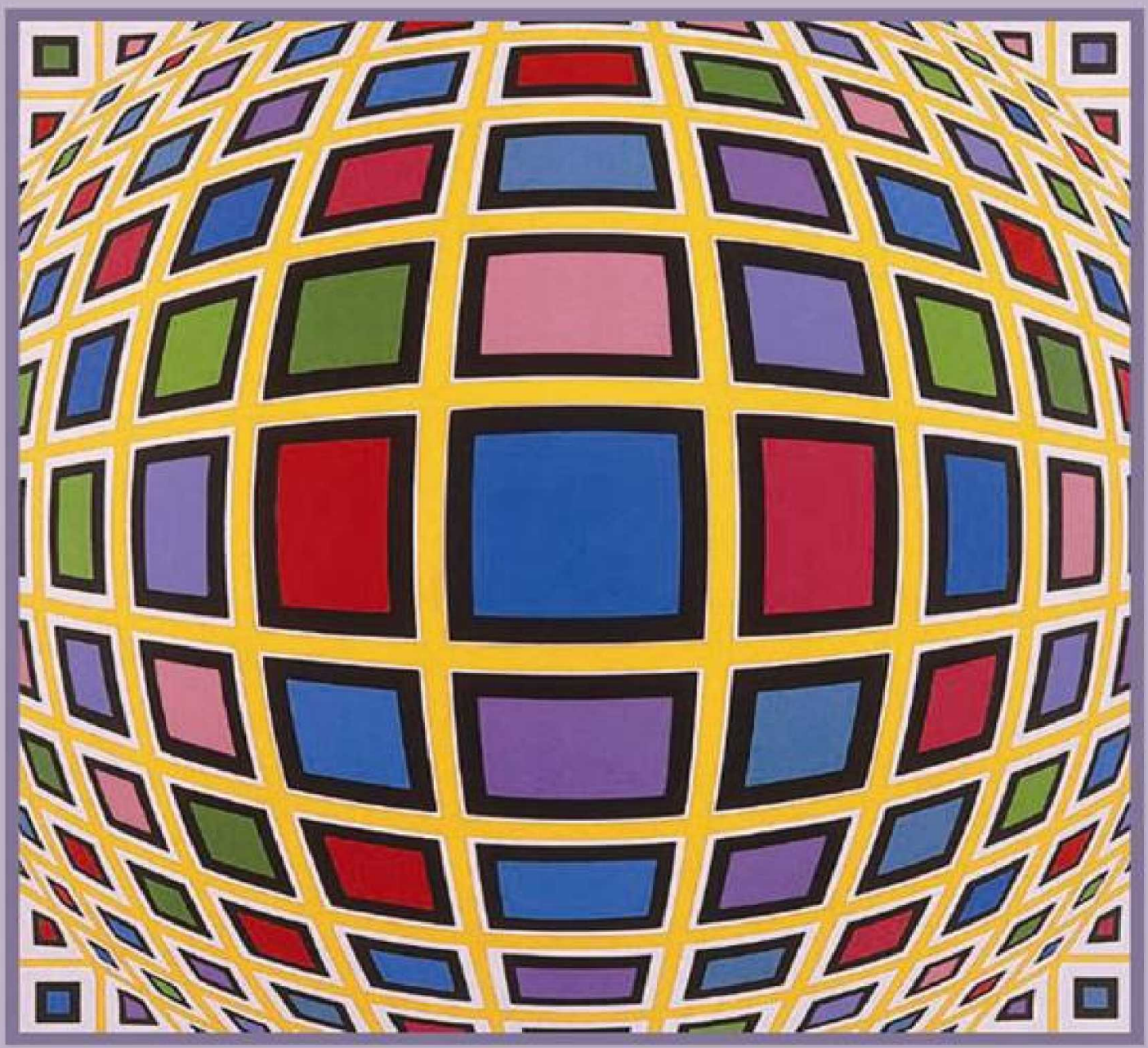
ALGORITHMS of Informatics



volume 2

ALGORITHMS

OF INFORMATICS

Volume 2

APPLICATIONS

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 Balogh, 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), Gábor Ivanyos (6), János Gonda (7), Lajos Rónyai (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: Anikó Hörmann, László Orosz, Miklós Péter Pintér, Csaba Schneider, Veronika Vöröss

Cover art: Victor Vasarely, *Kubtuz*, 1983. With the permission of © HUNGART, 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 2: ISBN 978-963-87596-2-7;
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 **586**
Introduction **587**

IV. COMPUTER NETWORKS **590**
13. Distributed Algorithms (Burkhard Englert, Dariusz Kowalski, Grzegorz Malewicz, Alexander Allister Shvartsman) **591**
13.1. Message passing systems and algorithms 592
13.1.1. Modeling message passing systems 592
13.1.2. Asynchronous systems 592
13.1.3. Synchronous systems 593
13.2. Basic algorithms 594
13.2.1. Broadcast 594
13.2.2. Construction of a spanning tree 595
13.3. Ring algorithms 599
13.3.1. The leader election problem 599
13.3.2. The leader election algorithm 600
13.3.3. Analysis of the leader election algorithm 603
13.4. Fault-tolerant consensus 605
13.4.1. The consensus problem 606
13.4.2. Consensus with crash failures 607
13.4.3. Consensus with Byzantine failures 608
13.4.4. Lower bound on the ratio of faulty processors 608
13.4.5. A polynomial algorithm 608
13.4.6. Impossibility in asynchronous systems 610
13.5. Logical time, causality, and consistent state 611
13.5.1. Logical time 611
13.5.2. Causality 613
13.5.3. Consistent state 615
13.6. Communication services 618
13.6.1. Properties of broadcast services 618
13.6.2. Ordered broadcast services 619
13.6.3. Multicast services 623
13.7. Rumor collection algorithms 624

13.7.1. Rumor collection problem and requirements . . . 624
13.7.2. Efficient gossip algorithms . . . 625
13.8. Mutual exclusion in shared memory . . . 632
13.8.1. Shared memory systems . . . 632
13.8.2. The mutual exclusion problem . . . 633
13.8.3. Mutual exclusion using powerful primitives . . . 634
13.8.4. Mutual exclusion using read/write registers . . . 635
13.8.5. Lamport's fast mutual exclusion algorithm . . . 638
14. Network Simulation (Tibor Gyires) . . . 643
14.1. Types of simulation . . . 643
14.2. The need for communications network modelling and simulation . . . 644
14.3. Types of communications networks, modelling constructs . . . 646
14.4. Performance targets for simulation purposes . . . 648
14.5. Traffic characterisation . . . 651
14.6. Simulation modelling systems . . . 659
14.6.1. Data collection tools and network analysers . . . 659
14.6.2. Model specification . . . 659
14.6.3. Data collection and simulation . . . 659
14.6.4. Analysis . . . 660
14.6.5. Network Analysers – Application Characterisation Environment (ACE) . . . 661
14.6.6. Sniffer . . . 668
14.7. Model Development Life Cycle (MDLC) . . . 668
14.8. Modelling of traffic burstiness . . . 674
14.8.1. Model parameters . . . 679
14.8.2. Implementation of the Hurst parameter in the COMNET modelling tool . . . 680
14.8.3. Validation of the baseline model . . . 682
14.8.4. Consequences of traffic burstiness . . . 686
14.8.5. Conclusion . . . 689
14.9. Appendix A . . . 690
14.9.1. Measurements for link utilisation . . . 690
14.9.2. Measurements for message delays . . . 690
15. Parallel Computations (Antal Iványi, Claudia Leopold) . . . 702
15.1. Parallel architectures . . . 704
15.1.1. SIMD architectures . . . 704
15.1.2. Symmetric multiprocessors . . . 705
15.1.3. Cache-coherent NUMA architectures: . . . 706
15.1.4. Non-cache-coherent NUMA architectures: . . . 707
15.1.5. No remote memory access architectures . . . 707
15.1.6. Clusters . . . 707
15.1.7. Grids . . . 708
15.2. Performance in practice . . . 708
15.3. Parallel programming . . . 712
15.3.1. MPI programming . . . 713

15.3.2. OpenMP programming . . . 716
15.3.3. Other programming models . . . 718
15.4. Computational models . . . 719
15.4.1. PRAM . . . 719
15.4.2. BSP, LogP and QSM . . . 720
15.4.3. Mesh, hypercube and butterfly . . . 721
15.5. Performance in theory . . . 723
15.6. PRAM algorithms . . . 727
15.6.1. Prefix . . . 728
15.6.2. Ranking . . . 733
15.6.3. Merge . . . 735
15.6.4. Selection . . . 739
15.6.5. Sorting . . . 744
15.7. Mesh algorithms . . . 747
15.7.1. Prefix on chain . . . 747
15.7.2. Prefix on square . . . 748
16. Systolic Systems (Eberhard Zehendner) . . . 751
16.1. Basic concepts of systolic systems . . . 752
16.1.1. An introductory example: matrix product . . . 753
16.1.2. Problem parameters and array parameters . . . 754
16.1.3. Space coordinates . . . 754
16.1.4. Serialising generic operators . . . 755
16.1.5. Assignment-free notation . . . 756
16.1.6. Elementary operations . . . 757
16.1.7. Discrete timesteps . . . 757
16.1.8. External and internal communication . . . 758
16.1.9. Pipelining . . . 760
16.2. Space-time transformation and systolic arrays . . . 761
16.2.1. Further example: matrix product without stationary variables . . . 762
16.2.2. The space-time transformation as a global view . . . 762
16.2.3. Parametric space coordinates . . . 764
16.2.4. Symbolically deriving the running time . . . 767
16.2.5. How to unravel the communication topology . . . 767
16.2.6. Inferring the structure of the cells . . . 768
16.3. Input/output schemes . . . 770
16.3.1. From data structure indices to iteration vectors . . . 771
16.3.2. Snapshots of data structures . . . 772
16.3.3. Superposition of input/output schemes . . . 773
16.3.4. Data rates induced by space-time transformations . . . 774
16.3.5. Input/output expansion and extended input/output scheme . . . 774
16.3.6. Coping with stationary variables . . . 775
16.3.7. Interleaving of calculations . . . 776
16.4. Control . . . 778
16.4.1. Cells without control . . . 778
16.4.2. Global control . . . 779
16.4.3. Local control . . . 780

16.4.4. Distributed control . . . 783
16.4.5. The cell program as a local view . . . 787
16.5. Linear systolic arrays . . . 791
16.5.1. Matrix-vector product . . . 791
16.5.2. Sorting algorithms . . . 792
16.5.3. Lower triangular linear equation systems . . . 793

V. DATA BASES . . . 796
17. Memory Management (Ádám Balogh, Antal Iványi) . . . 797
17.1. Partitioning . . . 797
17.1.1. Fixed partitions . . . 798
17.1.2. Dynamic partitions . . . 805
17.2. Page replacement algorithms . . . 813
17.2.1. Static page replacement . . . 814
17.2.2. Dynamic paging . . . 821
17.3. Anomalies . . . 823
17.3.1. Page replacement . . . 824
17.3.2. Scheduling with lists . . . 825
17.3.3. Parallel processing with interleaved memory . . . 832
17.3.4. Avoiding the anomaly . . . 836
17.4. Optimal file packing . . . 836
17.4.1. Approximation algorithms . . . 836
17.4.2. Optimal algorithms . . . 840
17.4.3. Shortening of lists (SL) . . . 841
17.4.4. Upper and lower estimations (ULE) . . . 841
17.4.5. Pairwise comparison of the algorithms . . . 842
17.4.6. The error of approximate algorithms . . . 844
18. Relational Data Base Design (János Demetrovics, Attila Sali) . . . 849
18.1. Functional dependencies . . . 850
18.1.1. Armstrong-axioms . . . 850
18.1.2. Closures . . . 851
18.1.3. Minimal cover . . . 854
18.1.4. Keys . . . 856
18.2. Decomposition of relational schemata . . . 857
18.2.1. Lossless join . . . 859
18.2.2. Checking the lossless join property . . . 859
18.2.3. Dependency preserving decompositions . . . 863
18.2.4. Normal forms . . . 865
18.2.5. Multivalued dependencies . . . 871
18.3. Generalised dependencies . . . 877
18.3.1. Join dependencies . . . 877
18.3.2. Branching dependencies . . . 878
19. Query Rewriting in Relational Databases (János Demetrovics, Attila Sali) . . . 882
19.1. Queries . . . 882

19.1.1. Conjunctive queries . . . 884
19.1.2. Extensions . . . 889
19.1.3. Complexity of query containment . . . 897
19.2. Views . . . 901
19.2.1. View as a result of a query . . . 902
19.3. Query rewriting . . . 904
19.3.1. Motivation . . . 904
19.3.2. Complexity problems of query rewriting . . . 908
19.3.3. Practical algorithms . . . 912
20. Semi-structured Databases (Attila Kiss) . . . **931**
20.1. Semi-structured data and XML . . . 931
20.2. Schemas and simulations . . . 934
20.3. Queries and indexes . . . 938
20.4. Stable partitions and the PT-algorithm . . . 944
20.5. A(k)-indexes . . . 951
20.6. D(k)- and M(k)-indexes . . . 953
20.7. Branching queries . . . 960
20.8. Index refresh . . . 964

VI. APPLICATIONS . . . **972**
21. Bioinformatics (István Miklós) . . . **973**
21.1. Algorithms on sequences . . . 973
21.1.1. Distances of two sequences using linear gap penalty . . . 973
21.1.2. Dynamic programming with arbitrary gap function . . . 976
21.1.3. Gotoh algorithm for affine gap penalty . . . 977
21.1.4. Concave gap penalty . . . 977
21.1.5. Similarity of two sequences, the Smith-Waterman algorithm . . . 980
21.1.6. Multiple sequence alignment . . . 981
21.1.7. Memory-reduction with the Hirschberg algorithm . . . 983
21.1.8. Memory-reduction with corner-cutting . . . 984
21.2. Algorithms on trees . . . 986
21.2.1. The small parsimony problem . . . 986
21.2.2. The Felsenstein algorithm . . . 987
21.3. Algorithms on stochastic grammars . . . 989
21.3.1. Hidden Markov Models: Forward, Backward and the Viterbi algorithms . . . 989
21.3.2. Stochastic context-free grammars: the Inside, Outside and the CYK algorithms . . . 991
21.4. Comparing structures . . . 994
21.4.1. Aligning labelled, rooted trees . . . 994
21.4.2. Co-emission probability of two HMMs . . . 995
21.5. Distance based algorithms for constructing evolutionary trees . . . 997
21.5.1. Clustering algorithms . . . 998
21.5.2. Neighbour joining . . . 1000
21.6. Miscellaneous topics . . . 1005

21.6.1. Genome rearrangement . . . 1005
21.6.2. Shotgun sequencing . . . 1007
22. Computer Graphics (László Szirmay-Kalos) . . . 1013
22.1. Fundamentals of analytic geometry . . . 1013
22.1.1. Cartesian coordinate system . . . 1014
22.2. Description of point sets with equations . . . 1014
22.2.1. Solids . . . 1015
22.2.2. Surfaces . . . 1015
22.2.3. Curves . . . 1016
22.2.4. Normal vectors . . . 1017
22.2.5. Curve modelling . . . 1018
22.2.6. Surface modelling . . . 1023
22.2.7. Solid modelling with blobs . . . 1024
22.2.8. Constructive solid geometry . . . 1025
22.3. Geometry processing and tessellation algorithms . . . 1027
22.3.1. Polygon and polyhedron . . . 1027
22.3.2. Vectorization of parametric curves . . . 1028
22.3.3. Tessellation of simple polygons . . . 1028
22.3.4. Tessellation of parametric surfaces . . . 1030
22.3.5. Subdivision curves and meshes . . . 1032
22.3.6. Tessellation of implicit surfaces . . . 1034
22.4. Containment algorithms . . . 1036
22.4.1. Point containment test . . . 1036
22.4.2. Polyhedron-polyhedron collision detection . . . 1040
22.4.3. Clipping algorithms . . . 1041
22.5. Translation, distortion, geometric transformations . . . 1045
22.5.1. Projective geometry and homogeneous coordinates . . . 1046
22.5.2. Homogeneous linear transformations . . . 1050
22.6. Rendering with ray tracing . . . 1053
22.6.1. Ray surface intersection calculation . . . 1054
22.6.2. Speeding up the intersection calculation . . . 1056
22.7. Incremental rendering . . . 1071
22.7.1. Camera transformation . . . 1073
22.7.2. Normalizing transformation . . . 1074
22.7.3. Perspective transformation . . . 1075
22.7.4. Clipping in homogeneous coordinates . . . 1077
22.7.5. Viewport transformation . . . 1078
22.7.6. Rasterization algorithms . . . 1079
22.7.7. Incremental visibility algorithms . . . 1084
23. Human-Computer Interaction (Ingo Althöfer, Stefan Schwarz) . . . 1094
23.1. Multiple-choice systems . . . 1094
23.1.1. Examples of multiple-choice systems . . . 1095
23.2. Generating multiple candidate solutions . . . 1098
23.2.1. Generating candidate solutions with heuristics and repeated heuristics . . . 1098

23.2.2. Penalty method with exact algorithms 1101
23.2.3. The linear programming - penalty method 1109
23.2.4. Penalty method with heuristics 1113
23.3. More algorithms for interactive problem solving 1114
23.3.1. Anytime algorithms 1115
23.3.2. Interactive evolution and generative design 1116
23.3.3. Successive fixing 1116
23.3.4. Interactive multicriteria decision making 1116
23.3.5. Miscellaneous 1117
Bibliography **1119**
Subject Index **1133**
Name Index **1143**

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 Vörös. 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 Vörös (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)

IV. COMPUTER NETWORKS

13. Distributed Algorithms

We define a distributed system as a collection of individual computing devices that can communicate with each other. This definition is very broad, it includes anything, from a VLSI chip, to a tightly coupled multiprocessor, to a local area cluster of workstations, to the Internet. Here we focus on more loosely coupled systems. In a distributed system as we view it, each processor has its semi-independent agenda, but for various reasons, such as sharing of resources, availability, and fault-tolerance, processors need to coordinate their actions.

Distributed systems are highly desirable, but it is notoriously difficult to construct efficient distributed algorithms that perform well in realistic system settings. These difficulties are not just of a more practical nature, they are also fundamental in nature. In particular, many of the difficulties are introduced by the three factors of: asynchrony, limited local knowledge, and failures. Asynchrony means that global time may not be available, and that both absolute and relative times at which events take place at individual computing devices can often not be known precisely. Moreover, each computing device can only be aware of the information it receives, it has therefore an inherently local view of the global status of the system. Finally, computing devices and network components may fail independently, so that some remain functional while others do not.

We will begin by describing the models used to analyse distributed systems in the message-passing model of computation. We present and analyze selected distributed algorithms based on these models. We include a discussion of fault-tolerance in distributed systems and consider several algorithms for reaching agreement in the messages-passing models for settings prone to failures. Given that global time is often unavailable in distributed systems, we present approaches for providing logical time that allows one to reason about causality and consistent states in distributed systems. Moving on to more advanced topics, we present a spectrum of broadcast services often considered in distributed systems and present algorithms implementing these services. We also present advanced algorithms for rumor gathering algorithms. Finally, we also consider the mutual exclusion problem in the shared-memory model of distributed computation.

13.1. Message passing systems and algorithms

We present our first model of distributed computation, for message passing systems without failures. We consider both synchronous and asynchronous systems and present selected algorithms for message passing systems with arbitrary network topology, and both synchronous and asynchronous settings.

13.1.1. Modeling message passing systems

In a message passing system, processors communicate by sending messages over communication channels, where each channel provides a bidirectional connection between two specific processors. We call the pattern of connections described by the channels, the ***topology*** of the system. This topology is represented by an undirected graph, where each node represents a processor, and an edge is present between two nodes if and only if there is a channel between the two processors represented by the nodes. The collection of channels is also called the ***network***. An algorithm for such a message passing system with a specific topology consists of a local program for each processor in the system. This local program provides the ability to the processor to perform local computations, to send and receive messages from each of its neighbours in the given topology.

Each processor in the system is modeled as a possibly infinite state machine. A ***configuration*** is a vector $C = (q_0, \ldots, q_{n-1})$ where each q_i is the state of a processor p_i. Activities that can take place in the system are modeled as ***events*** (or *actions*) that describe indivisible system operations. Examples of events include local computation events and delivery events where a processor receives a message. The behaviour of the system over time is modeled as an ***execution***, a (finite or infinite) sequence of configurations (C_i) alternating with events (a_i): $C_0, a_1, C_1, a_2, C_2, \ldots$. Executions must satisfy a variety of conditions that are used to represent the correctness properties, depending on the system being modeled. These conditions can be classified as either safety or liveness conditions. A ***safety condition*** for a system is a condition that must hold in every finite prefix of any execution of the system. Informally it states that nothing *bad* has happened yet. A ***liveness condition*** is a condition that must hold a certain (possibly infinite) number of times. Informally it states that eventually something *good* must happen. An important liveness condition is ***fairness***, which requires that an (infinite) execution contains infinitely many actions by a processor, unless after some configuration no actions are enabled at that processor.

13.1.2. Asynchronous systems

We say that a system is ***asynchronous*** if there is no fixed upper bound on how long it takes for a message to be delivered or how much time elapses between consecutive steps of a processor. An obvious example of such an asynchronous system is the Internet. In an implementation of a distributed system there are often upper bounds on message delays and processor step times. But since these upper bounds are often very large and can change over time, it is often desirable to develop an algorithm

that is independent of any timing parameters, that is, an asynchronous algorithm.

In the asynchronous model we say that an execution is ***admissible*** if each processor has an infinite number of computation events, and every message sent is eventually delivered. The first of these requirements models the fact that processors do not fail. (It does not mean that a processor's local program contains an infinite loop. An algorithm can still terminate by having a transition function not change a processors state after a certain point.)

We assume that each processor's set of states includes a subset of ***terminated*** states. Once a processor enters such a state it remains in it. The algorithm has ***terminated*** if all processors are in terminated states and no messages are in transit.

The ***message complexity*** of an algorithm in the asynchronous model is the maximum over all admissible executions of the algorithm, of the total number of (point-to-point) messages sent.

A ***timed execution*** is an execution that has a nonnegative real number associated with each event, the *time* at which the event occurs. To measure the *time complexity* of an asynchronous algorithm we first assume that the maximum message delay in any execution is one unit of time. Hence the ***time complexity*** is the maximum time until termination among all timed admissible executions in which every message delay is at most one. Intuitively this can be viewed as taking any execution of the algorithm and normalising it in such a way that the longest message delay becomes one unit of time.

13.1.3. Synchronous systems

In the synchronous model processors execute in lock-step. The execution is partitioned into rounds so that every processor can send a message to each neighbour, the messages are delivered, and every processor computes based on the messages just received. This model is very convenient for designing algorithms. Algorithms designed in this model can in many cases be automatically simulated to work in other, more realistic timing models.

In the synchronous model we say that an execution is admissible if it is infinite. From the round structure it follows then that every processor takes an infinite number of computation steps and that every message sent is eventually delivered. Hence in a synchronous system with no failures, once a (deterministic) algorithm has been fixed, the only relevant aspect determining an execution that can change is the initial configuration. On the other hand in an asynchronous system, there can be many different executions of the same algorithm, even with the same initial configuration and no failures, since here the interleaving of processor steps, and the message delays, are not fixed.

The notion of *terminated states* and the *termination* of the algorithm is defined in the same way as in the asynchronous model.

The *message complexity* of an algorithm in the synchronous model is the maximum over all admissible executions of the algorithm, of the total number of messages sent.

To measure time in a synchronous system we simply count the number of rounds until termination. Hence the ***time complexity*** of an algorithm in the synchronous model is the maximum number of rounds in any admissible execution of the algorithm until the algorithm has terminated.

13.2. Basic algorithms

We begin with some simple examples of algorithms in the message passing model.

13.2.1. Broadcast

We start with a simple algorithm Spanning-Tree-Broadcast for the (single message) broadcast problem, assuming that a spanning tree of the network graph with n nodes (processors) is already given. Later, we will remove this assumption. A processor p_i wishes to send a message M to all other processors. The spanning tree rooted at p_i is maintained in a distributed fashion: Each processor has a distinguished channel that leads to its *parent* in the tree as well as a set of channels that lead to its *children* in the tree. The root p_i sends the message M on all channels leading to its children. When a processor receives the message on a channel from its parent, it sends M on all channels leading to its children.

Spanning-Tree-Broadcast

```
   Initially M is in transit from p_i to all its children in the spanning tree.
   Code for p_i:
1      upon receiving no message: // first computation event by p_i
2          terminate

   Code for p_j, 0 ≤ j ≤ n − 1, j ≠ i:
3      upon receiving M from parent:
4          send M to all children
5          terminate
```

The algorithm Spanning-Tree-Broadcast is correct whether the system is synchronous or asynchronous. Moreover, the message and time complexities are the same in both models.

Using simple inductive arguments we will first prove a lemma that shows that by the end of round t, the message M reaches all processors at distance t (or less) from p_r in the spanning tree.

Lemma 13.1 *In every admissible execution of the broadcast algorithm in the synchronous model, every processor at distance t from p_r in the spanning tree receives the message M in round t.*

Proof We proceed by induction on the distance t of a processor from p_r. First let $t = 1$. It follows from the algorithm that each child of p_r receives the message in round 1.

Assume that each processor at distance $t-1$ received the message M in round $t-1$. We need to show that each processor p_t at distance t receives the message in round t. Let p_s be the parent of p_t in the spanning tree. Since p_s is at distance $t-1$ from p_r, by the induction hypothesis, p_s received M in round $t-1$. By the algorithm, p_t will hence receive M in round t. ■

By Lemma 13.1 the time complexity of the broadcast algorithm is d, where d is the depth of the spanning tree. Now since d is at most $n-1$ (when the spanning tree is a chain) we have:

Theorem 13.2 *There is a synchronous broadcast algorithm for n processors with message complexity $n-1$ and time complexity d, when a rooted spanning tree with depth d is known in advance.*

We now move to an asynchronous system and apply a similar analysis.

Lemma 13.3 *In every admissible execution of the broadcast algorithm in the asynchronous model, every processor at distance t from p_r in the spanning tree receives the message M by time t.*

Proof We proceed by induction on the distance t of a processor from p_r. First let $t=1$. It follows from the algorithm that M is initially in transit to each processor p_i at distance 1 from p_r. By the definition of time complexity for the asynchronous model, p_i receives M by time 1.

Assume that each processor at distance $t-1$ received the message M at time $t-1$. We need to show that each processor p_t at distance t receives the message by time t. Let p_s be the parent of p_t in the spanning tree. Since p_s is at distance $t-1$ from p_r, by the induction hypothesis, p_s sends M to p_t when it receives M at time $t-1$. By the algorithm, p_t will hence receive M by time t. ■

We immediately obtain:

Theorem 13.4 *There is an asynchronous broadcast algorithm for n processors with message complexity $n-1$ and time complexity d, when a rooted spanning tree with depth d is known in advance.*

13.2.2. Construction of a spanning tree

The asynchronous algorithm called Flood, discussed next, constructs a spanning tree rooted at a designated processor p_r. The algorithm is similar to the Depth First Search (DFS) algorithm. However, unlike DFS where there is just one processor with "global knowledge" about the graph, in the Flood algorithm, each processor has "local knowledge" about the graph, processors coordinate their work by exchanging messages, and processors and messages may get delayed arbitrarily. This makes the design and analysis of Flood algorithm challenging, because we need to show that the algorithm indeed constructs a spanning tree despite conspiratorial selection of these delays.

Algorithm description

Each processor has four local variables. The links adjacent to a processor are identified with distinct numbers starting from 1 and stored in a local variable called

neighbours. We will say that the ***spanning tree has been constructed***, when the variable *parent* stores the identifier of the link leading to the parent of the processor in the spanning tree, except that this variable is NONE for the designated processor p_r; *children* is a set of identifiers of the links leading to the children processors in the tree; and *other* is a set of identifiers of all other links. So the knowledge about the spanning tree may be "distributed" across processors.

The code of each processor is composed of segments. There is a segment (lines 1–4) that describes how local variables of a processor are initialised. Recall that the local variables are initialised that way before time 0. The next three segments (lines 5–11, 12–15 and 16–19) describe the instructions that any processor executes in response to having received a message: <*adopt*>, <*approved*> or <*rejected*>. The last segment (lines 20–22) is only included in the code of processor p_r. This segment is executed only when the local variable *parent* of processor p_r is NIL. At some point of time, it may happen that more than one segment can be executed by a processor (e.g., because the processor received <*adopt*> messages from two processors). Then the processor executes the segments serially, one by one (segments of any given processor are never executed concurrently). However, instructions of different processor may be arbitrarily interleaved during an execution. Every message that can be processed is eventually processed and every segment that can be executed is eventually executed (fairness).

FLOOD

```
    Code for any processor p_k, 1 ≤ k ≤ n
 1  initialisation
 2      parent ← NIL
 3      children ← ∅
 4      other ← ∅

 5  process message <adopt> that has arrived on link j
 6      if parent = NIL
 7          then parent ← j
 8               send <approved> to link j
 9               send <adopt> to all links in neighbours \ {j}
10          else send <rejected> to link j

11  process message <approved> that has arrived on link j
12      children ← children ∪ {j}
13      if children ∪ other = neighbours \ {parent}
14          then terminate

15  process message <rejected> that has arrived on link j
16      other ← other ∪ {j}
17      if children ∪ other = neighbours \ {parent}
18          then terminate
```

```
    Extra code for the designated processor p_r
19  if parent = NIL
20     then parent ← NONE
21          send <adopt> to all links in neighbours
```

Let us outline how the algorithm works. The designated processor sends an $<$*adopt*$>$ message to all its neighbours, and assigns NONE to the *parent* variable (NIL and NONE are two distinguished values, different from any natural number), so that it never again sends the message to any neighbour.

When a processor processes message $<$*adopt*$>$ for the first time, the processor assigns to its own *parent* variable the identifier of the link on which the message has arrived, responds with an $<$*approved*$>$ message to that link, and forwards an $<$*adopt*$>$ message to every other link. However, when a processor processes message $<$*adopt*$>$ again, then the processor responds with a $<$*rejected*$>$ message, because the *parent* variable is no longer NIL.

When a processor processes message $<$*approved*$>$, it adds the identifier of the link on which the message has arrived to the set *children*. It may turn out that the sets *children* and *other* combined form identifiers of all links adjacent to the processor except for the identifier stored in the *parent* variable. In this case the processor enters a terminating state.

When a processor processes message $<$*rejected*$>$, the identifier of the link is added to the set *other*. Again, when the union of *children* and *other* is large enough, the processor enters a terminating state.

Correctness proof We now argue that FLOOD constructs a spanning tree. The key moments in the execution of the algorithm are when any processor assigns a value to its *parent* variable. These assignments determine the "shape" of the spanning tree. The facts that any processor eventually executes an instruction, any message is eventually delivered, and any message is eventually processed, ensure that the knowledge about these assignments spreads to neighbours. Thus the algorithm is expanding a subtree of the graph, albeit the expansion may be slow. Eventually, a spanning tree is formed. Once a spanning tree has been constructed, eventually every processor will terminate, even though some processors may have terminated even before the spanning tree has been constructed.

Lemma 13.5 *For any $1 \leq k \leq n$, there is time t_k which is the first moment when there are exactly k processors whose parent variables are not* NIL*, and these processors and their parent variables form a tree rooted at p_r.*

Proof We prove the statement of the lemma by induction on k. For the base case, assume that $k = 1$. Observe that processor p_r eventually assigns NONE to its *parent* variable. Let t_1 be the moment when this assignment happens. At that time, the *parent* variable of any processor other than p_r is still NIL, because no $<$*adopt*$>$ messages have been sent so far. Processor p_r and its *parent* variable form a tree with a single node and not arcs. Hence they form a rooted tree. Thus the inductive hypothesis holds for $k = 1$.

For the inductive step, suppose that $1 \leq k < n$ and that the inductive hypothesis holds for k. Consider the time t_k which is the first moment when there are exactly k processors whose *parent* variables are not *nil*. Because $k < n$, there is a non-tree processor. But the graph G is connected, so there is a non-tree processor adjacent to the tree. (For any subset T of processors, a processor p_i is ***adjacent*** to T if and only if there an edge in the graph G from p_i to a processor in T.) Recall that by definition, *parent* variable of such processor is NIL. By the inductive hypothesis, the k processors must have executed line 7 of their code, and so each either has already sent or will eventually send $<$*adopt*$>$ message to all its neighbours on links other than the *parent* link. So the non-tree processors adjacent to the tree have already received or will eventually receive $<$*adopt*$>$ messages. Eventually, each of these adjacent processors will, therefore, assign a value other than NIL to its *parent* variable. Let $t_{k+1} > t_k$ be the first moment when any processor performs such assignment, and let us denote this processor by p_i. This cannot be a tree processor, because such processor never again assigns any value to its *parent* variable. Could p_i be a non-tree processor that is not adjacent to the tree? It could not, because such processor does not have a direct link to a tree processor, so it cannot receive $<$*adopt*$>$ directly from the tree, and so this would mean that at some time t' between t_k and t_{k+1} some other non-tree processor p_j must have sent $<$*adopt*$>$ message to p_i, and so p_j would have to assign a value other than NIL to its *parent* variable some time after t_k but before t_{k+1}, contradicting the fact the t_{k+1} is the first such moment. Consequently, p_i is a non-tree processor adjacent to the tree, such that, at time t_{k+1}, p_i assigns to its *parent* variable the index of a link leading to a tree processor. Therefore, time t_{k+1} is the first moment when there are exactly $k+1$ processors whose *parent* variables are not NIL, and, at that time, these processors and their *parent* variables form a tree rooted at p_r. This completes the inductive step, and the proof of the lemma. ■

Theorem 13.6 *Eventually each processor terminates, and when every processor has terminated, the subgraph induced by the parent variables forms a spanning tree rooted at p_r.*

Proof By Lemma 13.5, we know that there is a moment t_n which is the first moment when all processors and their *parent* variables form a spanning tree.

Is it possible that every processor has terminated before time t_n? By inspecting the code, we see that a processor terminates only after it has received $<$*rejected*$>$ or $<$*approved*$>$ messages from all its neighbours other than the one to which *parent* link leads. A processor receives such messages only in response to $<$*adopt*$>$ messages that the processor sends. At time t_n, there is a processor that still has not even sent $<$*adopt*$>$ messages. Hence, not every processor has terminated by time t_n.

Will every processor eventually terminate? We notice that by time t_n, each processor either has already sent or will eventually send $<$*adopt*$>$ message to all its neighbours other than the one to which *parent* link leads. Whenever a processor receives $<$*adopt*$>$ message, the processor responds with $<$*rejected*$>$ or $<$*approved*$>$, even if the processor has already terminated. Hence, eventually, each processor will receive either $<$*rejected*$>$ or $<$*approved*$>$ message on each link to which the processor has sent $<$*adopt*$>$ message. Thus, eventually, each processor terminates. ■

We note that the fact that a processor has terminated does not mean that a

spanning tree has already been constructed. In fact, it may happen that processors in a different part of the network have not even received any message, let alone terminated.

Theorem 13.7 *Message complexity of* FLOOD *is* $O(e)$*, where* e *is the number of edges in the graph* G*.*

The proof of this theorem is left as Problem *13-1*.

Exercises

13.2-1 It may happen that a processor has terminated even though a processor has not even received any message. Show a simple network and how to delay message delivery and processor computation to demonstrate that this can indeed happen.

13.2-2 It may happen that a processor has terminated but may still respond to a message. Show a simple network and how to delay message delivery and processor computation to demonstrate that this can indeed happen.

13.3. Ring algorithms

One often needs to coordinate the activities of processors in a distributed system. This can frequently be simplified when there is a single processor that acts as a coordinator. Initially, the system may not have any coordinator, or an existing coordinator may fail and so another may need to be elected. This creates the problem where processors must elect exactly one among them, a *leader*. In this section we study the problem for special types of networks—rings. We will develop an asynchronous algorithm for the problem. As we shall demonstrate, the algorithm has asymptotically optimal message complexity. In the current section, we will see a distributed analogue of the well-known divide-and-conquer technique often used in sequential algorithms to keep their time complexity low. The technique used in distributed systems helps reduce the message complexity.

13.3.1. The leader election problem

The leader election problem is to elect exactly leader among a set of processors. Formally each processor has a local variable *leader* initially equal to NIL. An algorithm is said to ***solve the leader election problem*** if it satisfies the following conditions:

1. in any execution, exactly one processor eventually assigns TRUE to its *leader* variable, all other processors eventually assign FALSE to their *leader* variables, and
2. in any execution, once a processor has assigned a value to its *leader* variable, the variable remains unchanged.

Ring model

We study the leader election problem on a special type of network — the ring. Formally, the graph G that models a distributed system consists of n nodes that form a simple cycle; no other edges exist in the graph. The two links adjacent to a

processor are labeled ***CW*** (**C**lock-**W**ise) and CCW (counter clock-wise). Processors agree on the orientation of the ring i.e., if a message is passed on in CW direction n times, then it visits all n processors and comes back to the one that initially sent the message; same for CCW direction. Each processor has a unique ***identifier*** that is a natural number, i.e., the identifier of each processor is different from the identifier of any other processor; the identifiers do not have to be consecutive numbers $1, \ldots, n$. Initially, no processor knows the identifier of any other processor. Also processors do not know the size n of the ring.

13.3.2. The leader election algorithm

BULLY elects a leader among asynchronous processors $p_1, \ldots, p_n$. Identifiers of processors are used by the algorithm in a crucial way. Briefly speaking, each processor tries to become the leader, the processor that has the largest identifier among all processors blocks the attempts of other processors, declares itself to be the leader, and forces others to declare themselves not to be leaders.

Let us begin with a simpler version of the algorithm to exemplify some of the ideas of the algorithm. Suppose that each processor sends a message around the ring containing the identifier of the processor. Any processor passes on such message *only* if the identifier that the message carries is strictly larger than the identifier of the processor. Thus the message sent by the processor that has the largest identifier among the processors of the ring, will always be passed on, and so it will eventually travel around the ring and come back to the processor that initially sent it. The processor can detect that such message has come back, because no other processor sends a message with this identifier (identifiers are distinct). We observe that, no other message will make it all around the ring, because the processor with the largest identifier will not pass it on. We could say that the processor with the largest identifier "swallows" these messages that carry smaller identifiers. Then the processor becomes the leader and sends a special message around the ring forcing all others to decide not to be leaders. The algorithm has $\Theta(n^2)$ message complexity, because each processor induces at most n messages, and the leader induces n extra messages; and one can assign identifiers to processors and delay processors and messages in such a way that the messages sent by a constant fraction of n processors are passed on around the ring for a constant fraction of n hops. The algorithm can be improved so as to reduce message complexity to $O(n \lg n)$, and such improved algorithm will be presented in the remainder of the section.

The key idea of the BULLY algorithm is to make sure that not too many messages travel far, which will ensure $O(n \lg n)$ message complexity. Specifically, the activity of any processor is divided into phases. At the beginning of a phase, a processor sends "probe" messages in both directions: CW and CCW. These messages carry the identifier of the sender and a certain "time-to-live" value that limits the number of hops that each message can make. The probe message may be passed on by a processor provided that the identifier carried by the message is larger than the identifier of the processor. When the message reaches the limit, and has not been swallowed, then it is "bounced back". Hence when the initial sender receives two bounced back messages, each from each direction, then the processor is certain

that there is no processor with larger identifier up until the limit in CW nor CCW directions, because otherwise such processor would swallow a probe message. Only then does the processor enter the next phase through sending probe messages again, this time with the time-to-live value increased by a factor, in an attempt to find if there is no processor with a larger identifier in twice as large neighbourhood. As a result, a probe message that the processor sends will make many hops only when there is no processor with larger identifier in a large neighbourhood of the processor. Therefore, fewer and fewer processors send messages that can travel longer and longer distances. Consequently, as we will soon argue in detail, message complexity of the algorithm is $O(n \lg n)$.

We detail the BULLY algorithm. Each processor has five local variables. The variable *id* stores the unique identifier of the processor. The variable *leader* stores TRUE when the processor decides to be the leader, and FALSE when it decides not to be the leader. The remaining three variables are used for bookkeeping: *asleep* determines if the processor has ever sent a <*probe,id*,0,0> message that carries the identifier *id* of the processor. Any processor may send <*probe,id,phase*,$2^{phase-1}$> message in both directions (CW and CCW) for different values of *phase*. Each time a message is sent, a <*reply*, *id*, *phase*> message may be sent back to the processor. The variables *CWreplied* and *CCWreplied* are used to remember whether the replies have already been processed the processor.

The code of each processor is composed of five segments. The first segment (lines 1–5) initialises the local variables of the processor. The second segment (lines 6–8) can only be executed when the local variable *asleep* is TRUE. The remaining three segments (lines 9–17, 1–26, and 27–31) describe the actions that the processor takes when it processes each of the three types of messages: <*probe*, *ids*, *phase*, *ttl*>, <*reply*, *ids*, *phase*> and <*terminate*> respectively. The messages carry parameters *ids*, *phase* and *ttl* that are natural numbers.

We now describe how the algorithm works. Recall that we assume that the local variables of each processor have been initialised before time 0 of the global clock. Each processor eventually sends a <*probe,id*,0,0> message carrying the identifier *id* of the processor. At that time we say that the processor *enters* phase number zero. In general, when a processor sends a message <*probe,id,phase*,$2^{phase}-1$>, we say that the processor *enters* phase number *phase*. Message <*probe,id*,0,0> is never sent again because FALSE is assigned to *asleep* in line 7. It may happen that by the time this message is sent, some other messages have already been processed by the processor.

When a processor processes message <*probe*, *ids*, *phase*, *ttl*> that has arrived on link CW (the link leading in the clock-wise direction), then the actions depend on the relationship between the parameter *ids* and the identifier *id* of the processor. If *ids* is smaller than *id*, then the processor does nothing else (the processor swallows the message). If *ids* is equal to *id* and processor has not yet decided, then, as we shall see, the probe message that the processor sent has circulated around the entire ring. Then the processor sends a <*terminate*> message, decides to be the leader, and terminates (the processor may still process messages after termination). If *ids* is larger than *id*, then actions of the processor depend on the value of the parameter *ttl* (time-to-live). When the value is strictly larger than zero, then the processor

passes on the probe message with *ttl* decreased by one. If, however, the value of *ttl* is already zero, then the processor sends back (in the CW direction) a reply message. Symmetric actions are executed when the $<probe, ids, phase, ttl>$ message has arrived on link CCW, in the sense that the directions of sending messages are respectively reversed – see the code for details.

BULLY

```
    Code for any processor p_k, 1 ≤ k ≤ n
 1  initialisation
 2      asleep ← TRUE
 3      CWreplied ← FALSE
 4      CCWreplied ← FALSE
 5      leader ← NIL

 6  if asleep
 7     then asleep ←FALSE
 8          send <probe,id,0,0> to links CW and CCW

 9  process message <probe, ids, phase, ttl> that has arrived on link CW (resp. CCW)
10     if id = ids and leader = NIL
11        then send <terminate> to link CCW
12             leader ← TRUE
13             terminate
14     if ids > id and ttl > 0
15        then send < probe,ids,phase,ttl − 1 > to link CCW (resp. CW)
16     if ids > id and ttl = 0
17        then send <reply,ids,phase> to link CW (resp. CCW)

18  process message <reply,ids,phase> that has arrived on link CW (resp. CCW)
19     if id ≠ ids
20        then send <reply,ids,phase> to link CCW (resp. CW)
21        else  CWreplied ← TRUE (resp. CCWreplied)
22        if CWreplied and CCWreplied
23        then CWreplied ← FALSE
24             CCWreplied ← FALSE
25             send <probe,id,phase+1,2^(phase+1) − 1> to links CW and CCW
```

```
27  process message <terminate> that has arrived on link CW
28      if leader NIL
29          then send <terminate> to link CCW
30               leader ← FALSE
31               terminate
```

When a processor processes message <*reply*, *ids*, *phase*> that has arrived on link CW, then the processor first checks if *ids* is different from the identifier *id* of the processor. If so, the processor merely passes on the message. However, if $ids = id$, then the processor records the fact that a reply has been received from direction CW, by assigning TRUE to *CWreplied*. Next the processor checks if both *CWreplied* and *CCWreplied* variables are *true*. If so, the processor has received replies from both directions. Then the processor assigns *false* to both variables. Next the processor sends a probe message. This message carries the identifier *id* of the processor, the next phase number $phase + 1$, and an increased time-to-live parameter $2^{phase+1} - 1$. Symmetric actions are executed when <*reply,ids,phase*> has arrived on link CCW.

The last type of message that a processor can process is <*terminate*>. The processor checks if it has already decided to be or not to be the leader. When no decision has been made so far, the processor passes on the <*terminate*> message and decides not to be the leader. This message eventually reaches a processor that has already decided, and then the message is no longer passed on.

13.3.3. Analysis of the leader election algorithm

We begin the analysis by showing that the algorithm BULLY solves the leader election problem.

Correctness proof

Theorem 13.8 BULLY *solves the leader election problem on any ring with asynchronous processors.*

Proof We need to show that the two conditions listed at the beginning of the section are satisfied. The key idea that simplifies the argument is to focus on one processor. Consider the processor p_i with maximum *id* among all processors in the ring. This processor eventually executes lines 6–8. Then the processor sends <*probe,id*,0,0> messages in CW and CCW directions. Note that whenever the processor sends <*probe,id,phase*,$2^{phase} - 1$> messages, each such message is always passed on by other processors, until the *ttl* parameter of the message drops down to zero, or the message travels around the entire ring and arrives at p_i. If the message never arrives at p_i, then a processor eventually receives the probe message with *ttl* equal to zero, and the processor sends a response back to p_i. Then, eventually p_i receives messages <*reply,id,phase*> from each directions, and enters phase number $phase + 1$ by sending probe messages <*probe,id,phase*+1,$2^{phase+1} - 1$> in both directions. These messages carry a larger time-to-live value compared to the value from the previous phase number *phase*. Since the ring is finite, eventually *ttl* becomes so large that

processor p_i receives a probe message that carries the identifier of p_i. Note that p_i will eventually receive two such messages. The first time when p_i processes such message, the processor sends a <*terminate*> message and terminates as the leader. The second time when p_i processes such message, lines 11–13 are not executed, because variable *leader* is no longer NIL. Note that no other processor p_j can execute lines 11–13, because a probe message originated at p_j cannot travel around the entire ring, since p_i is on the way, and p_i would swallow the message; and since identifiers are distinct, no other processor sends a probe message that carries the identifier of processor p_j. Thus no processor other than p_i can assign TRUE to its *leader* variable. Any processor other than p_i will receive the <*terminate*> message, assign FALSE to its *leader* variable, and pass on the message. Finally, the <*terminate*> message will arrive at p_i, and p_i will not pass it anymore. The argument presented thus far ensures that eventually exactly one processor assigns TRUE to its *leader* variable, all other processors assign FALSE to their *leader* variables, and once a processor has assigned a value to its *leader* variable, the variable remains unchanged. ■

Our next task is to give an upper bound on the number of messages sent by the algorithm. The subsequent lemma shows that the number of processors that can enter a phase decays exponentially as the phase number increases.

Lemma 13.9 *Given a ring of size n, the number k of processors that enter phase number $i \geq 0$ is at most $n/2^{i-1}$.*

Proof There are exactly n processors that enter phase number $i = 0$, because each processor eventually sends <*probe,id*,0,0> message. The bound stated in the lemma says that the number of processors that enter phase 0 is at most $2n$, so the bound evidently holds for $i = 0$. Let us consider any of the remaining cases i.e., let us assume that $i \geq 1$. Suppose that a processor p_j enters phase number i, and so by definition it sends message <*probe,id,i*,$2^i - 1$>. In order for a processor to send such message, each of the two probe messages <*probe,id,i*-1,$2^{i-1} - 1$> that the processor sent in the previous phase in both directions must have made 2^{i-1} hops always arriving at a processor with strictly lower identifier than the identifier of p_j (because otherwise, if a probe message arrives at a processor with strictly larger or the same identifier, than the message is swallowed, and so a reply message is not generated, and consequently p_j cannot enter phase number i). As a result, if a processor enters phase number i, then there is no other processor 2^{i-1} hops away in both directions that can ever enter the phase. Suppose that there are $k \geq 1$ processors that enter phase i. We can associate with each such processor p_j, the 2^{i-1} consecutive processors that follow p_j in the CW direction. This association assigns 2^{i-1} distinct processors to each of the k processors. So there must be at least $k + k \cdot 2^{i-1}$ distinct processor in the ring. Hence $k(1 + 2^{i-1}) \leq n$, and so we can weaken this bound by dropping 1, and conclude that $k \cdot 2^{i-1} \leq n$, as desired. ■

Theorem 13.10 *The algorithm* BULLY *has $O(n \lg n)$ message complexity, where n is the size of the ring.*

Proof Note that any processor in phase i, sends messages that are intended to travel 2^i away and back in each direction (CW and CCW). This contributes at most $4 \cdot 2^i$

messages per processor that enters phase number i. The contribution may be smaller than $4 \cdot 2^i$ if a probe message gets swallowed on the way away from the processor. Lemma 13.9 provides an upper bound on the number of processors that enter phase number k. What is the highest phase that a processor can ever enter? The number k of processors that can be in phase i is at most $n/2^{i-1}$. So when $n/2^{i-1} < 1$, then there can be no processor that ever enters phase i. Thus no processor can enter any phase beyond phase number $h = 1 + \lceil \log_2 n \rceil$, because $n < 2^{(h+1)-1}$. Finally, a single processor sends one termination message that travels around the ring once. So for the total number of messages sent by the algorithm we get the

$$n + \sum_{i=0}^{1+\lceil \log_2 n \rceil} \left(n/2^{i-1} \cdot 4 \cdot 2^i \right) = n + \sum_{i=0}^{1+\lceil \log_2 n \rceil} 8n = O(n \lg n)$$

upper bound. ■

Burns furthermore showed that the asynchronous leader election algorithm is asymptotically optimal: Any uniform algorithm solving the leader election problem in an asynchronous ring must send the number of messages at least proportional to $n \lg n$.

Theorem 13.11 *Any uniform algorithm for electing a leader in an asynchronous ring sends $\Omega(n \lg n)$ messages.*

The proof, for any algorithm, is based on constructing certain executions of the algorithm on rings of size $n/2$. Then two rings of size $n/2$ are pasted together in such a way that the constructed executions on the smaller rings are combined, and $\Theta(n)$ additional messages are received. This construction strategy yields the desired logarithmic multiplicative overhead.

Exercises

13.3-1 Show that the simplified BULLY algorithm has $\Omega(n^2)$ message complexity, by appropriately assigning identifiers to processors on a ring of size n, and by determining how to delay processors and messages.

13.3-2 Show that the algorithm BULLY has $\Omega(n \lg n)$ message complexity.

13.4. Fault-tolerant consensus

The algorithms presented so far are based on the assumption that the system on which they run is reliable. Here we present selected algorithms for unreliable distributed systems, where the active (or correct) processors need to coordinate their activities based on common decisions.

It is inherently difficult for processors to reach agreement in a distributed setting prone to failures. Consider the deceptively simple problem of two failure-free processors attempting to agree on a common bit using a communication medium where messages may be lost. This problem is known as the ***two generals problem***. Here two generals must coordinate an attack using couriers that may be destroyed by the enemy. It turns out that it is not possible to solve this problem using a finite

number of messages. We prove this fact by contradiction. Assume that there is a protocol used by processors A and B involving a finite number of messages. Let us consider such a protocol that uses the smallest number of messages, say k messages. Assume without loss of generality that the last k^{th} message is sent from A to B. Since this final message is not acknowledged by B, A must determine the decision value whether or not B receives this message. Since the message may be lost, B must determine the decision value without receiving this final message. But now both A and B decide on a common value without needing the k^{th} message. In other words, there is a protocol that uses only $k-1$ messages for the problem. But this contradicts the assumption that k is the smallest number of messages needed to solve the problem.

In the rest of this section we consider agreement problems where the communication medium is reliable, but where the processors are subject to two types of failures: ***crash failures***, where a processor stops and does not perform any further actions, and ***Byzantine failures***, where a processor may exhibit arbitrary, or even malicious, behaviour as the result of the failure.

The algorithms presented deal with the so called ***consensus problem***, first introduced by Lamport, Pease, and Shostak. The consensus problem is a fundamental coordination problem that requires processors to agree on a common output, based on their possibly conflicting inputs.

13.4.1. The consensus problem

We consider a system in which each processor p_i has a special state component x_i, called the ***input*** and y_i, called the ***output*** (also called the ***decision***). The variable x_i initially holds a value from some well ordered set of possible inputs and y_i is undefined. Once an assignment to y_i has been made, it is irreversible. Any solution to the consensus problem must guarantee:

- **Termination:** In every admissible execution, y_i is eventually assigned a value, for every nonfaulty processor p_i.

- **Agreement:** In every execution, if y_i and y_j are assigned, then $y_i = y_j$, for all nonfaulty processors p_i and p_j. That is nonfaulty processors do not decide on conflicting values.

- **Validity:** In every execution, if for some value v, $x_i = v$ for all processors p_i, and if y_i is assigned for some nonfaulty processor p_i, then $y_i = v$. That is, if all processors have the same input value, then any value decided upon must be that common input.

Note that in the case of crash failures this validity condition is equivalent to requiring that every nonfaulty decision value is the input of some processor. Once a processor crashes it is of no interest to the algorithm, and no requirements are put on its decision.

We begin by presenting a simple algorithm for consensus in a synchronous message passing system with crash failures.

13.4.2. Consensus with crash failures

Since the system is synchronous, an execution of the system consists of a series of rounds. Each round consists of the delivery of all messages, followed by one computation event for every processor. The set of faulty processors can be different in different executions, that is, it is not known in advance. Let F be a subset of at most f processors, the faulty processors. Each round contains exactly one computation event for the processors not in F and at most one computation event for every processor in F. Moreover, if a processor in F does not have a computation event in some round, it does not have such an event in any further round. In the last round in which a faulty processor has a computation event, an arbitrary subset of its outgoing messages are delivered.

CONSENSUS-WITH-CRASH-FAILURES

```
   Code for processor p_i, 0 ≤ i ≤ n − 1.
   Initially V = {x}
   round k, 1 ≤ k ≤ f + 1
1  send {v ∈ V : p_i has not already sent v} to all processors
2  receive S_j from p_j, 0 ≤ j ≤ n − 1, j ≠ i
3  V ← V ∪ ⋃_{j=0}^{n−1} S_j
4  if k = f + 1
5     then y ← min(V)
```

In the previous algorithm, which is based on an algorithm by Dolev and Strong, each processor maintains a set of the values it knows to exist in the system. Initially, the set contains only its own input. In later rounds the processor updates its set by joining it with the sets received from other processors. It then broadcasts any new additions to the set of all processors. This continues for $f+1$ rounds, where f is the maximum number of processors that can fail. At this point, the processor decides on the smallest value in its set of values.

To prove the correctness of this algorithm we first notice that the algorithm requires exactly $f+1$ rounds. This implies termination. Moreover the validity condition is clearly satisfied since the decision value is the input of some processor. It remains to show that the agreement condition holds. We prove the following lemma:

Lemma 13.12 *In every execution at the end of round $f+1$, $V_i = V_j$, for every two nonfaulty processors p_i and p_j.*

Proof We prove the claim by showing that if $x \in V_i$ at the end of round $f+1$ then $x \in V_j$ at the end of round $f+1$.

Let r be the first round in which x is added to V_i for any nonfaulty processor p_i. If x is initially in V_i let $r = 0$. If $r \leq f$ then, in round $r+1 \leq f+1$ p_i sends x to each p_j, causing p_j to add x to V_j, if not already present.

Otherwise, suppose $r = f+1$ and let p_j be a nonfaulty processor that receives x for the first time in round $f+1$. Then there must be a chain of $f+1$ processors

$p_{i_1}, \ldots p_{i_{f+1}}$ that transfers the value x to p_j. Hence p_{i_1} sends x to p_{i_2} in round one etc. until $p_{i_{f+1}}$ sends x to p_j in round $f+1$. But then $p_{i_1}, \ldots, p_{i_{f+1}}$ is a chain of $f+1$ processors. Hence at least one of them, say p_{i_k} must be nonfaulty. Hence p_{i_k} adds x to its set in round $k-1 < r$, contradicting the minimality of r. ■

This lemma together with the before mentioned observations hence implies the following theorem.

Theorem 13.13 *The previous consensus algorithm solves the consensus problem in the presence of f crash failures in a message passing system in $f+1$ rounds.*

The following theorem was first proved by Fischer and Lynch for Byzantine failures. Dolev and Strong later extended it to crash failures. The theorem shows that the previous algorithm, assuming the given model, is optimal.

Theorem 13.14 *There is no algorithm which solves the consensus problem in less than $f+1$ rounds in the presence of f crash failures, if $n \geq f+2$.*

What if failures are not benign? That is can the consensus problem be solved in the presence of *Byzantine* failures? And if so, how?

13.4.3. Consensus with Byzantine failures

In a computation step of a faulty processor in the Byzantine model, the new state of the processor and the message sent are completely unconstrained. As in the reliable case, every processor takes a computation step in every round and every message sent is delivered in that round. Hence a faulty processor can behave arbitrarily and even maliciously. For example, it could send different messages to different processors. It can even appear that the faulty processors coordinate with each other. A faulty processor can also mimic the behaviour of a crashed processor by failing to send any messages from some point on.

In this case, the definition of the consensus problem is the same as in the message passing model with crash failures. The validity condition in this model, however, is not equivalent with requiring that every nonfaulty decision value is the input of some processor. Like in the crash case, no conditions are put on the output of faulty processors.

13.4.4. Lower bound on the ratio of faulty processors

Pease, Shostak and Lamport first proved the following theorem.

Theorem 13.15 *In a system with n processors and f Byzantine processors, there is no algorithm which solves the consensus problem if $n \leq 3f$.*

13.4.5. A polynomial algorithm

The following algorithm uses messages of constant size, takes $2(f+1)$ rounds, and assumes that $n > 4f$. It was presented by Berman and Garay.

This consensus algorithm for Byzantine failures contains $f+1$ phases, each

taking two rounds. Each processor has a preferred decision for each phase, initially its input value. At the first round of each phase, processors send their preferences to each other. Let v_i^k be the majority value in the set of values received by processor p_i at the end of the first round of phase k. If no majority exists, a default value $v_\perp$ is used. In the second round of the phase processor p_k, called the *king* of the phase, sends its majority value v_k^k to all processors. If p_i receives more than $n/2 + f$ copies of v_i^k (in the first round of the phase) then it sets its preference for the next phase to be v_i^k; otherwise it sets its preference to the phase kings preference, v_k^k received in the second round of the phase. After $f + 1$ phases, the processor decides on its preference. Each processor maintains a local array *pref* with n entries.

We prove correctness using the following lemmas. Termination is immediate. We next note the persistence of agreement:

Lemma 13.16 *If all nonfaulty processors prefer v at the beginning of phase k, then they all prefer v at the end of phase k, for all k, $1 \leq k \leq f + 1$.*

Proof Since all nonfaulty processors prefer v at the beginning of phase k, they all receive at least $n - f$ copies of v (including their own) in the first round of phase k. Since $n > 4f$, $n - f > n/2 + f$, implying that all nonfaulty processors will prefer v at the end of phase k. ■

CONSENSUS-WITH-BYZANTINE-FAILURES

```
    Code for processor p_i, 0 ≤ i ≤ n − 1.
    Initially pref[j] = v_⊥, for any j ≠ i
    round 2k − 1, 1 ≤ k ≤ f + 1
 1  send ⟨pref[i]⟩ to all processors
 2  receive ⟨v_j⟩ from p_j and assign to pref[j], for all 0 ≤ j ≤ n − 1, j ≠ i
 3  let maj be the majority value of pref[0],...,pref[n − 1](v_⊥ if none)
 4  let mult be the multiplicity of maj

    round 2k, 1 ≤ k ≤ f + 1
 5  if i = k
 6     then send ⟨maj⟩ to all processors
 7  receive ⟨king-maj⟩ from p_k (v_⊥ if none)
 8  if mult > n/2 + f
 9     then pref[i] ← maj
10     else pref[i] ← king − maj
11  if k = f + 1
12     then y ←pref[i]
```

This implies the validity condition: If they all start with the same input v they will continue to prefer v and finally decide on v in phase $f+1$. Agreement is achieved by the king breaking ties. Since each phase has a different king and there are $f + 1$

phases, at least one round has a nonfaulty king.

Lemma 13.17 *Let g be a phase whose king p_g is nonfaulty. Then all nonfaulty processors finish phase g with the same preference.*

Proof Suppose all nonfaulty processors use the majority value received from the king for their preference. Since the king is nonfaulty, it sends the same message and hence all the nonfaulty preferences are the same.

Suppose a nonfaulty processor p_i uses its own majority value v for its preference. Thus p_i receives more than $n/2 + f$ messages for v in the first round of phase g. Hence every processor, including p_g receives more than $n/2$ messages for v in the first round of phase g and sets its majority value to v. Hence every nonfaulty processor has v for its preference. ■

Hence at phase $g+1$ all processors have the same preference and by Lemma 13.16 they will decide on the same value at the end of the algorithm. Hence the algorithm has the agreement property and solves consensus.

Theorem 13.18 *There exists an algorithm for n processors which solves the consensus problem in the presence of f Byzantine failures within $2(f + 1)$ rounds using constant size messages, if $n > 4f$.*

13.4.6. Impossibility in asynchronous systems

As shown before, the consensus problem can be solved in synchronous systems in the presence of both crash (benign) and Byzantine (severe) failures. What about asynchronous systems? Under the assumption that the communication system is completely reliable, and the only possible failures are caused by unreliable processors, it can be shown that if the system is completely asynchronous then there is no consensus algorithm even in the presence of only a single processor failure. The result holds even if the processors only fail by crashing. The impossibility proof relies heavily on the system being asynchronous. This result was first shown in a breakthrough paper by Fischer, Lynch and Paterson. It is one of the most influential results in distributed computing.

The impossibility holds for both shared memory systems if only read/write registers are used, and for message passing systems. The proof first shows it for shared memory systems. The result for message passing systems can then be obtained through simulation.

Theorem 13.19 *There is no consensus algorithm for a read/write asynchronous shared memory system that can tolerate even a single crash failure.*

And through simulation the following assertion can be shown.

Theorem 13.20 *There is no algorithm for solving the consensus problem in an asynchronous message passing system with n processors, one of which may fail by crashing.*

Note that these results do not mean that consensus can never be solved in

asynchronous systems. Rather the results mean that there are no algorithms that guarantee termination, agreement, and validity, in all executions. It is reasonable to assume that agreement and validity are essential, that is, if a consensus algorithm terminates, then agreement and validity are guaranteed. In fact there are efficient and useful algorithms for the consensus problem that are not guaranteed to terminate in all executions. In practice this is often sufficient because the special conditions that cause non-termination may be quite rare. Additionally, since in many real systems one can make some timing assumption, it may not be necessary to provide a solution for asynchronous consensus.

Exercises

13.4-1 Prove the correctness of algorithm CONSENSUS-CRASH.

13.4-2 Prove the correctness of the consensus algorithm in the presence of Byzantine failures.

13.4-3 Prove Theorem 13.20.

13.5. Logical time, causality, and consistent state

In a distributed system it is often useful to compute a global state that consists of the states of all processors. Having access to the global can allows us to reason about the system properties that depend on all processors, for example to be able to detect a deadlock. One may attempt to compute global state by stopping all processors, and then gathering their states to a central location. Such a method is will-suited for many distributed systems that must continue computation at all times. This section discusses how one can compute global state that is quite intuitive, yet consistent, in a precise sense. We first discuss a distributed algorithm that imposes a global order on instructions of processors. This algorithm creates the illusion of a global clock available to processors. Then we introduce the notion of one instruction causally affecting other instruction, and an algorithm for computing which instruction affects which. The notion turns out to be very useful in defining a consistent global state of distributed system. We close the section with distributed algorithms that compute a consistent global state of distributed system.

13.5.1. Logical time

The design of distributed algorithms is easier when processors have access to (Newtonian) global clock, because then each event that occurs in the distributed system can be labeled with the reading of the clock, processors agree on the ordering of any events, and this consensus can be used by algorithms to make decisions. However, construction of a global clock is difficult. There exist algorithms that approximate the ideal global clock by periodically synchronising drifting local hardware clocks. However, it is possible to totally order events without using hardware clocks. This idea is called the ***logical clock***.

Recall that an execution is an interleaving of instructions of the n programs. Each instruction can be either a computational step of a processor, or sending a message, or receiving a message. Any instruction is performed at a distinct point of

global time. However, the reading of the global clock is not available to processors. Our goal is to assign values of the logical clock to each instruction, so that these values appear to be readings of the global clock. That is, it possible to postpone or advance the instants when instructions are executed in such a way, that each instruction x that has been assigned a value t_x of the logical clock, is executed exactly at the instant t_x of the global clock, and that the resulting execution is a valid one, in the sense that it can actually occur when the algorithm is run with the modified delays.

The LOGICAL-CLOCK algorithm assigns logical time to each instruction. Each processor has a local variable called *counter*. This variable is initially zero and it gets incremented every time processor executes an instruction. Specifically, when a processor executes any instruction other than sending or receiving a message, the variable *counter* gets incremented by one. When a processor sends a message, it increments the variable by one, and attaches the resulting value to the message. When a processor receives a message, then the processor retrieves the value attached to the message, then calculates the maximum of the value and the current value of *counter*, increments the maximum by one, and assigns the result to the *counter* variable. Note that every time instruction is executed, the value of *counter* is incremented by at least one, and so it grows as processor keeps on executing instructions. The value of logical time assigned to instruction x is defined as the pair $(counter, id)$, where *counter* is the value of the variable *counter* right after the instruction has been executed, and id is the identifier of the processor. The values of logical time form a total order, where pairs are compared lexicographically. This logical time is also called Lamport time. We define t_x to be a quotient $counter + 1/(id + 1)$, which is an equivalent way to represent the pair.

Remark 13.21 *For any execution, logical time satisfies three conditions:*
(i) if an instruction x is performed by a processor before an instruction y is performed by the same processor, then the logical time of x is strictly smaller than that of y,
(ii) any two distinct instructions of any two processors get assigned different logical times,
(iii) if instruction x sends a message and instruction y receives this message, then the logical time of x is strictly smaller than that of y.

Our goal now is to argue that logical clock provides to processors the illusion of global clock. Intuitively, the reason why such an illusion can be created is that we can take any execution of a deterministic algorithm, compute the logical time t_x of each instruction x, and run the execution again delaying or speeding up processors and messages in such a way that each instruction x is executed at the instant t_x of the global clock. Thus, without access to a hardware clock or other external measurements not captured in our model, the processors cannot distinguish the reading of logical clock from the reading of a real global clock. Formally, the reason why the re-timed sequence is a valid execution that is indistinguishable from the original execution, is summarised in the subsequent corollary that follows directly from Remark 13.21.

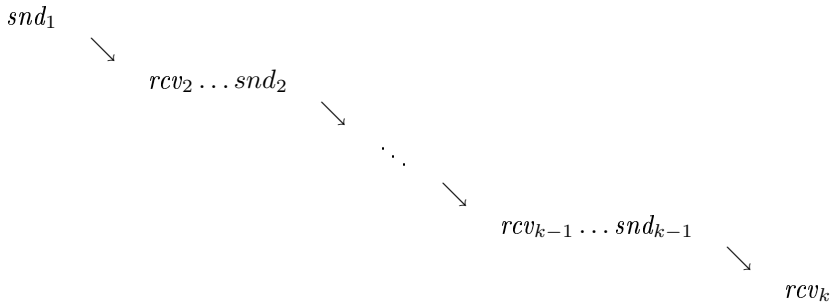
Corollary 13.22 *For any execution α, let T be the assignment of logical time to*

instructions, and let β be the sequence of instructions ordered by their logical time in α. Then for each processor, the subsequence of instructions executed by the processor in α is the same as the subsequence in β. Moreover, each message is received in β after it is sent in β.

13.5.2. Causality

In a system execution, an instruction can affect another instruction by altering the state of the computation in which the second instruction executes. We say that one instruction can ***causally*** affect (or influence) another, if the information that one instruction produces can be passed on to the other instruction. Recall that in our model of distributed system, each instruction is executed at a distinct instant of global time, but processors do not have access to the reading of the global clock. Let us illustrate causality. If two instructions are executed by the same processor, then we could say that the instruction executed earlier can causally affect the instruction executed later, because it is possible that the result of executing the former instruction was used when the later instruction was executed. We stress the word possible, because in fact the later instruction may not use any information produced by the former. However, when defining causality, we simplify the problem of capturing how processors influence other processors, and focus on what is possible. If two instructions x and y are executed by two different processors, then we could say that instruction x can causally affect instruction y, when the processor that executes x sends a message when or after executing x, and the message is delivered before or during the execution of y at the other processor. It may also be the case that influence is passed on through intermediate processors or multiple instructions executed by processors, before reaching the second processor.

We will formally define the intuition that one instruction can causally affect another in terms of a relation called ***happens before***, and that relates pairs of instructions. The relation is defined for a given execution, i.e., we fix a sequence of instructions executed by the algorithm and instances of global clock when the instructions were executed, and define which pairs of instructions are related by the happens before relation. The relation is introduced in two steps. If instructions x and y are executed by the same processor, then we say that x *happens before* y if and only if x is executed before y. When x and y are executed by two different processors, then we say that x happens before y if and only if there is a chain of instructions and messages



for $k \geq 2$, such that snd_1 is either equal to x or is executed after x by the same processor that executes x; rcv_k is either equal to y or is executed before y by the same processor that executes y; rcv_h is executed before snd_h by the same processor, $2 \leq h < k$; and snd_h sends a message that is received by rcv_{h+1} , $1 \leq h < k$. Note that no instruction happens before itself. We write $x <_{HB} y$ when x happens before y. We omit the reference to the execution for which the relation is defined, because it will be clear from the context which execution we mean. We say that two instructions x and y are *concurrent* when neither $x <_{HB} y$ nor $y <_{HB} x$. The question stands how processors can determine if one instruction happens before another in a given execution according to our definition. This question can be answered through a generalisation of the LOGICAL-CLOCK algorithm presented earlier. This generalisation is called vector clocks.

The VECTOR-CLOCKS algorithm allows processors to relate instructions, and this relation is exactly the happens before relation. Each processor p_i maintains a vector V_i of n integers. The j-th coordinate of the vector is denoted by $V_i[j]$. The vector is initialised to the zero vector $(0, \ldots, 0)$. A vector is modified each time processor executes an instruction, in a way similar to the way *counter* was modified in the LOGICAL-CLOCK algorithm. Specifically, when a processor p_i executes any instruction other than sending or receiving a message, the coordinate $V_i[i]$ gets incremented by one, and other coordinates remain intact. When a processor sends a message, it increments $V_i[i]$ by one, and attaches the resulting vector V_i to the message. When a processor p_j receives a message, then the processor retrieves the vector V attached to the message, calculates coordinate-wise maximum of the current vector V_j and the vector V, except for coordinate $V_j[j]$ that gets incremented by one, and assigns the result to the variable V_j.

$V_j[j] \leftarrow V_j[j] + 1$
for all $k \in [n] \setminus \{j\}$
    $V_j[k] \leftarrow \max\{V_j[k], V[k]\}$

We label each instruction x executed by processor p_i with the value of the vector V_i right after the instruction has been executed. The label is denoted by $VT(x)$ and is called ***vector timestamp*** of instruction x. Intuitively, $VT(x)$ represents the knowledge of processor p_i about how many instructions each processor has executed at the moment when p_i has executed instruction x. This knowledge may be obsolete.

Vector timestamps can be used to order instructions that have been executed. Specifically, given two instructions x and y, and their vector timestamps $VT(x)$ and $VT(y)$, we write that $x \leq_{VT} y$ when the vector $VT(x)$ is majorised by the vector $VT(y)$ i.e., for all k, the coordinate $VT(x)[k]$ is at most the corresponding coordinate $VT(y)[k]$. We write $x <_{VT} y$ when $x \leq_{VT} y$ but $VT(x) \neq VT(y)$.

The next theorem explains that the VECTOR-CLOCKS algorithm indeed implements the happens before relation, because we can decide if two instructions happen or not before each other, just by comparing the vector timestamps of the instructions.

Theorem 13.23 *For any execution and any two instructions x and y, $x <_{HB} y$ if and only if $x <_{VT} y$.*

Proof We first show the forward implication. Suppose that $x <_{HB} y$. Hence x and y are two different instructions. If the two instructions are executed on the same processor, then x must be executed before y. Only finite number of instructions have been executed by the time y has been executed. The VECTOR-CLOCK algorithm increases a coordinate by one as it calculates vector timestamps of instructions from x until y inclusive, and no coordinate is ever decreased. Thus $x <_{VT} y$. If x and y were executed on different processors, then by the definition of happens before relation, there must be a finite chain of instructions and messages leading from x to y. But then by the VECTOR-CLOCK algorithm, the value of a coordinate of vector timestamp gets increased at each move, as we move along the chain, and so again $x <_{VT} y$.

Now we show the reverse implication. Suppose that it is not the case that $x <_{HB} y$. We consider a few subcases always concluding that it is not that case that $x <_{VT} y$. First, it could be the case that x and y are the same instruction. But then obviously vector clocks assigned to x and y are the same, and so it cannot be the case that $x <_{VT} y$. Let us, therefore, assume that x and y are different instructions. If they are executed by the same processor, then x cannot be executed before y, and so x is executed after y. Thus, by monotonicity of vector timestamps, $y <_{VT} x$, and so it is not the case that $x <_{VT} y$. The final subcase is when x and y are executed by two distinct processors p_i and p_j. Let us focus on the component i of vector clock V_i of processor p_i right after x was executed. Let its value be k. Recall that other processors can only increase the value of their components i by adopting the value sent by other processors. Hence, in order for the value of component i of processor p_j to be k or more at the moment y is executed, there must be a chain of instructions and messages that passes a value at least k, originating at processor p_i. This chain starts at x or at an instruction executed by p_i subsequent to x. But the existence of such chain would imply that x happens before y, which we assumed was not the case. So the component i of vector clock $VT(y)$ is strictly smaller than the component i of vector clock $VT(x)$. Thus it cannot be the case that $x <_{VT} y$. ■

This theorem tells us that we can decide if two distinct instructions x and y are concurrent, by checking that it is not the case that $VT(x) < VT(y)$ nor is it the case that $VT(x) > VT(y)$.

13.5.3. Consistent state

The happens before relation can be used to compute a global state of distributed system, such that this state is in some sense consistent. Shortly, we will formally define the notion of consistency. Each processor executes instructions. A ***cut*** K is defined as a vector $K = (k_1, \ldots, k_n)$ of non-negative integers. Intuitively, the vector K denotes the states of processors. Formally, k_i denotes the number of instructions that processor p_i has executed. Not all cuts correspond to collections of states of distributed processors that could be considered natural or consistent. For example, if a processor p_i has received a message from p_j and we record the state of p_i in the cut by making k_i appropriately large, but make k_j so small that the cut contains the state of the sender before the moment when the message was sent, then we could say that such cut is not natural—there are instructions recorded in the cut

that are causally affected by instructions that are not recorded in the cut. Such cuts we consider not consistent and so undesirable. Formally, a cut $K = (k_1, \ldots, k_n)$ is inconsistent when there are processors p_i and p_j such that the instruction number k_i of processor p_i is causally affected by an instruction subsequent to instruction number k_j of processor p_j. So in an inconsistent cut there is a message that "crosses" the cut in a backward direction. Any cut that is not inconsistent is called a ***consistent cut***.

The CONSISTENT-CUT algorithm uses vector timestamps to find a consistent cut. We assume that each processor is given the same cut $K = (k_1, \ldots, k_n)$ as an input. Then processors must determine a consistent cut K' that is majorised by K. Each processor p_i has an infinite table $VT_i[0, 1, 2, \ldots]$ of vectors. Processor executes instructions, and stores vector timestamps in consecutive entries of the table. Specifically, entry m of the table is the vector timestamp $VT_i[m]$ of the m-th instruction executed by the processor; we define $VT_i[0]$ to be the zero vector. Processor p_i begins calculating a cut right after the moment when the processor has executed instruction number k_i. The processor determines the largest number $k'_i \geq 0$ that is at most k_i, such that the vector $VT_i[k'_i]$ is majorised by K. The vector $K' = (k'_1, \ldots, k'_n)$ that processors collectively find turns out to be a consistent cut.

Theorem 13.24 *For any cut K, the cut K' computed by the* CONSISTENT-CUT *algorithm is a consistent cut majorised by K.*

Proof First observe that there is no need to consider entries of VT_i further than k_i. Each of these entries is not majorised by K, because the i-th coordinate of any of these vectors is strictly larger than k_i. So we can indeed focus on searching among the first k_i entries of VT_i. Let $k'_i \geq 0$ be the largest entry such that the vector $VT_i[k'_i]$ is majorised by the vector K. We know that such vector exists, because $VT_i[0]$ is a zero vector, and such vector is majorised by any cut K.

We argue that $(k'_1, \ldots, k'_n)$ is a consistent cut by way of contradiction. Suppose that the vector $(k'_1, \ldots, k'_n)$ is an inconsistent cut. Then, by definition, there are processors p_i and p_j such that there is an instruction x of processor p_i subsequent to instruction number k'_i, such that x happens before instruction number k'_j of processor p_j. Recall that k'_i is the furthest entry of VT_i majorised by K. So entry $k'_i + 1$ is not majorised by K, and since all subsequent entries, including the one for instruction x, can have only larger coordinates, the entries are not majorised by K either. But, x happens before instruction number k'_j, so entry k'_j can only have lager coordinates than respective coordinates of the entry corresponding to x, and so $VT_j[k'_j]$ cannot be majorised by K either. This contradicts the assumption that $VT_j[k'_j]$ is majorised by K. Therefore, $(k'_1, \ldots, k'_n)$ must be a consistent cut. ■

There is a trivial algorithm for finding a consistent cut. The algorithm picks $K' = (0, \ldots, 0)$. However, the CONSISTENT-CUT algorithm is better in the sense that the consistent cut found is maximal. That this is indeed true, is left as an exercise.

There is an alternative way to find a consistent cut. The Consistent Cut algorithm requires that we attach vector timestamps to messages and remember vector timestamps for all instructions executed so far by the algorithm A which consistent cut we want to compute. This may be too costly. The algorithm called DISTRIBUTED-

SNAPSHOT avoids this cost. In the algorithm, a processor initiates the calculation of consistent cut by flooding the network with a special message that acts like a sword that cuts the execution of algorithm A consistently. In order to prove that the cut is indeed consistent, we require that messages are received by the recipient in the order they were sent by the sender. Such ordering can be implemented using sequence number.

In the DISTRIBUTED-SNAPSHOT algorithm, each processor p_i has a variable called *counter* that counts the number of instructions of algorithm A executed by the processor so far. In addition the processor has a variable k_i that will store the i-th coordinate of the cut. This variable is initialised to $\perp$. Since the variables *counter* only count the instructions of algorithm A, the instructions of DISTRIBUTED-SNAPSHOT algorithm do not affect the *counter* variables. In some sense the snapshot algorithm runs in the "background". Suppose that there is exactly one processor that can decide to take a snapshot of the distributed system. Upon deciding, the processor "floods" the network with a special message $<$*Snapshot*$>$. Specifically, the processor sends the message to all its neighbours and assigns *counter* to k_i. Whenever a processor p_j receives the message and the variable k_j is still $\perp$, then the processor sends $<$*Snapshot*$>$ message to all its neighbours and assigns *current* to k_j. The sending of $<$*Snapshot*$>$ messages and assignment are done by the processor without executing any instruction of A (we can think of DISTRIBUTED-SNAPSHOT algorithm as an "interrupt"). The algorithm calculates a consistent cut.

Theorem 13.25 *Let for any processors p_i and p_j, the messages sent from p_i to p_j be received in the order they are sent. The* DISTRIBUTED-SNAPSHOT *algorithm eventually finds a consistent cut $(k_1, \ldots, k_n)$. The algorithm sends $O(e)$ messages, where e is the number of edges in the graph.*

Proof The fact that each variable k_i is eventually different from $\perp$ follows from our model, because we assumed that instructions are eventually executed and messages are eventually received, so the $<$*Snapshot*$>$ messages will eventually reach all nodes.

Suppose that $(k_1, \ldots, k_n)$ is not a consistent cut. Then there is a processor p_j such that instruction number $k_j + 1$ or later sends a message $<M>$ other than $<$*Snapshot*$>$, and the message is received on or before a processor p_i executes instruction number k_i. So the message $<M>$ must have been sent after the message $<$*Snapshot*$>$ was sent from p_j to p_i. But messages are received in the order they are sent, so p_i processes $<$*Snapshot*$>$ before it processes $<M>$. But then message $<M>$ arrives after snapshot was taken at p_i. This is a desired contradiction. ■

Exercises

13.5-1 Show that logical time preserves the *happens before* ($<_{HB}$) relation. That is, show that if for events x and y it is the case that $x <_{HB} y$, then $LT(x) < LT(y)$, where $LT(\cdot)$ is the logical time of an event.

13.5-2 Show that any vector clock that captures concurrency between n processors must have at least n coordinates.

13.5-3 Show that the vector K' calculated by the algorithm CONSISTENT-CUT is in fact a maximal consistent cut majorised by K. That is that there is no K'' that majorises K' and is different from K', such that K'' is majorised by K.

13.6. Communication services

Among the fundamental problems in distributed systems where processors communicate by message passing are the tasks of spreading and gathering information. Many distributed algorithms for communication networks can be constructed using building blocks that implement various broadcast and multicast services. In this section we present some basic communication services in the message-passing model. Such services typically need to satisfy some quality of service requirements dealing with ordering of messages and reliability. We first focus on broadcast services, then we discuss more general multicast services.

13.6.1. Properties of broadcast services

In the broadcast problem, a selected processor p_i, called a ***source*** or a *sender*, has the message m, which must be delivered to all processors in the system (including the source). The interface of the broadcast service is specified as follows:

bc-send$_i(m, qos)$ **:** an event of processor p_i that sends a message m to all processors.

bc-recv$_i(m, j, qos)$ **:** an event of processor p_i that receives a message m sent by processor p_j.

In above definitions *qos* denotes the ***quality of service*** provided by the system. We consider two kinds of quality service:

Ordering: how the order of received messages depends on the order of messages sent by the source?

Reliability: how the set of received messages depends on the failures in the system?

The basic model of a message-passing distributed system normally does not guarantee any ordering or reliability of messaging operations. In the basic model we only assume that each pair of processors is connected by a link, and message delivery is independent on each link — the order of received messages may not be related to the order of the sent messages, and messages may be lost in the case of crashes of senders or receivers.

We present some of the most useful requirements for ordering and reliability of broadcast services. The main question we address is how to implement a stronger service on top of the weaker service, starting with the basic system model.

Variants of ordering requirements

Applying the definition of *happens before* to messages, we say that message m happens before message m' if either m and m' are sent by the same processor and m is sent before m', or the bc-recv event for m happens before the bc-send event for m'.

We identify four common broadcast services with respect to the message ordering properties:

Basic Broadcast: no order of messages is guaranteed.

Single-Source FIFO (*first-in-first-out*): messages sent by one processor are received by each processor in the same order as sent; more precisely, for all processors p_i, p_j and messages m, m', if processor p_i sends m before it sends m' then processor p_j does not receive message m' before message m.

Causal Order: messages are received in the same order as they happen; more precisely, for all messages m, m' and every processor p_i, if m happens before m' then p_i does not receive m' before m.

Total Order: the same order of received messages is preserved in each processor; more precisely, for all processors p_i, p_j and messages m, m', if processor p_i receives m before it receives m' then processor p_j does not receive message m' before message m.

It is easy to see that Causal Order implies Single-Source FIFO requirements (since the relation "happens before" for messages includes the order of messages sent by one processor), and each of the given services trivially implies Basic Broadcast. There are no additional relations between these four services. For example, there are executions that satisfy Single-Source FIFO property, but not Causal Order. Consider two processors p_0 and p_1. In the first event p_0 broadcasts message m, next processor p_1 receives m, and then p_1 broadcasts message m'. It follows that m happens before m'. But if processor p_0 receives m' before m, which may happen, then this execution violates Causal Order. Note that trivially Single-Source FIFO requirement is preserved, since each processor broadcasts only one message.

We denote by *bb* the Basic Broadcast service, by *ssf* the Single-Source FIFO, by *co* the Causal Order and by *to* the Total Order service.

Reliability requirements

In the model without failures we would like to guarantee the following properties of broadcast services:

Integrity: each message m received in event bc-recv has been sent in some bc-send event.

No-Duplicates: each processor receives a message not more than once.

Liveness: each message sent is received by all processors.

In the model with failures we define the notion of *reliable* broadcast service, which satisfies Integrity, No-Duplicates and two kinds of Liveness properties:

Nonfaulty Liveness: each message m sent by non-faulty processor p_i must be received by every non-faulty processor.

Faulty Liveness: each message sent by a faulty processor is either received by all non-faulty processors or by none of them.

We denote by *rbb* the **R**eliable **B**asic **B**roadcast service, by *rssf* the **R**eliable **S**ingle-**S**ource **FIFO**, by *rco* the **R**eliable **C**ausal **O**rder, and by *rto* the **R**eliable **T**otal **O**rder service.

13.6.2. Ordered broadcast services

We now describe implementations of algorithms for various broadcast services.

Implementing basic broadcast on top of asynchronous point-to-point messaging

The *bb* service is implemented as follows. If event bc-send$_i(m, bb)$ occurs then processor p_i sends message m via every link from p_i to p_j, where $0 \leq i \leq n-1$. If a message m comes to processor p_j then it enables event bc-recv$_j(m, i, bb)$.

To provide reliability we do the following. We build the reliable broadcast on the top of basic broadcast service. When bc-send$_i(m, rbb)$ occurs, processor p_i enables event bc-send$_i(\langle m, i\rangle, bb)$. If event bc-recv$_j(\langle m, i\rangle, k, bb)$ occurs and message-coordinate m appears for the first time then processor p_j first enables event bc-send$_j(\langle m, i\rangle, bb)$ (to inform other non-faulty processors about message m in case when processor p_i is faulty), and next enables event bc-recv$_j(m, i, rbb)$.

We prove that the above algorithm provides reliability for the basic broadcast service. First observe that Integrity and No-Duplicates properties follow directly from the fact that each processor p_j enables bc-recv$_j(m, i, rbb)$ only if message-coordinate m is received for the first time. Nonfaulty liveness is preserved since links between non-faulty processors enables events bc-recv$_j(\cdot, \cdot, bb)$ correctly. Faulty Liveness is guaranteed by the fact that if there is a non-faulty processor p_j which receives message m from the faulty source p_i, then before enabling bc-recv$_j(m, i, rbb)$ processor p_j sends message m using bc-send$_j$ event. Since p_j is non-faulty, each non-faulty processor p_k gets message m in some bc-recv$_k(\langle m, i\rangle, \cdot, bb)$ event, and then accepts it (enabling event bc-recv$_k(m, i, rbb)$) during the first such event.

Implementing single-source FIFO on top of basic broadcast service

Each processor p_i has its own counter (timestamp), initialised to 0. If event bc-send$_i(m, ssf)$ occurs then processor p_i sends message m with its current timestamp attached, using bc-send$_i(< m, timestamp >, bb)$. If an event bc-recv$_j(< m, t >, i, bb)$ occurs then processor p_j enables event bc-recv$_j(m, i, ssf)$ just after events bc-recv$_j(m_0, i, ssf)$, ..., bc-recv$_j(m_{t-1}, i, ssf)$ have been enabled, where $m_0, \ldots, m_{t-1}$ are the messages such that events bc-recv$_j(< m_0, 0 >, i, bb)$,...,bc-recv$_j(< m_{t-1}, t-1 >, i, bb)$ have been enabled.

Note that if we use reliable Basic Broadcast instead of Basic Broadcast as the background service, the above implementation of Single-Source FIFO becomes Reliable Single-Source FIFO service. We leave the proof to the reader as an exercise.

Implementing causal order and total order on the top of single-source FIFO service

We present an ordered broadcast algorithm which works in the asynchronous message-passing system providing single-source FIFO broadcast service. It uses the idea of timestamps, but in more advanced way than in the implementation of *ssf*. We denote by *cto* the service satisfying causal and total orders requirements.

Each processor p_i maintains in a local array T its own increasing counter (timestamp), and the estimated values of timestamps of other processors. Timestamps are used to mark messages before sending—if p_i is going to broadcast a message, it increases its timestamp and uses it to tag this message (lines 11-13). During the execution processor p_i estimates values of timestamps of other processors in the local vector T—if processor p_i receives a message from processor p_j with a tag t (times-

tamp of p_j), it puts t into $T[j]$ (lines 23–32). Processor p_i sets its current timestamp to be the maximum of the estimated timestamps in the vector T plus one (lines 24–26). After updating the timestamp processor sends an update message. Processor accepts a message m with associated timestamp t from processor j if pair (t, j) is the smallest among other received messages (line 42), and each processor has at least as large a timestamp as known by processor p_i (line 43). The details are given in the code below.

ORDERED-BROADCAST

```
     Code for any processor p_i, 0 ≤ i ≤ n − 1
01   initialisation
02       T[j] ← 0 for every 0 ≤ j ≤ n − 1

11   if bc-send_i(m, cto) occurs
12      then T[i] ← T[i] + 1
13           enable bc-send_i(< m, T[i] >, ssf)

21   if bc-recv_i(< m, t >, j, ssf) occurs
22      then add triple (m, t, j) to pending
23           T[j] ← t
24      if t > T[i]
25         then T[i] ← t
26              enable bc-send_i(< update, T[i] >, ssf)
31   if bc-recv_i(< update, t >, j, ssf) occurs
32      then T[j] ← t
41   if
42      (m, t, j) is the pending triple with the smallest (t, j) and
        t ≤ T[k] for every 0 ≤ k ≤ n − 1
43   then enable bc-recv_i(m, j, cto)
44        remove triple (m, t, j) from pending
```

ORDERED-BROADCAST satisfies the causal order requirement. We leave the proof to the reader as an exercise (in the latter part we show how to achieve stronger reliable causal order service and provide the proof for that stronger case).

Theorem 13.26 ORDERED-BROADCAST *satisfies the total order requirement.*

Proof Integrity follows from the fact that each processor can enable event bc-recv$_i(m, j, cto)$ only if the triple (m, t, j) is pending (lines 41–45), which may happen after receiving a message m from processor j (lines 21–22). No-Duplicates property is guaranteed by the fact that there is at most one pending triple containing message m sent by processor j (lines 13 and 21–22).

Liveness follows from the fact that each pending triple satisfies conditions in lines 42–43 in some moment of the execution. The proof of this fact is by induction on the events in the execution — suppose to the contrary that (m, t, j) is the triple with smallest (t, j) which does not satisfy conditions in lines 42–43 at any moment of the execution. It follows that there is a moment from which triple (m, t, j) has

smallest (t, j) coordinates among pending triples in processor p_i. Hence, starting from this moment, it must violate condition in line 43 for some k. Note that $k \neq i, j$, by updating rules in lines 23–25. It follows that processor p_i never receives a message from p_k with timestamp greater than $t - 1$, which by updating rules in lines 24-26 means that processor p_k never receives a message $< m, t >$ from j, which contradicts the liveness property of ssf broadcast service.

To prove Total Order property it is sufficient to prove that for every processor p_i and messages m, m' sent by processors p_k, p_l with timestamps t, t' respectively, each of the triples (m, t, k), (m', t', l) are accepted according to the lexicographic order of $(t, k), (t', l)$. There are two cases.

Case 1. Both triples are pending in processor p_i at some moment of the execution. Then condition in line 42 guarantees acceptance in order of $(t, k), (t', l)$.

Case 2. Triple (m, t, k) (without loss of generality) is accepted by processor p_i before triple (m', t', l) is pending. If $(t, k) < (t', l)$ then still the acceptance is according to the order of $(t, k), (t', l)$. Otherwise $(t, k) > (t', l)$, and by condition in line 43 we get in particular that $t \leq T[l]$, and consequently $t' \leq T[l]$. This can not happen because of the ssf requirement and the assumption that processor p_i has not yet received message $< m', t' >$ from l via the ssf broadcast service. ■

Now we address reliable versions of Causal Order and Total Order services. A Reliable Causal Order requirements can be implemented on the top of Reliable Basic Broadcast service in asynchronous message-passing system with processor crashes using the following algorithm. It uses the same data structures as previous ORDERED-BBROADCAST. The main difference between reliable CAUSALLY-ORDERED-BROADCAST and ORDERED-BROADCAST are as follows: instead of using integer timestamps processors use vector timestamps T, and they do not estimate timestamps of other processors, only compare in lexicographic order their own (vector) timestamps with received ones. The intuition behind vector timestamp of processor p_i is that it stores information how many messages have been sent by p_i and how many have been accepted by p_i from every p_k, where $k \neq i$.

In the course of the algorithm processor p_i increases corresponding position i in its vector timestamp T before sending a new message (line 12), and increases jth position of its vector timestamp after accepting new message from processor p_j (line 38). After receiving a new message from processor p_j together with its vector timestamp $\hat{T}$, processor p_i adds triple $(m, \hat{T}, j)$ to pending and accepts this triple if it is first not accepted message received from processor p_j (condition in line 33) and the number of accepted messages (from each processor $p_k \neq p_i$) by processor p_j was not bigger in the moment of sending m than it is now in processor p_i (condition in line 34). Detailed code of the algorithm follows.

RELIABLE-CAUSALLY-ORDERED-BROADCAST

Code for any processor p_i, $0 \leq i \leq n-1$

01 **initialisation**
02 $T[j] \leftarrow 0$ for every $0 \leq j \leq n-1$
03 pending list is empty

11 **if** bc-send$_i(m, rco)$ occurs
12 **then** $T[i] \leftarrow T[i] + 1$
13 **enable** bc-send$_i(< m, T >, rbb)$

21 **if** bc-recv$_i(< m, \hat{T} >, j, rbb)$ occurs
22 **then add** triple $(m, \hat{T}, j)$ to pending

31 **if** $(m, \hat{T}, j)$ is the pending triple, and
32 $\hat{T}[j] = T[j] + 1$, and
33 $\hat{T}[k] \leq T[k]$ for every $k \neq i$
34 **then enable** bc-recv$_i(m, j, rco)$
35 **remove** triple $(m, \hat{T}, j)$ from pending
36 $T[j] \leftarrow T[j] + 1$

We argue that the algorithm RELIABLE-CAUSALLY-ORDERED-BROADCAST provides Reliable Causal Order broadcast service on the top of the system equipped with the Reliable Basic Broadcast service. Integrity and No-Duplicate properties are guaranteed by *rbb* broadcast service and facts that each message is added to pending at most once and non-received message is never added to pending. Nonfaulty and Faulty Liveness can be proved by one induction on the execution, using facts that non-faulty processors have received all messages sent, which guarantees that conditions in lines 33-34 are eventually satisfied. Causal Order requirement holds since if message m happens before message m' then each processor p_i accepts messages m, m' according to the lexicographic order of $\hat{T}, \hat{T}'$, and these vector-arrays are comparable in this case. Details are left to the reader.

Note that Reliable Total Order broadcast service can not be implemented in the general asynchronous setting with processor crashes, since it would solve consensus in this model — first accepted message would determine the agreement value (against the fact that consensus is not solvable in the general model).

13.6.3. Multicast services

Multicast services are similar to the broadcast services, except each multicast message is destined for a specified subset of all processors.In the multicast service we provide two types of events, where *qos* denotes a quality of service required:

mc-send$_i(m, D,$ ***qos***$)$ **:** an event of processor p_i which sends a message m together with its id to all processors in a destination set $D \subseteq \{0, \ldots, n-1\}$.

mc-recv$_i(m, j,$ ***qos***$)$ **:** an event of processor p_i which receives a message m sent by processor p_j.

Note that the event mc-recv is similar to bc-recv.

As in case of a broadcast service, we would like to provide useful ordering and reliable properties of the multicast services. We can adapt ordering requirements from the broadcast services. Basic Multicast does not require any ordering properties. Single-Source FIFO requires that if one processor multicasts messages (possibly to different destination sets), then the messages received in each processors (if any) must be received in the same order as sent by the source. Definition of Causal Order remains the same. Instead of Total Order, which is difficult to achieve since destination sets may be different, we define another ordering property:

Sub-Total Order: orders of received messages in all processors may be extended to the total order of messages; more precisely, for any messages m, m' and processors p_i, p_j, if p_i and p_j receives both messages m, m' then they are received in the same order by p_i and p_j.

The reliability conditions for multicast are somewhat different from the conditions for reliable broadcast.

Integrity: each message m received in event mc-recv$_i$ was sent in some mc-send event with destination set containing processor p_i.

No Duplicates: each processor receives a message not more than once.

Nonfaulty Liveness: each message m sent by non-faulty processor p_i must be received in every non-faulty processor in the destination set.

Faulty Liveness: each message sent by a faulty processor is either received by all non-faulty processors in the destination set or by none of them.

One way of implementing ordered and reliable multicast services is to use the corresponding broadcast services (for Sub-Total Order the corresponding broadcast requirement is Total Order). More precisely, if event mc-send$_i(m, D, qos)$ occurs processor p_i enables event bc-send$_i(< m, D >, qos)$. When an event bc-recv$_j(< m, D >, i, qos)$ occurs, processor p_j enables event mc-recv$_j(m, i, qos)$ if $p_j \in D$, otherwise it ignores this event. The proof that such method provides required multicast quality of service is left as an exercise.

13.7. Rumor collection algorithms

Reliable multicast services can be used as building blocks in constructing algorithms for more advanced communication problems. In this section we illustrate this method for the problem of collecting rumors by synchronous processors prone to crashes. (Since we consider only fair executions, we assume that at least one processor remains operational to the end of the computation).

13.7.1. Rumor collection problem and requirements

The classic problem of ***collecting rumors***, or ***gossip***, is defined as follows:

> At the beginning, each processor has its distinct piece of information, called a *rumor,* the goal is to make every processor know all the rumors.

However in the model with processor crashes we need to re-define the gossip problem to respect crash failures of processors. Both Integrity and No-Duplicates properties are the same as in the reliable broadcast service, the only difference (which follows from the specification of the gossip problem) is in Liveness requirements:

Non-faulty Liveness: the rumor of every non-faulty processor must be known by each non-faulty processor.

Faulty Liveness: if processor p_i has crashed during execution then each non-faulty processor either knows the rumor of p_i or knows that p_i is crashed.

The efficiency of gossip algorithms is measured in terms of time and message complexity. Time complexity measures number of (synchronous) steps from the beginning to the termination. Message complexity measures the total number of point-to-point messages sent (more precisely, if a processor sends a message to three other processors in one synchronous step, it contributes three to the message complexity).

The following simple algorithm completes gossip in just one synchronous step: each processor broadcasts its rumor to all processors. The algorithm is correct, because each message received contains a rumor, and a message not received means the failure of its sender. A drawback of such a solution is that a quadratic number of messages could be sent, which is quite inefficient.

We would like to perform gossip not only quickly, but also with fewer point-to-point messages. There is a natural trade-off between time and communication. Note that in the system without processor crashes such a trade-off may be achieved, e.g., sending messages over the (almost) complete binary tree, and then time complexity is $O(\lg n)$, while the message complexity is $O(n \lg n)$. Hence by slightly increasing time complexity we may achieve almost linear improvement in message complexity. However, if the underlying communication network is prone to failures of components, then irregular failure patterns disturb a flow of information and make gossiping last longer. The question we address in this section is what is the best trade-off between time and message complexity in the model with processor crashes?

13.7.2. Efficient gossip algorithms

In this part we describe the family of gossip algorithms, among which we can find some efficient ones. They are all based on the same generic code, and their efficiency depends on the quality of two data structures put in the generic algorithm. Our goal is to prove that we may find some of those data structures that obtained algorithm is always correct, and efficient if the number of crashes in the execution is at most f, where $f \leq n - 1$ is a parameter.

We start with description of these structures: communication graph and communication schedules.

Communication graph

A graph $G = (V, E)$ consists of a set V of *vertices* and a set E of *edges.* Graphs in this paper are always ***simple***, which means that edges are pairs of vertices, with

no direction associated with them. Graphs are used to describe communication patterns. The set V of vertices of a graph consists of the processors of the underlying distributed system. Edges in E determine the pairs of processors that communicate directly by exchanging messages, but this does not necessarily mean an existence of a physical link between them. We abstract form the communication mechanism: messages that are exchanged between two vertices connected by an edge in E may need to be routed and traverse a possibly long path in the underlying physical communication network. Graph topologies we use, for a given number n of processors, vary depending on an upper bound f on the number of crashes we would like to tolerate in an execution. A graph that matters, at a given point in an execution, is the one induced by the processors that have not crashed till this step of the execution.

To obtain an efficient gossip algorithm, communication graphs should satisfy some suitable properties, for example the following property $\mathcal{R}(n, f)$:

Definition 13.27 *Let $f < n$ be a pair of positive integers. Graph G is said to* satisfy property $\mathcal{R}(n, f)$, *if G has n vertices, and if, for each subgraph $R \subseteq G$ of size at least $n - f$, there is a subgraph $P(R)$ of G, such that the following hold:*

1 : $P(R) \subseteq R$
2 : $|P(R)| = |R|/7$
3 : *The diameter of $P(R)$ is at most* $2 + 30 \ln n$
4 : *If $R_1 \subseteq R_2$, then $P(R_1) \subseteq P(R_2)$*

In the above definition, clause (1.) requires the existence of subgraphs $P(R)$ whose vertices has the potential of (informally) inheriting the properties of the vertices of R, clause (2.) requires the subgraphs to be sufficiently large, linear in size, clause (3.) requires the existence of paths in the subgraphs that can be used for communication of at most logarithmic length, and clause (4.) imposes monotonicity on the required subgraphs.

Observe that graph $P(R)$ is connected, even if R is not, since its diameter is finite. The following result shows that graphs satisfying property $\mathcal{R}(n, f)$ can be constructed, and that their degree is not too large.

Theorem 13.28 *For each $f < n$, there exists a graph $G(n, f)$ satisfying property $\mathcal{R}(n, f)$. The maximum degree Δ of graph $G(n, f)$ is $O\left(\frac{n}{n-f}\right)^{1.837}$.*

Communication schedules
A *local permutation* is a permutation of all the integers in the range $[0 \,..\, n-1]$. We assume that prior the computation there is given set Π of n local permutations. Each processor p_i has such a permutation π_i from Π. For simplicity we assume that $\pi_i(0) = p_i$. Local permutation is used to collect rumor in systematic way according to the order given by this permutation, while communication graphs are rather used to exchange already collected rumors within large and compact non-faulty graph component.

Generic algorithm
We start with specifying a goal that gossiping algorithms need to achieve. We say

that *processor p_i has heard about processor p_j* if either p_i knows the original input rumor of p_j or p knows that p_j has already failed. We may reformulate correctness of a gossiping algorithm in terms of hearing about other processors: algorithm is correct if Integrity and No-Duplicates properties are satisfied and if each processor has hard about any other processor by the termination of the algorithm.

The code of a gossiping algorithm includes objects that depend on the number n of processors in the system, and also on the bound $f < n$ on the number of failures which are "efficiently tolerated" (if the number of failures is at most f then message complexity of design algorithm is small). The additional parameter is a termination threshold τ which influences time complexity of the specific implementation of the generic gossip scheme. Our goal is to construct the generic gossip algorithm which is correct for any additional parameters f, τ and any communication graph and set of schedules, while efficient for some values f, τ and structures $G(n, f)$ and Π.

Each processor starts gossiping as a *collector*. Collectors seek actively information about rumors of the other processors, by sending direct inquiries to some of them. A collector becomes a ***disseminator*** after it has heard about all the processors. Processors with this status disseminate their knowledge by sending local views to selected other processors.

Local views. Each processor p_i starts with knowing only its ID and its input information $rumor_i$. To store incoming data, processor p_i maintains the following arrays:

$$\texttt{Rumors}_i, \texttt{Active}_i \text{ and } \texttt{Pending}_i,$$

each of size n. All these arrays are initialised to store the value `nil`. For an array $\texttt{X}_i$ of processor p_i, we denote its jth entry by $\texttt{X}_i[j]$ - intuitively this entry contains some information about processor p_j. The array `Rumor` is used to store all the rumors that a processor knows. At the start, processor p_i sets $\texttt{Rumors}_i[i]$ to its own input $rumor_i$. Each time processor p_i learns some $rumor_j$, it immediately sets $\texttt{Rumors}_i[j]$ to this value. The array `Active` is used to store a set of all the processors that the owner of the array knows as crashed. Once processor p_i learns that some processor p_j has failed, it immediately sets $\texttt{Active}_i[j]$ to *failed*. Notice that processor p_i has heard about processor p_j, if one among the values $\texttt{Rumors}_i[j]$ and $\texttt{Active}_i[j]$ is not equal to NIL.

The purpose of using the array `Pending` is to facilitate dissemination. Each time processor p_i learns that some other processor p_j is fully informed, that is, it is either a disseminator itself or has been notified by a disseminator, then it marks this information in $\texttt{Pending}_i[j]$. Processor p_i uses the array $\texttt{Pending}_i$ to send dissemination messages in a systematic way, by scanning $\texttt{Pending}_i$ to find those processors that possibly still have not heard about some processor.

The following is a useful terminology about the current contents of the arrays `Active` and `Pending`. Processor p_j is said *to be active according to p_i,* if p_i has not yet received any information implying that p_j crashed, which is the same as having `nil` in $\texttt{Active}_i[j]$. Processor p_j is said *to need to be notified by p_i* if it is active according to p_i and $\texttt{Pending}_i[j]$ is equal to `nil`.

Phases. An execution of a gossiping algorithm starts with the processors initialising all the local objects. Processor p_i initialises its list $\texttt{Rumors}_i$ with `nil` at all the

locations, except for the ith one, which is set equal to $rumor_i$. The remaining part of execution is structured as a loop, in which phases are iterated. Each phase consists of three parts: receiving messages, local computation, and multicasting messages. Phases are of two kinds: ***regular phase*** and ***ending phase***. During regular phases processor: receives messages, updates local knowledge, checks its status, sends its knowledge to neighbours in communication graphs as well as inquiries about rumors and replies about its own rumor. During ending phases processor: receives messages, sends inquiries to all processors from which it has not heard yet, and replies about its own rumor. The regular phases are performed τ times; the number τ is a *termination threshold.* After this, the ending phase is performed four times. This defines a generic gossiping algorithm.

GENERIC-GOSSIP

```
    Code for any processor p_i, 0 ≤ i ≤ n − 1
01  initialisation
02      processor p_i becomes a collector
03      initialisation of arrays Rumors_i, Active_i and Pending_i

11  repeat τ times
12      perform regular phase

20  repeat 4 times
21      perform ending phase
```

Now we describe communication and kinds of messages used in regular and ending phases.

Graph and range messages used during regular phases. A processor p_i may send a message to its neighbour in the graph $G(n, f)$, provided that it is is still active according to p_i. Such a message is called a ***graph*** one. Sending these messages only is not sufficient to complete gossiping, because the communication graph may become disconnected as a result of node crashes. Hence other messages are also sent, to cover all the processors in a systematic way. In this kind of communication processor p_i considers the processors as ordered by its local permutation π_i, that is, in the order $\pi_i(0), \pi_i(1), \ldots, \pi_i(n-1)$. Some of additional messages sent in this process are called ***range*** ones.

During regular phase processors send the following kind of range messages: inquiring, reply and notifying messages. A collector p_i sends an ***inquiring*** message to the first processor about which p_i has not heard yet. Each recipient of such a message sends back a range message that is called a ***reply*** one.

Disseminators send range messages also to subsets of processors. Such messages are called ***notifying*** ones. The target processor selected by disseminator p_i is the first one that still needs to be notified by p_i. Notifying messages need not to be replied to: a sender already knows the rumors of all the processors, that are active according to it, and the purpose of the message is to disseminate this knowledge.

REGULAR-PHASE

```
     Code for any processor p_i, 0 ≤ i ≤ n − 1
01   receive messages

11   perform local computation
12       update the local arrays
13       if p_i is a collector, that has already heard about all the processors
14          then p_i becomes a disseminator
15       compute set of destination processors: for each processor p_j
16          if p_j is active according to p_i and p_j is a neighbour of p_i in graph G(n, t)
17             then add p_j to destination set for a graph message
18          if p_i     is a collector and p_j is the first processor
                       about which p_i has not heard yet
19             then send an inquiring message to p_j
20          if p_i is a disseminator and p_j is the first processor
                  that needs to be notified by p_i
21             then send a notifying message to p_j
22          if p_j is a collector, from which an inquiring message was received
                  in the receiving step of this phase
23             then send a reply message to p_j

30   send graph/inquiring/notifying/reply messages to corresponding destination sets
```

Last-resort messages used during ending phases. Messages sent during the ending phases are called *last-resort* ones. These messages are categorised into inquiring, replying, and notifying, similarly as the corresponding range ones, which is because they serve a similar purpose. Collectors that have not heard about some processors yet send direct inquiries to *all* of these processors simultaneously. Such messages are called *inquiring* ones. They are replied to by the non-faulty recipients in the next step, by way of sending *reply* messages. This phase converts *all* the collectors into disseminators. In the next phase, each disseminator sends a message to *all* the processors that need to be notified by it. Such messages are called *notifying* ones.

The number of graph messages, sent by a processor at a step of the regular phase, is at most as large as the maximum node degree in the communication graph. The number of range messages, sent by a processor in a step of the regular phase, is at most as large as the number of inquiries received plus a constant - hence the global number of point-to-point range messages sent by all processors during regular phases may be accounted as a constant times the number of inquiries sent (which is one per processor per phase). In contrast to that, there is no *a priori* upper bound on the number of messages sent during the ending phase. By choosing the termination threshold τ to be large enough, one may control how many rumors still needs to be collected during the ending phases.

Updating local view. A message sent by a processor carries its current local knowledge. More precisely, a message sent by processor p_i brings the following: the ID p_i, the arrays $\texttt{Rumors}_i$, $\texttt{Active}_i$, and $\texttt{Pending}_i$, and a label to notify the

recipient about the character of the message. A label is selected from the following: *graph_message, inquiry_from_collector, notification_from_disseminator, this_is_a_reply*, their meaning is self-explanatory. A processor p_i scans a newly received message from some processor p_j to learn about rumors, failures, and the current status of other processors. It copies each rumor from the received copy of $\texttt{Rumors}_j$ into $\texttt{Rumors}_i$, unless it is already there. It sets $\texttt{Active}_i[k]$ to *failed,* if this value is at $\texttt{Active}_j[k]$. It sets $\texttt{Pending}_i[k]$ to *done,* if this value is at $\texttt{Pending}_j[k]$. It sets $\texttt{Pending}_i[j]$ to *done,* if p_j is a disseminator and the received message is a range one. If p_i is itself a disseminator, then it sets $\texttt{Pending}_i[j]$ to *done* immediately after sending a range message to p_j. If a processor p_i expects a message to come from processor p_j, for instance a graph one from a neighbour in the communication graph, or a reply one, and the message does not arrive, then p_i knows that processor p_j has failed, and it immediately sets $\texttt{Active}_i[j]$ to *failed.*

ENDING-PHASE

```
    Code for any processor p_i, 0 ≤ i ≤ n − 1
01  receive messages

11  perform local computation
12      update the local arrays
13      if p_i is a collector, that has already heard about all the processors
14          then p_i becomes a disseminator
15      compute set of destination processors: for each processor p_j
16          if p_i is a collector and it has not heard about p_j yet
17              then send an inquiring message to p_j
18          if p_i is a disseminator and p_j needs to be notified by p_i
19              then send a notifying message to p_j
20          if an inquiring message was received from p_j
                in the receiving step of this phase
21              then send a reply message to p_j

30  send inquiring/notifying/reply messages to corresponding destination sets
```

Correctness. Ending phases guarantee correctness, as is stated in the next fact.

Lemma 13.29 GENERIC-GOSSIP *is correct for every communication graph* $G(n, f)$ *and set of schedules* Π.

Proof Integrity and No-Duplicates properties follow directly from the code and the multicast service in synchronous message-passing system. It remains to prove that each processor has heard about all processors. Consider the step just before the first ending phases. If a processor p_i has not heard about some other processor p_j yet, then it sends a last-resort message to p_j in the first ending phase. It is replied to in the second ending phase, unless processor p_j has crashed already. In any case, in the third ending phase, processor p_i either learns the input rumor of p_j or it gets to know that p_j has failed. The fourth ending phase provides an opportunity to receive notifying messages, by all the processors that such messages were sent to by p_i. ■

The choice of communication graph $G(n, f)$, set of schedules Π and termination threshold τ influences however time and message complexities of the specific implementation of Generic Gossip Algorithm. First consider the case when $G(n, f)$ is a communication graph satisfying property $\mathcal{R}(n, f)$ from Definition 13.27, Π contains n random permutations, and $\tau = c \log^2 n$ for sufficiently large positive constant c. Using Theorem 13.28 we get the following result.

Theorem 13.30 *For every n and $f \leq c \cdot n$, for some constant $0 \leq c < 1$, there is a graph $G(n, f)$ such that the implementation of the generic gossip scheme with $G(n, f)$ as a communication graph and a set Π of random permutations completes gossip in expected time $O(\log^2 n)$ and with expected message complexity $O(n \log^2 n)$, if the number of crashes is at most f.*

Consider a small modification of Generic Gossip scheme: during regular phase every processor p_i sends an inquiring message to the first Δ (instead of one) processors according to permutation π_i, where Δ is a maximum degree of used communication graph $G(n, f)$. Note that it does not influence the asymptotic message complexity, since besides inquiring messages in every regular phase each processor p_i sends Δ graph messages.

Theorem 13.31 *For every n there are parameters $f \leq n-1$ and $\tau = O(\log^2 n)$ and there is a graph $G(n, f)$ such that the implementation of the modified Generic Gossip scheme with $G(n, f)$ as a communication graph and a set Π of random permutations completes gossip in expected time $O(\log^2 n)$ and with expected message complexity $O(n^{1.838})$, for any number of crashes.*

Since in the above theorem set Π is selected prior the computation, we obtain the following existential deterministic result.

Theorem 13.32 *For every n there are parameters $f \leq n - 1$ and $\tau = O(\lg n)$ and there are graph $G(n, f)$ and set of schedules Π such that the implementation of the modified Generic Gossip scheme with $G(n, f)$ as a communication graph and schedules Π completes gossip in time $O(\lg n)$ and with message complexity $O(n^{1.838})$, for any number of crashes.*

Exercises

13.7-1 Design executions showing that there is no relation between Causal Order and Total Order and between Single-Source FIFO and Total Order broadcast services. For simplicity consider two processors and two messages sent.

13.7-2 Does broadcast service satisfying Single-Source FIFO and Causal Order requirements satisfy a Total Order property? Does broadcast service satisfying Single-Source FIFO and Total Order requirements satisfy a Causal Order property? If yes provide a proof, if not show a counterexample.

13.7-3 Show that using reliable Basic Broadcast instead of Basic Broadcast in the implementation of Single-Source FIFO service, then we obtain reliable Single-Source FIFO broadcast.

13.7-4 Prove that the Ordered Broadcast algorithm implements Causal Order service on a top of Single-Source FIFO one.
13.7-5 What is the total number of point-to-point messages sent in the algorithm Ordered-Broadcast in case of k broadcasts?
13.7-6 Estimate the total number of point-to-point messages sent during the execution of Reliable-Causally-Ordered-Broadcast, if it performs k broadcast and there are $f < n$ processor crashes during the execution.
13.7-7 Show an execution of the algorithm Reliable-Causally-Ordered-Broadcast which violates Total Order requirement.
13.7-8 Write a code of the implementation of reliable Sub-Total Order multicast service.
13.7-9 Show that the described method of implementing multicast services on the top of corresponding broadcast services is correct.
13.7-10 Show that the random graph $G(n, f)$ - in which each node selects independently at random $\frac{n}{n-f} \log n$ edges from itself to other processors - satisfies property $\mathcal{R}(n, f)$ from Definition 13.27 and has degree $O(\frac{n}{n-f} \lg n)$ with probability at least $1 - O(1/n)$.
13.7-11 Leader election problem is as follows: all non-faulty processors must elect one non-faulty processor in the same synchronous step. Show that leader election can not be solved faster than gossip problem in synchronous message-passing system with processors crashes.

13.8. Mutual exclusion in shared memory

We now describe the second main model used to describe distributed systems, the shared memory model. To illustrate algorithmic issues in this model we discuss solutions for the mutual exclusion problem.

13.8.1. Shared memory systems

The shared memory is modeled in terms of a collection of *shared variables*, commonly referred to as *registers*. We assume the system contains n processors, $p_0, \ldots, p_{n-1}$, and m registers $R_0, \ldots, R_{m-1}$. Each processor is modeled as a state machine. Each register has a *type*, which specifies:

1. the values it can hold,
2. the operations that can be performed on it,
3. the value (if any) to be returned by each operation, and
4. the new register value resulting from each operation.

Each register can have an initial value.

For example, an integer valued read/write register R can take on all integer values and has operations *read(R,v)* and *write(R,v)*. The read operation returns the value v of the last preceding write, leaving R unchanged. The *write(R,v)* operation has an integer parameter v, returns no value and changes R's value to v. A *configu-*

ration is a vector $C = (q_0, \ldots, q_{n-1}, r_0, \ldots, r_{m-1})$, where q_i is a state of p_i and r_j is a value of register R_j. The *events* are computation steps at the processors where the following happens atomically (indivisibly):

1. p_i chooses a shared variable to access with a specific operation, based on p_i's current state,
2. the specified operation is performed on the shared variable,
3. p_i's state changes based on its transition function, based on its current state and the value returned by the shared memory operation performed.

A finite sequence of configurations and events that begins with an initial configuration is called an *execution*. In the asynchronous shared memory system, an infinite execution is admissible if it has an infinite number of computation steps.

13.8.2. The mutual exclusion problem

In this problem a group of processors need to access a shared resource that cannot be used simultaneously by more than a single processor. The solution needs to have the following two properties. (1) *Mutual exclusion*: Each processor needs to execute a code segment called a *critical section* so that at any given time at most one processor is executing it (i.e., is in the critical section). (2) *Deadlock freedom*: If one or more processors attempt to enter the critical section, then one of them eventually succeeds as long as no processor stays in the critical section forever. These two properties do not provide any individual guarantees to any processor. A stronger property is (3) *No lockout*: A processor that wishes to enter the critical section eventually succeeds as long as no processor stays in the critical section forever. Original solutions to this problem relied on special synchronisation support such as semaphores and monitors. We will present some of the *distributed solutions* using only ordinary shared variables.

We assume the program of a processor is partitioned into the following sections:

- **Entry / Try**: the code executed in preparation for entering the critical section.
- **Critical**: the code to be protected from concurrent execution.
- **Exit**: the code executed when leaving the critical section.
- **Remainder**: the rest of the code.

A processor cycles through these sections in the order: remainder, entry, critical and exit. A processor that wants to enter the critical section first executes the entry section. After that, if successful, it enters the critical section. The processor releases the critical section by executing the exit section and returning to the remainder section. We assume that a processor may transition any number of times from the remainder to the entry section. Moreover, variables, both shared and local, accessed in the entry and exit section are not accessed in the critical and remainder section. Finally, no processor stays in the critical section forever. An algorithm for a shared memory system solves the mutual exclusion problem with no deadlock (or no lockout) if the following hold:

- **Mutual Exclusion:** In every configuration of every execution at most one processor is in the critical section.
- **No deadlock:** In every admissible execution, if some processor is in the entry section in a configuration, then there is a later configuration in which *some* processor is in the critical section.
- **No lockout:** In every admissible execution, if some processor is in the entry section in a configuration, then there is a later configuration in which *that same* processor is in the critical section.

In the context of mutual exclusion, an execution is *admissible* if for every processor p_i, p_i either takes an infinite number of steps or p_i ends in the remainder section. Moreover, no processor is ever stuck in the exit section (unobstructed exit condition).

13.8.3. Mutual exclusion using powerful primitives

A single bit suffices to guarantee mutual exclusion with no deadlock if a powerful test&set register is used. A test&set variable V is a binary variable which supports two atomic operations, test&set and reset, defined as follows:

```
test&set(V: memory address) returns binary value:
    temp ← V
    V ← 1
    return (temp)
reset(V: memory address):
    V ← 0
```

The test&set operation atomically reads and updates the variable. The reset operation is merely a write. There is a simple mutual exclusion algorithm with no deadlock, which uses one test&set register.

MUTUAL EXCLUSION USING ONE TEST&SET REGISTER

Initially V equals 0

```
    ⟨Entry⟩:
1   wait until test&set(V) = 0
    ⟨Critical Section⟩
    ⟨Exit⟩:
2   reset(V)
    ⟨Remainder⟩
```

Assume that the initial value of V is 0. In the entry section, processor p_i repeatedly tests V until it returns 0. The last such test will assign 1 to V, causing any following test by other processors to return 1, prohibiting any other processor from entering the critical section. In the exit section p_i resets V to 0; another processor

waiting in the entry section can now enter the critical section.

Theorem 13.33 *The algorithm using one test &set register provides mutual exclusion without deadlock.*

13.8.4. Mutual exclusion using read/write registers

If a powerful primitive such as test&set is not available, then mutual exclusion must be implemented using only read/write operations.

The bakery algorithm

Lamport's bakery algorithm for mutual exclusion is an early, classical example of such an algorithm that uses only shared read/write registers. The algorithm guarantees mutual exclusion and no lockout for n processors using $O(n)$ registers (but the registers may need to store integer values that cannot be bounded ahead of time).

Processors wishing to enter the critical section behave like customers in a bakery. They all get a number and the one with the smallest number in hand is the next one to be "served". Any processor not standing in line has number 0, which is not counted as the smallest number.

The algorithm uses the following shared data structures: *Number* is an array of n integers, holding in its i-th entry the current number of processor p_i. *Choosing* is an array of n boolean values such that *Choosing*$[i]$ is true while p_i is in the process of obtaining its number. Any processor p_i that wants to enter the critical section attempts to choose a number greater than any number of any other processor and writes it into *Number*$[i]$. To do so, processors read the array *Number* and pick the greatest number read $+1$ as their own number. Since however several processors might be reading the array at the same time, symmetry is broken by choosing (*Number*$[i]$, i) as i's ticket. An ordering on tickets is defined using the lexicographical ordering on pairs. After choosing its ticket, p_i waits until its ticket is minimal: For all other p_j, p_i waits until p_j is not in the process of choosing a number and then compares their tickets. If p_j's ticket is smaller, p_i waits until p_j executes the critical section and leaves it.

BAKERY

Code for processor p_i, $0 \leq i \leq n-1$.
Initially *Number*$[i] = 0$ and
Choosing$[i]$ = FALSE, for $0 \leq i \leq n-1$

⟨Entry⟩:
1 *Choosing*$[i] \leftarrow$ TRUE
2 *Number*$[i] \leftarrow \max(Number[0], \ldots, Number[n-1]) + 1$
3 *Choosing*$[i] \leftarrow$ FALSE

```
4  for j ← 1 to n (≠ i) do
5      wait until Choosing[j] = FALSE
6      wait until Number[j] = 0 or (Number[j],j > (Number[i],i) ⟨Critical Section⟩
   ⟨Exit⟩:
7  Number[i] ← 0
   ⟨Remainder⟩
```

We leave the proofs of the following theorems as Exercises 13.8-2 and 13.8-3.

Theorem 13.34 BAKERY *guarantees mutual exclusion.*

Theorem 13.35 BAKERY *guarantees no lockout.*

A bounded mutual exclusion algorithm for n processors
Lamports BAKERY algorithm requires the use of unbounded values. We next present an algorithm that removes this requirement. In this algorithm, first presented by Peterson and Fischer, processors compete pairwise using a two-processor algorithm in a *tournament tree* arrangement. All pairwise competitions are arranged in a complete binary tree. Each processor is assigned to a specific leaf of the tree. At each level, the winner in a given node is allowed to proceed to the next higher level, where it will compete with the winner moving up from the other child of this node (if such a winner exists). The processor that finally wins the competition at the root node is allowed to enter the critical section.

Let $k = \lceil \log n \rceil - 1$. Consider a complete binary tree with 2^k leaves and a total of $2^{k+1} - 1$ nodes. The nodes of the tree are numbered inductively in the following manner: The root is numbered 1; the left child of node numbered m is numbered $2m$ and the right child is numbered $2m + 1$. Hence the leaves of the tree are numbered 2^k, $2^k + 1$,. . . ,$2^{k+1} - 1$.

With each node m, three binary shared variables are associated: $Want^m[0]$, $Want^m[1]$, and $Priority^m$. All variables have an initial value of 0. The algorithm is recursive. The code of the algorithm consists of a procedure NODE(m, *side*) which is executed when a processor accesses node m, while assuming the role of processor *side*. Each node has a critical section. It includes the entry section at all the nodes on the path from the nodes parent to the root, the original critical section and the exit code on all nodes from the root to the nodes parent. To begin, processor p_i executes the code of node $(2^k + \lfloor i/2 \rfloor, i \mod 2)$.

TOURNAMENT-TREE

```
   procedure NODE(m: integer; side: 0..1)
1  Want^m[side] ← 0
2  wait until (Want^m[1 − side] = 0 or Priority^m = side)
3  Want^m[side] ← 1
```

```
 4  if Priority^m = 1−side
 5     then if Want^m[1 − side] = 1)
 6             then goto line 1
 7             else  wait until Want^m[1 − side] = 0
 8  if v = 1
 9     then ⟨Critical Section⟩
10     else  NODE(⌊m/2⌋, m mod 2)
11     Priority^m = 1− side
12     Want^m[side] ← 0
   end procedure
```

This algorithm uses bounded values and as the next theorem shows, satisfies the mutual exclusion, no lockout properties:

Theorem 13.36 *The tournament tree algorithm guarantees mutual exclusion.*

Proof Consider any execution. We begin at the nodes closest to the leaves of the tree. A processor enters the critical section of this node if it reaches line 9 (it moves up to the next node). Assume we are at a node m that connects to the leaves where p_i and p_j start. Assume that two processors are in the critical section at some point. It follows from the code that then $Want^m[0] = Want^m[1] = 1$ at this point. Assume, without loss of generality that p_i's last write to $Want^m[0]$ before entering the critical section follows p_j's last write to $Want^m[1]$ before entering the critical section. Note that p_i can enter the critical section (of m) either through line 5 or line 6. In both cases p_i reads $Want^m[1] = 0$. However p_i's read of $Want^m[1]$, follows p_j's write to $Want^m[0]$, which by assumption follows p_j's write to $Want^m[1]$. Hence p_i's read of $Want^m[1]$ should return 1, a contradiction.

The claim follows by induction on the levels of the tree. ∎

Theorem 13.37 *The tournament tree algorithm guarantees no lockout.*

Proof Consider any admissible execution. Assume that some processor p_i is starved. Hence from some point on p_i is forever in the entry section. We now show that p_i cannot be stuck forever in the entry section of a node m. The claim then follows by induction.

Case 1: Suppose p_j executes line 10 setting $Priority^m$ to 0. Then $Priority^m$ equals 0 forever after. Thus p_i passes the test in line 2 and skips line 5. Hence p_i must be waiting in line 6, waiting for $Want^m[1]$ to be 0, which never occurs. Thus p_j is always executing between lines 3 and 11. But since p_j does not stay in the critical section forever, this would mean that p_j is stuck in the entry section forever which is impossible since p_j will execute line 5 and reset $Want^m[1]$ to 0.

Case 2: Suppose p_j never executes line 10 at some later point. Hence p_j must be waiting in line 6 or be in the remainder section. If it is in the entry section, p_j passes the test in line 2 ($Priority^m$ is 1). Hence p_i does not reach line 6. Therefore p_i waits in line 2 with $Want^m[0] = 0$. Hence p_j passes the test in line 6. So p_j cannot be forever in the entry section. If p_j is forever in the remainder section $Want^m[1]$

equals 0 henceforth. So p_i cannot be stuck at line 2, 5 or 6, a contradiction.

The claim follows by induction on the levels of the tree. ■

Lower bound on the number of read/write registers

So far, all deadlock-free mutual exclusion algorithms presented require the use of at least n shared variables, where n is the number of processors. Since it was possible to develop an algorithm that uses only bounded values, the question arises whether there is a way of reducing the number of shared variables used. Burns and Lynch first showed that any deadlock-free mutual exclusion algorithm using only shared read/write registers must use at least n shared variables, regardless of their size. The proof of this theorem allows the variables to be multi-writer variables. This means that each processor is allowed to write to each variable. Note that if the variables are single writer, that the theorem is obvious since each processor needs to write something to a (separate) variable before entering the critical section. Otherwise a processor could enter the critical section without any other processor knowing, allowing another processor to enter the critical section concurrently, a contradiction to the mutual exclusion property.

The proof by Burns and Lynch introduces a new proof technique, a ***covering argument***: Given any no deadlock mutual exclusion algorithm A, it shows that there is some reachable configuration of A in which each of the n processors is about to write to a ***distinct*** shared variable. This is called a ***covering*** of the shared variables. The existence of such a configuration can be shown using induction and it exploits the fact that any processor before entering the critical section, must write to at least one shared variable. The proof constructs a covering of all shared variables. A processor then enters the critical section. Immediately thereafter the covering writes are released so that no processor can detect the processor in the critical section. Another processor now concurrently enters the critical section, a contradiction.

Theorem 13.38 *Any no deadlock mutual exclusion algorithm using only read/write registers must use at least n shared variables.*

13.8.5. Lamport's fast mutual exclusion algorithm

In all mutual exclusion algorithms presented so far, the number of steps taken by processors before entering the critical section depends on n, the number of processors even in the absence of contention (where multiple processors attempt to concurrently enter the critical section), when a single processor is the only processor in the entry section. In most real systems however, the expected contention is usually much smaller than n.

A mutual exclusion algorithm is said to be ***fast*** if a processor enters the critical section within a constant number of steps when it is the only processor trying to enter the critical section. Note that a fast algorithm requires the use of multi-writer, multi-reader shared variables. If only single writer variables are used, a processor would have to read at least n variables.

Such a fast mutual exclusion algorithm is presented by Lamport.

FAST-MUTUAL-EXCLUSION

Code for processor p_i, $0 \leq i \leq n-1$. Initially *Fast-Lock* and *Slow-Lock* are 0, and *Want*[i] is false for all i, $0 \leq i \leq n-1$

```
    ⟨ Entry ⟩:
 1  Want[i] ← TRUE
 2  Fast-Lock ← i
 3  if Slow-Lock ≠ 0
 4     then Want[i] ← FALSE
 5     WAIT UNTIL Slow-Lock =0
 6     goto 1
 7  Slow-Lock ← i
 8  if Fast-Lock ≠ i
 9     then Want[i] ← FALSE
10          for all j, wait until Want[j] = FALSE
11          if Slow-Lock ≠ i
12             then wait until Slow-Lock = 0
13          goto 1
    ⟨Critical Section⟩
    ⟨Exit⟩:
14  Slow-Lock ← 0
15  Want[i] ← false
    ⟨Remainder⟩
```

Lamport's algorithm is based on the correct combination of two mechanisms, one for allowing fast entry when no contention is detected, and the other for providing deadlock freedom in the case of contention. Two variables, *Fast-Lock* and *Slow-Lock* are used for controlling access when there is no contention. In addition, each processor p_i has a boolean variable *Want*[i] whose value is true if p_i is interested in entering the critical section and false otherwise. A processor can enter the critical section by either finding *Fast-Lock* $= i$ - in this case it enters the critical section on the *fast path* - or by finding *Slow-Lock* $= i$ in which case it enters the critical section along the *slow path*.

Consider the case where no processor is in the critical section or in the entry section. In this case, *Slow-Lock* is 0 and all *Want* entries are 0. Once p_i now enters the entry section, it sets *Want*[i] to 1 and *Fast-Lock* to i. Then it checks *Slow-Lock* which is 0. then it checks *Fast-Lock* again and since no other processor is in the entry section it reads i and enters the critical section along the fast path with three writes and two reads.

If *Fast-Lock* $\neq i$ then p_i waits until all *Want* flags are reset. After some processor executes the for loop in line 10, the value of *Slow-Lock* remains unchanged until some processor leaving the critical section resets it. Hence at most one processor p_j may find *Slow-Lock*$= j$ and this processor enters the critical section along the slow path. Note that the Lamport's Fast Mutual Exclusion algorithm does not guarantee

lockout freedom.

Theorem 13.39 *Algorithm* FAST-MUTUAL-EXCLUSION *guarantees mutual exclusion without deadlock.*

Exercises

13.8-1 An algorithm solves the *2-mutual exclusion* problem if at any time at most two processors are in the critical section. Present an algorithm for solving the 2-mutual exclusion problem using test & set registers.

13.8-2 Prove that bakery algorithm satisfies the mutual exclusion property.

13.8-3 Prove that bakery algorithm provides no lockout.

13.8-4 Isolate a bounded mutual exclusion algorithm with no lockout for two processors from the tournament tree algorithm. Show that your algorithm has the mutual exclusion property. Show that it has the no lockout property.

13.8-5 Prove that algorithm FAST-MUTUAL-EXCLUSION has the mutual exclusion property.

13.8-6 Prove that algorithm FAST-MUTUAL-EXCLUSION has the no deadlock property.

13.8-7 Show that algorithm FAST-MUTUAL-EXCLUSION does not satisfy the no lockout property, i.e. construct an execution in which a processor is locked out of the critical section.

13.8-8 Construct an execution of algorithm FAST-MUTUAL-EXCLUSION in which two processors are in the entry section and both read at least $\Omega(n)$ variables before entering the critical section.

Problems

***13-1 Number of messages of the algorithm* FLOOD**

Prove that the algorithm FLOOD sends $O(e)$ messages in any execution, given a graph G with n vertices and e edges. What is the exact number of messages as a function of the number of vertices and edges in the graph?

13-2 Leader election in a ring

Assume that messages can only be sent in CW direction, and design an asynchronous algorithm for leader election on a ring that has $O(n \lg n)$ message complexity. *Hint.* Let processors work in phases. Each processor begins in the ***active mode*** with a ***value*** equal to the identifier of the processor, and under certain conditions can enter the ***relay mode***, where it just relays messages. An active processor waits for messages from two active processors, and then inspects the values sent by the processors, and decides whether to become the leader, remain active and adopt one of the values, or start relaying. Determine how the decisions should be made so as to ensure that if there are three or more active processors, then at least one will remain active; and no matter what values active processors have in a phase, at most half of them will still be active in the next phase.

13-3 Validity condition in asynchronous systems
Show that the validity condition is equivalent to requiring that every nonfaulty processor decision be the input of some processor.

13-4 Single source consensus
An alternative version of the consensus problem requires that the input value of one distinguished processor (the *general*) be distributed to all the other processors (the *lieutenants*). This problem is also called ***single source consensus problem***. The conditions that need to be satisfied are:

- **Termination**: Every nonfaulty lieutenant must eventually decide,
- **Agreement**: All the nonfaulty lieutenants must have the same decision,
- **Validity**: If the general is nonfaulty, then the common decision value is the general's input.

So if the general is faulty, then the nonfaulty processors need not decide on the general's input, but they must still agree with each other. Consider the synchronous message passing system with Byzantine faults. Show how to transform a solution to the consensus problem (in Subsection 13.4.5) into a solution to the general's problem and vice versa. What are the message and round overheads of your transformation?

13-5 Bank transactions
Imagine that there are n banks that are interconnected. Each bank i starts with an amount of money m_i. Banks do not remember the initial amount of money. Banks keep on transferring money among themselves by sending messages of type $<10>$ that represent the value of a transfer. At some point of time a bank decides to find the total amount of money in the system. Design an algorithm for calculating $m_1 + \cdots + m_n$ that does not stop monetary transactions.

Chapter notes

The definition of the distributed systems presented in the chapter are derived from the book by Attiya and Welch [24]. The model of distributed computation, for message passing systems without failures, was proposed by Attiya, Dwork, Lynch and Stockmeyer [23].

Modeling the processors in the distributed systems in terms of automata follows the paper of Lynch and Fisher [228].

The concept of the execution sequences is based on the papers of Fischer, Gries, Lamport and Owicki [228, 260, 261].

The definition of the asynchronous systems reflects the presentation in the papers of Awerbuch [25], and Peterson and Fischer [269].

The algorithm Spanning-Tree-Broadcast is presented after the paper due to Segall [296].

The leader election algorithm Bully was proposed by Hector Garcia-Molina in 1982 [127]. The asymptotic optimality of this algorithm was proved by Burns [52].

The *two generals problem* is presented as in the book of Gray [143].

The *consensus problem* was first studied by Lamport, Pease, and Shostak [211, 267]. They proved that the Byzantine consensus problem is unsolvable if $n \leq 3f$ [267].

One of the basic results in the theory of asynchronous systems is that the consensus problem is not solvable even if we have reliable communication systems, and one single faulty processor which fails by crashing. This result was first shown in a breakthrough paper by Fischer, Lynch and Paterson [108].

The algorithm CONSENSUS-WITH-CRASH-FAILURES is based on the paper of Dolev and Strong [90].

Berman and Garay [41] proposed an algorithm for the solution of the Byzantine consensus problem for the case $n > 4f$. Their algorithm needs $2(f + 1)$ rounds.

The bakery algorithm [209] for mutual exclusion using only shared read/write registers to solve mutual exclusion is due to Lamport [209]. This algorithm requires arbitrary large values. This requirement is removed by Peterson and Fischer [269]. After this Burns and Lynch proved that any deadlock-free mutual exclusion algorithm using only shared read/write registers must use at least n shared variables, regardless of their size [53].

The algorithm FAST-MUTUAL-EXCLUSION is presented by Lamport [210].

The source of the problems *13-3*, *13-4*, *13-5* is the book of Attiya and Welch [24].

Important textbooks on distributed algorithms include the monumental volume by Nancy Lynch [227] published in 1997, the book published by Gerard Tel [320] in 2000, and the book by Attiya and Welch [24]. Also of interest is the monograph by Claudia Leopold [220] published in 2001, and the book by Nicola Santoro [295], which appeared in 2006.

Finally, several important open problems in distributed computing can be found in a recent paper of Aspnes et al. [21].

14. Network Simulation

In this chapter we discuss methods and techniques to simulate the operations of computer network systems and network applications in real-world environment. Simulation is one of the most widely used techniques in network design and management to predict the performance of a network system or network application before the network is physically built or the application is rolled out.

14.1. Types of simulation

A network system is a set of network elements, such as routers, switches, links, users, and applications working together to achieve some tasks. The scope of a simulation study may only be a system that is part of another system as in the case of subnetworks. The state of a network system is the set of relevant variables and parameters that describe the system at a certain time that comprise the scope of the study. For instance, if we are interested in the utilisation of a link, we want to know only the number of bits transmitted via the link in a second and the total capacity of the link, rather than the amount of buffers available for the ports in the switches connected by the link.

Instead of building a physical model of a network, we build a mathematical model representing the behaviour and the logical and quantitative relations between network elements. By changing the relations between network elements, we can analyse the model without constructing the network physically, assuming that the model behaves similarly to the real system, i.e., it is a valid model. For instance, we can calculate the utilisation of a link analytically, using the formula $U = D/T$, where D is the amount of data sent at a certain time and T is the capacity of the link in bits per second. This is a very simple model that is very rare in real world problems. Unfortunately, the majority of real world problems are too complex to answer questions using simple mathematical equations. In highly complex cases simulation technique is more appropriate.

Simulation models can be classified in many ways. The most common classifications are as follows:

- *Static and dynamic simulation models:* A static model characterises a system independently of time. A dynamic model represents a system that changes over time.

- *Stochastic and deterministic models:* If a model represents a system that includes random elements, it is called a stochastic model. Otherwise it is deterministic. Queueing systems, the underlying systems in network models, contain random components, such as arrival time of packets in a queue, service time of packet queues, output of a switch port, etc.

- *Discrete and continuous models:* A continuous model represents a system with state variables changing continuously over time. Examples are differential equations that define the relationships for the extent of change of some state variables according to the change of time. A discrete model characterises a system where the state variables change instantaneously at discrete points in time. At these discrete points some event or events may occur, changing the state of the system. For instance, the arrival of a packet at a router at a certain time is an event that changes the state of the port buffer in the router.

In our discussion, we assume dynamic, stochastic, and discrete network models. We refer to these models as discrete-event simulation models.

Due to the complex nature of computer communications, network models tend to be complex as well. The development of special computer programs for a certain simulation problem is a possibility, but it may be very time consuming and inefficient. Recently, the application of simulation and modelling packages has become more customary, saving coding time and allowing the modeller to concentrate on the modelling problem in hand instead of the programming details. At first glance, the use of such network simulation and modelling packages, as COMNET, OPNET, etc., creates the risk that the modeller has to rely on modelling techniques and hidden procedures that may be proprietary and may not be available to the public. In the following sections we will discuss the simulation methodology on how to overcome the fear of this risk by using validation procedures to make sure that the real network system will perform the same way as it has been predicted by the simulation model.

14.2. The need for communications network modelling and simulation

In a world of more and more data, computers, storage systems, and networks, the design and management of systems are becoming an increasingly challenging task. As networks become faster, larger, and more complex, traditional static calculations are no longer reasonable approaches for validating the implementation of a new network design and multimillion dollar investments in new network technologies. Complex static calculations and spreadsheets are not appropriate tools any more due to the stochastic nature of network traffic and the complexity of the overall system.

Organisations depend more and more on new network technologies and network applications to support their critical business needs. As a result, poor network performance may have serious impacts on the successful operation of their businesses. In order to evaluate the various alternative solutions for a certain design goal, network designers increasingly rely on methods that help them evaluate several design proposals before the final decision is made and the actual systems is built. A wi-

dely accepted method is performance prediction through simulation. A simulation model can be used by a network designer to analyse design alternatives and study the behaviour of a new system or the modifications to an existing system without physically building it. A simulation model can also represent the network topology and tasks performed in a network in order to obtain statistical results about the network's performance.

It is important to understand the difference between simulation and emulation. The purpose of emulation is to mimic the original network and reproduce every event that happens in every network element and application. In simulation, the goal is to generate statistical results that represent the behaviour of certain network elements and their functions. In discrete event simulation, we want to observe events as they happen over time, and collect performance measures to draw conclusions on the performance of the network, such as link utilisation, response times, routers' buffer sizes, etc.

Simulation of large networks with many network elements can result in a large model that is difficult to analyse due to the large amount of statistics generated during simulation. Therefore, it is recommended to model only those parts of the network which are significant regarding the statistics we are going to obtain from the simulation. It is crucial to incorporate only those details that are significant for the objectives of the simulation. Network designers typically set the following objectives:

- *Performance modelling:* Obtain statistics for various performance parameters of links, routers, switches, buffers, response time, etc.
- *Failure analysis:* Analyse the impacts of network element failures.
- *Network design:* Compare statistics about alternative network designs to evaluate the requirements of alternative design proposals.
- *Network resource planning:* Measure the impact of changes on the network's performance, such as addition of new users, new applications, or new network elements.

Depending on the objectives, the same network might need different simulation models. For instance, if the modeller wants to determine the overhead of a new service of a protocol on the communication links, the model's links need to represent only the traffic generated by the new service. In another case, when the modeller wants to analyse the response time of an application under maximum offered traffic load, the model can ignore the traffic corresponding to the new service of the protocol analysed in the previous model.

Another important question is the granularity of the model, i.e., the level of details at which a network element is modelled. For instance, we need to decide whether we want to model the internal architecture of a router or we want to model an entire packet switched network. In the former case, we need to specify the internal components of a router, the number and speed of processors, types of buses, number of ports, amount of port buffers, and the interactions between the router's components. But if the objective is to analyse the application level end-to-end response time in the entire packet switched network, we would specify the types of applications and protocols, the topology of the network and link capacities, rather then the internal details of the routers. Although the low level operations of the routers

affect the overall end-to-end response time, modelling the detailed operations do not significantly contribute to the simulation results when looking at an entire network. Modelling the details of the routers' internal operations in the order of magnitude of nanoseconds does not contribute significantly to the end-to-end delay analysis in the higher order of magnitude of microseconds or seconds. The additional accuracy gained from higher model granularity is far outweighed by the model's complexity and the time and effort required by the inclusion of the routers' details.

Simplification can also be made by applying statistical functions. For instance, modelling cell errors in an ATM network does not have to be explicitly modelled by a communication link by changing a bit in the cell's header, generating a wrong CRC at the receiver. Rather, a statistical function can be used to decide when a cell has been damaged or lost. The details of a cell do not have to be specified in order to model cell errors.

These examples demonstrate that the goal of network simulation is to reproduce the functionality of a network pertinent to a certain analysis, not to emulate it.

14.3. Types of communications networks, modelling constructs

A communications network consists of network elements, nodes (senders and receivers) and connecting communications media. Among several criteria for classifying networks we use two: transmission technology and scale. The scale or distance also determines the technique used in a network: wireline or wireless. The connection of two or more networks is called ***internetwork.*** The most widely known internetwork is the Internet.

According to transmission technology we can broadly classify networks as broadcast and point-to-point networks:

- In *broadcast networks* a single communication channel is shared by every node. Nodes communicate by sending packets or frames received by all the other nodes. The address field of the frame specifies the recipient or recipients of the frame. Only the addressed recipient(s) will process the frame. Broadcast technologies also allow the addressing of a frame to all nodes by dedicating it as a broadcast frame processed by every node in the network. It is also possible to address a frame to be sent to all or any members of only a group of nodes. The operations are called multicasting and any casting, respectively.
- *Point-to-point networks* consist of many connections between pairs of nodes. A packet or frame sent from a source to a destination may have to first traverse intermediate nodes where they are stored and forwarded until it reaches the final destination.

Regarding our other classification criterion, the scale of the network, we can classify networks by their physical area coverage:

- *Personal Area Networks* (PANs) support a person's needs. For instance, a wireless network of a keyboard, a mouse, and a *personal digital assistant (PDA)* can be considered as a PAN.

- *Local area networks* (LANs), typically owned by a person, department, a smaller organisation at home, on a single floor or in a building, cover a limited geographic area. LANs connect workstations, servers, and shared resources. LANs can be further classified based on the transmission technology, speed measured in bits per second, and topology. Transmissions technologies range from traditional 10 Mbps LANs to today's 10 Gbps LANs. In terms of topology, there are bus and ring networks and switched LANs.
- *Metropolitan area networks* (MANs) span a larger area, such as a city or a suburb. A widely deployed MAN is the cable television network distributing not just one-way TV programs but two-way Internet services as well in the unused portion of the transmission spectrum. Other MAN technologies are the *Fiber Distributed Data Interface* (FDDI) and IEEE wireless technologies as discussed below.
- *Wide area networks* (WANs) cover a large geographical area, a state, a country or even a continent. A WAN consists of hosts (clients and servers) connected by subnets owned by communications service providers. The subnets deliver messages from the source host to the destination host. A subnet may contain several transmission lines, each one connecting a pair of specialised hardware devices called routers. Transmission lines are made of various media; copper wire, optical fiber, wireless links, etc. When a message is to be sent to a destination host or hosts, the sending host divides the message into smaller chunks, called ***packets.*** When a packet arrives on an incoming transmission line, the router stores the packet before it selects an outgoing line and forwards the packet via that line. The selection of the outgoing line is based on a routing algorithm. The packets are delivered to the destination host(s) one-by-one where the packets are reassembled into the original message.

Wireless networks can be categorised as short-range radio networks, wireless LANs, and wireless WANs.

- In short range radio networks, for instance Bluetooth, various components, digital cameras, ***Global Positioning System*** (GPS) devices, headsets, computers, scanners, monitors, and keyboards are connected via short-range radio connections within 20–30 feet. The components are in primary-secondary relation. The main system unit, the primary component, controls the operations of the secondary components. The primary component determines what addresses the secondary devices use, when and on what frequencies they can transmit.
- A wireless LAN consists of computers and access points equipped with a radio modem and an antenna for sending and receiving. Computers communicate with each other directly in a peer-to-peer configuration or via the access point that connects the computers to other networks. Typical coverage area is around 300 feet. The wireless LAN protocols are specified under the family of IEEE 802.11 standards for a range of speed from 11 Mbps to 108 Mbps.
- Wireless WANs comprise of low bandwidth and high bandwidth networks. The low bandwidth radio networks used for cellular telephones have evolved through three generations. The first generation was designed only for voice communications utilising analog signalling. The second generation also transmitted only

voice but based on digital transmission technology. The current third generation is digital and transmits both voice and data at most 2Mbps. Fourth and further generation cellular systems are under development. High-bandwidth WANs provides high-speed access from homes and businesses bypassing the telephone systems. The emerging IEEE 802.16 standard delivers services to buildings, not mobile stations, as the IEEE 802.11 standards, and operates in much higher 10-66 GHz frequency range. The distance between buildings can be several miles.

- Wired or wireless home networking is getting more and more popular connecting various devices together that can be accessible via the Internet. Home networks may consists of PCs, laptops, PDAs, TVs, DVDs, camcorders, MP3 players, microwaves, refrigerator, A/C, lights, alarms, utility meters, etc. Many homes are already equipped with high-speed Internet access (cable modem, DSL, etc.) through which people can download music and movies on demand.

The various components and types of communications networks correspond to the modelling constructs and the different steps of building a simulation model. Typically, a network topology is built first, followed by adding traffic sources, destinations, workload, and setting the parameters for network operation. The simulation control parameters determine the experiment and the running of the simulation. Prior to starting a simulation various statistics reports can be activated for analysis during or after the simulation. Statistical distributions are available to represent specific parameterisations of built-in analytic distributions. As the model is developed, the modeller creates new model libraries that can be reused in other models as well.

14.4. Performance targets for simulation purposes

In this section we discuss a non-exhausting list of ***network attributes*** that have a profound effect on the perceived network performance and are usual targets of network modelling. These attributes are the goals of the statistical analysis, design, and optimisation of computer networks. Fundamentally, network models are constructed by defining the statistical distribution of the arrival and service rate in a queueing system that subsequently determines these attributes.

- *Link capacity*
  Channel or link capacity is the number of messages per unit time handled by a link. It is usually measured in bits per second. One of the most famous of all results of information theory is Shannon's channel coding theorem: "For a given channel there exists a code that will permit the error-free transmission across the channel at a rate R, provided $R \leq C$, where C is the channel capacity." Equality is achieved only when the Signal-to-noise Ratio (SNR) is infinite. See more details in textbooks on information and coding theory.
- *Bandwidth*
  Bandwidth is the difference between the highest and lowest frequencies available for network signals. Bandwidth is also a loose term used to describe the throughput capacity of a specific link or protocol measured in Kilobits, Megabits, Gigabits, Terabits, etc., in a second.

- *Response time*
  The ***response time*** is the time it takes a network system to react to a certain source's input. The response time includes the transmission time to the destination, the processing time at both the source and destination and at the intermediate network elements along the path, and the transmission time back to the source. Average response time is an important measure of network performance. For users, the lower the response time the better. Response time statistics (mean and variation) should be stationary; it should not dependent on the time of the day. Note that low average response time does not guarantee that there are no extremely long response times due to network congestions.

- *Latency*
  Delay or latency is the amount of time it takes for a unit of data to be transmitted across a network link. Latency and bandwidth are the two factors that determine the speed of a link. It includes the propagation delay (the time taken for the electrical or optical signals to travel the distance between two points) and processing time. For instance, the latency, or round-time delay between a ground station of a satellite communication link and back to another ground station (over 34,000 km each way) is approximately 270 milliseconds. The round-time delay between the east and west coast of the US is around 100 ms, and transglobal is about 125 ms. The end-to-end delay of a data path between source and destination spanning multiple segments is affected not only by the media' signal speed, but also by the network devices, routers, switches along the route that buffer, process, route, switch, and encapsulate the data payload. Erroneous packets and cells, signal loss, accidental device and link failures and overloads can also contribute to the overall network delay. Bad cells and packets force retransmission from the initial source. These packets are typically dropped with the expectation of a later retransmission resulting in slowdowns that cause packets to overflow buffers.

- *Routing protocols*
  The route is the path that network traffic takes from the source to the destination. The path in a LAN is not a critical issue because there is only one path from any source to any destination. When the network connects several enterprises and consists of several paths, routers, and links, finding the best route or routes becomes critical. A route may traverse through multiple links with different capacities, latencies, and reliabilities. Routes are established by routing protocols. The objective of the routing protocols is to find an optimal or near optimal route between source and destination avoiding congestions.

- *Traffic engineering*
  A new breed of routing techniques is being developed using the concept of traffic engineering. Traffic engineering implies the use of mechanisms to avoid congestion by allocating network resources optimally, rather than continually increasing network capacities. Traffic engineering is accomplished by mapping traffic flows to the physical network topology along predetermined paths. The optimal allocation of the forwarding capacities of routers and switches are the main target of traffic engineering. It provides the ability to diverge traffic flows away from the

optimal path calculated by the traditional routing protocols into a less congested area of the network. The purpose of traffic engineering is to balance the offered load on the links, routers, and switches in a way that none of these network elements is over or under utilised.

- *Protocol overhead*
  Protocol messages and application data are embedded inside the protocol data units, such as frames, packets, and cells. A main interest of network designers is the overhead of protocols. Protocol overhead concerns the question: How fast can we really transmit using a given communication path and protocol stack, i.e., how much bandwidth is left for applications? Most protocols also introduce additional overhead associated with in-band protocol management functions. Keep-alive packets, network alerts, control and monitoring messages, poll, select, and various signalling messages are transmitted along with the data streams.
- *Burstiness*
  The most dangerous cause of network congestion is the ***burstiness*** of the network traffic. Recent results make evident that high-speed Internet traffic is more bursty and its variability cannot be predicted as assumed previously. It has been shown that network traffic has similar statistical properties on many time scales. Traffic that is bursty on many or all time scales can be described statistically using the notion of *long-range dependency*. Long-range dependent traffic has observable bursts on all time scales. One of the consequences is that combining the various flows of data, as it happens in the Internet, does not result in the smoothing of traffic. Measurements of local and wide area network traffic have proven that the widely used Markovian process models cannot be applied for today's network traffic. If the traffic were Markovian process, the traffic's burst length would be smoothed by averaging over a long time scale, contradicting the observations of today's traffic characteristics. The harmful consequences of bursty traffic will be analysed in a case study in Section 14.9.
- *Frame size*
  Network designers are usually worried about large frames because they can fill up routers' buffers much faster than smaller frames resulting in lost frames and retransmissions. Although the processing delay for larger frames is the same as for smaller ones, i.e., larger packets are seemingly more efficient, routers and switches can process internal queues with smaller packets faster. Larger frames are also target for fragmentation by dividing them into smaller units to fit in the **M**aximum **T**ransmission **U**nit (MTU). MTU is a parameter that determines the largest datagram than can be transmitted by an IP interface. On the other hand, smaller frames may create more collision in an Ethernet network or have lower utilisation on a WAN link.
- *Dropped packet rate*
  Packets may be dropped by the data link and network layers of the OSI architecture. The transport layer maintains buffers for unacknowledged packets and retransmits them to establish an error-free connection between sender and receiver. The rate of dropping packets at the lower layers determines the rate of retransmitting packets at the transport layer. Routers and switches may also

drop packets due to the lack of internal buffers. Buffers fill up quicker when WAN links get congested which causes timeouts and retransmissions at the transport layer. The TCP's *slow start algorithm* tries to avoid congestions by continually estimating the round-trip propagation time and adjusting the transmission rate according to the measured variations in the roundtrip time.

14.5. Traffic characterisation

Communications networks transmit data with random properties. Measurements of network attributes are statistical samples taken from random processes, for instance, response time, link utilisation, interarrival time of messages, etc. In this section we review basic statistics that are important in network modelling and performance prediction. After a family of statistical distributions has been selected that corresponds to a network attribute under analysis, the next step is to estimate the parameters of the distribution. In many cases the sample average or mean and the sample variance are used to estimate the parameters of a hypothesised distribution. Advanced software tools include the computations for these estimates. The mean is interpreted as the most likely value about which the samples cluster. The following equations can be used when discrete or continues raw data available. Let $X_1, X_2, \ldots, X_n$ are samples of size n. The mean of the sample is defined by

$$\overline{X} = \frac{\sum_{i=1}^{n} X_i}{n} \ .$$

The sample variance S^2 is defined by

$$S^2 = \frac{\sum_{i=1}^{n} {X_i}^2 - n\overline{X}^2}{n-1} \ .$$

If the data are discrete and grouped in a frequency distribution, the equations above are modified as

$$\overline{X} = \frac{\sum\limits_{j=1}^{k} f_j X_j}{n} \ ,$$

$$S^2 = \frac{\sum\limits_{j=1}^{k} f_j {X_j}^2 - n\overline{X}^2}{n-1} \ ,$$

where k is the number of different values of X and fj is the frequency of the value Xj of X. The standard deviation S is the square root of the variance S^2.

The variance and standard deviation show the deviation of the samples around the mean value. Small deviation from the mean demonstrates a strong central tendency of the samples. Large deviation reveals little central tendency and shows large statistical randomness.

Numerical estimates of the distribution parameters are required to reduce the family of distributions to a single distribution and test the corresponding hypothesis.

distribution	parameter(s)	estimator(s)
Poisson	α	$\hat{\alpha} = \overline{X}$
exponential	λ	$\hat{\lambda} = 1/\overline{X}$
uniform	b	$\hat{b} = ((n+1)/n)[\max(X)]$ (unbiased)
normal	μ, σ^2	$\hat{\mu} = \overline{X}$ $\hat{\sigma}^2 = S^2$ (unbiased)

Figure 14.1. Estimation of the parameters of the most common distributions.

Figure 14.1 describes estimators for the most common distributions occurring in network modelling. If α denotes a parameter, the estimator is denoted by $\hat{\alpha}$. Except for an adjustment to remove bias in the estimates of σ^2 for the normal distribution and in the estimate of b of the uniform distribution, these estimators are the maximum likelihood estimators based on the sample data.

Probability distributions describe the random variations that occur in the real world. Although we call the variations random, randomness has different degrees; the different distributions correspond to how the variations occur. Therefore, different distributions are used for different simulation purposes. Probability distributions are represented by probability density functions. Probability density functions show how likely a certain value is. Cumulative density functions give the probability of selecting a number at or below a certain value. For example, if the cumulative density function value at 1 was equal to 0.85, then 85% of the time, selecting from this distribution would give a number less than 1. The value of a cumulative density function at a point is the area under the corresponding probability density curve to the left of that value. Since the total area under the probability density function curve is equal to one, cumulative density functions converge to one as we move toward the positive direction. In most of the modelling cases, the modeller does not need to know all details to build a simulation model successfully. He or she has only to know which distribution is the most appropriate one for the case.

Below, we summarise the most common statistical distributions. We use the simulation modelling tool COMNET to depict the respective probability density functions (PDF). From the practical point of view, a PDF can be approximated by a histogram with all the frequencies of occurrences converted into probabilities.

- *Normal distribution*
  It typically models the distribution of a compound process that can be described as the sum of a number of component processes. For instance, the time to transfer a file (response time) sent over the network is the sum of times required to send the individual blocks making up the file. In modelling tools the normal distribution function takes two positive, real numbers: mean and standard deviation. It returns a positive, real number. The stream parameter x specifies which random number stream will be used to provide the sample. It is also often used to model message sizes. For example, a message could be described with mean size of 20,000 bytes and a standard deviation of 5,000 bytes.

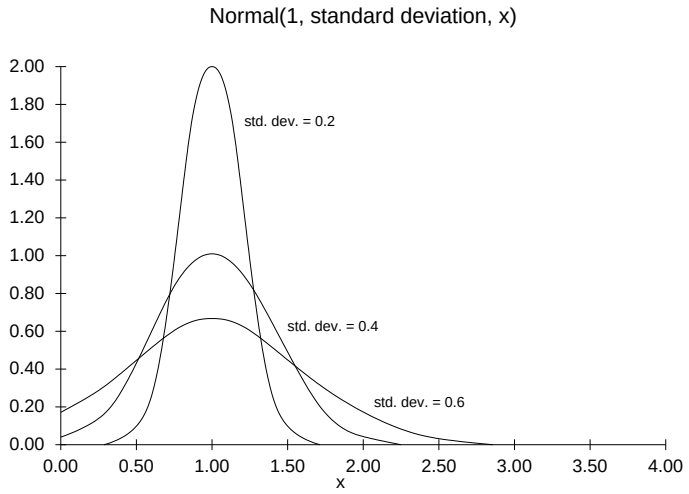


Figure 14.2. An example normal distribution.

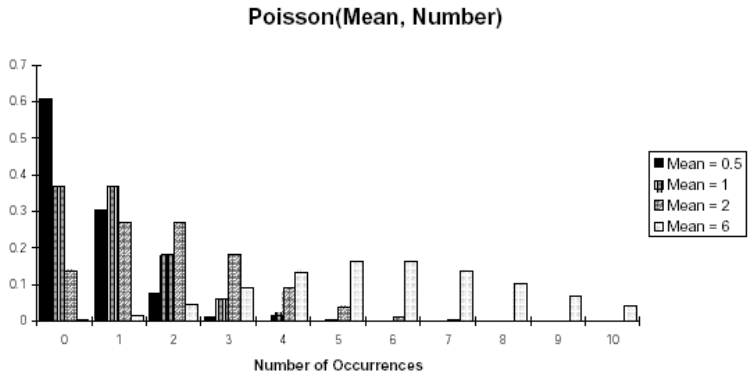


Figure 14.3. An example Poisson distribution.

- *Poisson distribution*
  It models the number of independent events occurring in a certain time interval; for instance, the number of packets of a packet flow received in a second or a minute by a destination. In modelling tools, the Poisson distribution function takes one positive, real number, the mean. The "number" parameter in Figure 14.3 specifies which random number stream will be used to provide the sample. This distribution, when provided with a time interval, returns an integer which is often used to represent the number of arrivals likely to occur in that time interval. Note that in simulation, it is more useful to have this information expressed as the time interval between successive arrivals. For this purpose, the exponential distribution is used.

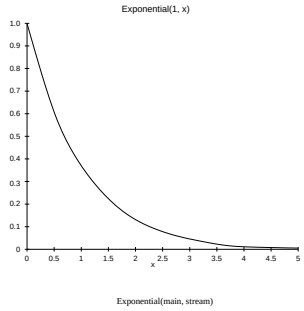


Figure 14.4. An example exponential distribution.

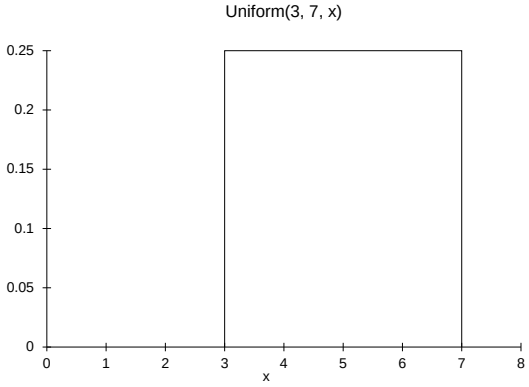


Figure 14.5. An example uniform distribution.

- *Exponential distribution*
  It models the time between independent events, such as the interarrival time between packets sent by the source of a packet flow. Note, that the number of events is Poisson, if the time between events is exponentially distributed. In modelling tools, the Exponential distribution function 14.4 takes one positive, real number, the mean and the stream parameter x that specifies which random number stream will be used to provide the sample. Other application areas include: Time between data base transactions, time between keystrokes, file access, emails, name lookup request, HTTP lookup, X-window protocol exchange, etc.
- *Uniform distribution*
  Uniform distribution models (see Figure 14.5) data that range over an interval of values, each of which is equally likely. The distribution is completely determined by the smallest possible value min and the largest possible value max. For discrete data, there is a related discrete uniform distribution as well. Packet lengths are often modelled by uniform distribution. In modelling tools the Uniform distribution function takes three positive, real numbers: min, max, and stream. The stream parameter x specifies which random number stream will be used to provide the sample.

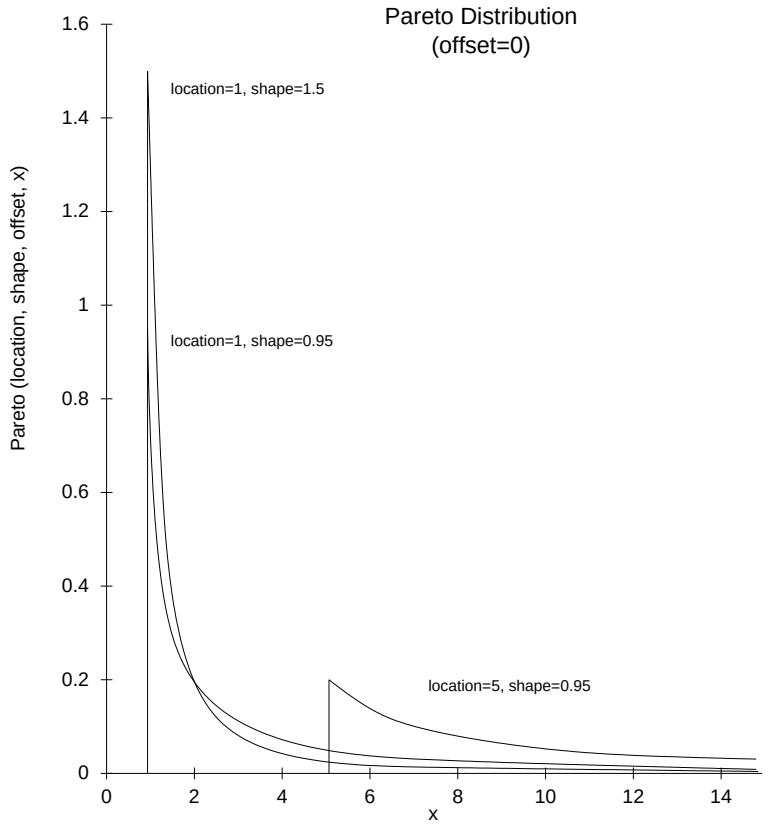


Figure 14.6. An example Pareto distribution.

- *Pareto distribution*
  The Pareto distribution (see 14.6) is a power-law type distribution for modelling bursty sources (not long-range dependent traffic). The distribution is heavily peaked but the tail falls off slowly. It takes three parameters: location, shape, and offset. The location specifies where the distribution starts, the shape specifies how quickly the tail falls off, and the offset shifts the distribution.

A common use of probability distribution functions is to define various network parameters. A typical network parameter for modelling purposes is the time between successive instances of messages when multiple messages are created. The specified time is from the start of one message to the start of the next message. As it is discussed above, the most frequent distribution to use for interarrival times is the exponential distribution (see Figure 14.7).

The parameters entered for the exponential distribution are the mean value and

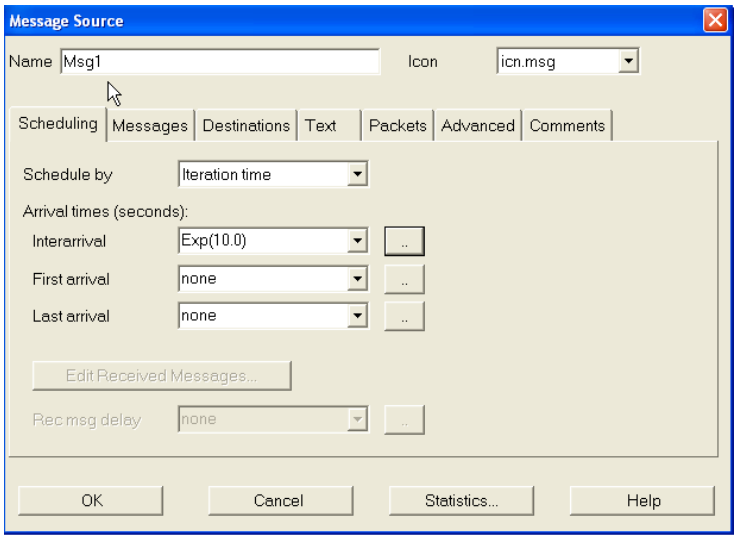


Figure 14.7. Exponential distribution of interarrival time with 10 sec on the average.

the random stream number to use. Network traffic is often described as a Poisson process. This generally means that the number of messages in successive time intervals has been observed and the distribution of the number of observations in an interval is Poisson distributed. In modelling tools, the number of messages per unit of time is not entered. Rather, the interarrival time between messages is required. It may be proven that if the number of messages per unit time interval is Poisson-distributed, then the interarrival time between successive messages is exponentially distributed. The interarrival distribution in the following dialog box for a message source in COMNET is defined by Exp (10.0). It means that the time from the start of one message to the start of the next message follows an exponential distribution with 10 seconds on the average. Figure 14.8 shows the corresponding probability density function.

Many simulation models focus on the simulation of various traffic flows. Traffic flows can be simulated by either specifying the traffic characteristics as input to the model or by importing actual traffic traces that were captured during certain application transactions under study. The latter will be discussed in a subsequent section on Baselining.

Network modellers usually start the modelling process by first analysing the captured traffic traces to visualise network attributes. It helps the modeller understand the application level processes deep enough to map the corresponding network events to modelling constructs. Common tools can be used before building the model. After the preliminary analysis, the modeller may disregard processes, events that are not important for the study in question. For instance, the capture of traffic traces of a database transaction reveals a large variation in frame lengths. Figure 14.9 helps visualise the anomalies:

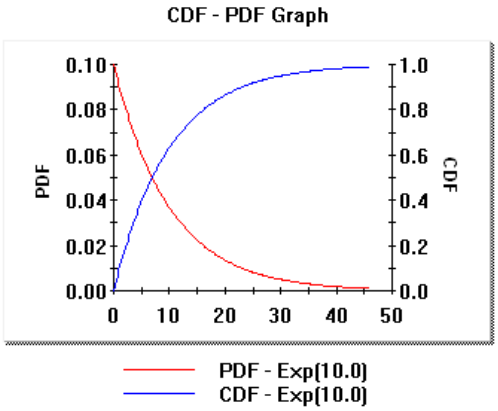


Figure 14.8. Probability density function of the Exp (10.0) interarrival time.

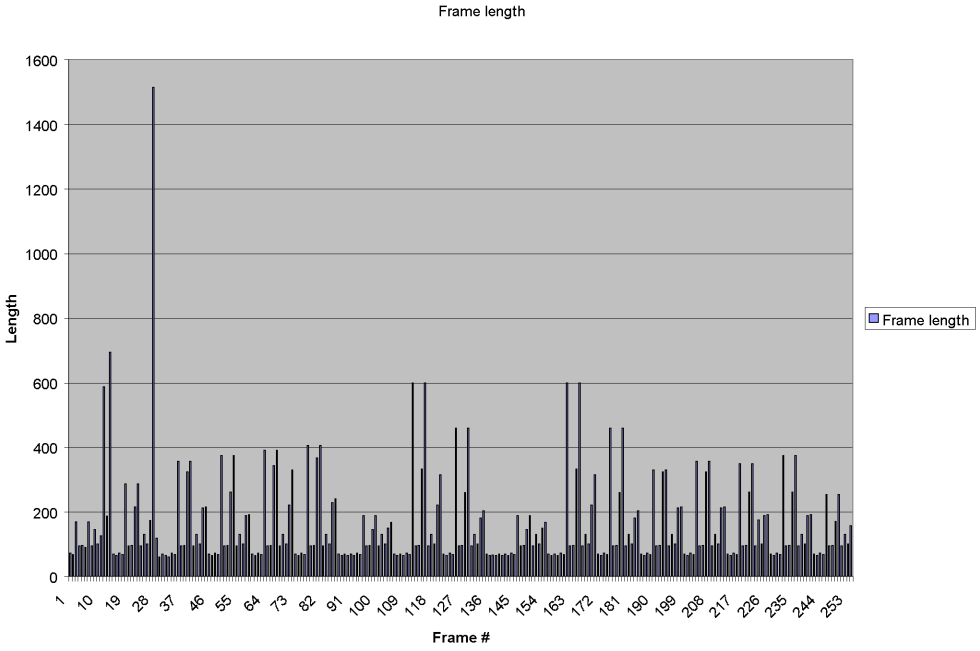


Figure 14.9. Visualisation of anomalies in packet lengths.

The analysis of the same trace (Figure 14.10) also discloses a large deviation of the interarrival times of the same frames (delta times):

Approximating the cumulative probability distribution function by a histogram

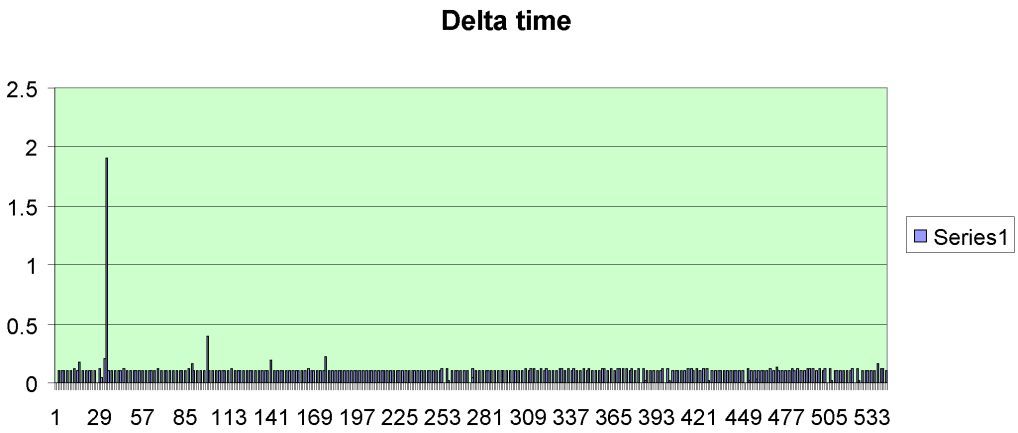


Figure 14.10. Large deviations between delta times.

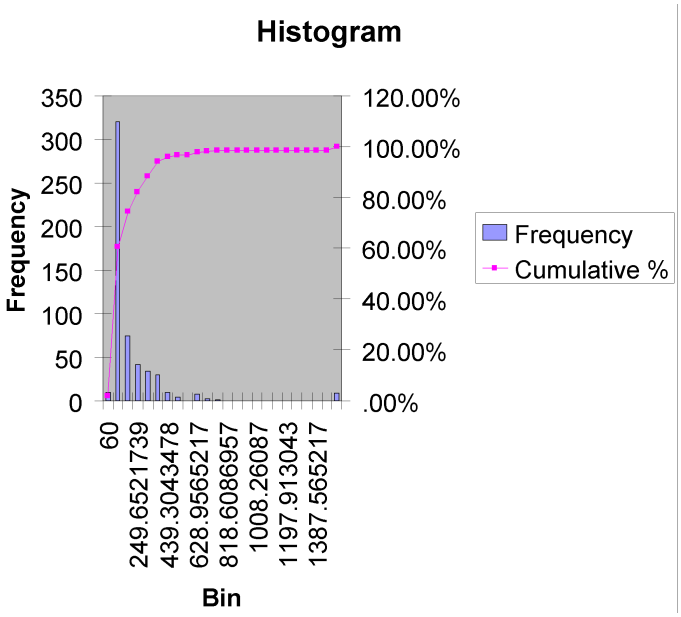


Figure 14.11. Histogram of frame lengths.

of the frame lengths of the captured traffic trace (Figure 14.11) helps the modeller determine the family of the distribution:

14.6. Simulation modelling systems

14.6.1. Data collection tools and network analysers

The section summaries the main features of the widely used discrete event simulation tools, OPNET and COMNET, and the supporting network analysers, Network Associates' Sniffer and OPNET's Application Characterisation Environment.

OPtimized Network Engineering Tools (OPNET) is a comprehensive simulation system capable of modelling communication networks and distributed systems with detailed protocol modelling and performance analysis. OPNET consists of a number of tools that fall into three categories corresponding to the three main phases of modelling and simulation projects: model specification, data collection and simulation, and analysis.

14.6.2. Model specification

During model specification the network modeller develops a representation of the network system under study. OPNET implements the concept of model reuse, i.e., models are based on embedded models developed earlier and stored in model libraries. The model is specified at various levels of details using specification editors. These editors categorise the required modelling information corresponding to the hierarchical structure of an actual network system. The highest level editor, the *Project Editor* develops network models consisting of network topology, subnets, links, and node models specified in the *Node Editor*. The Node Editor describes nodes' internal architecture, functional elements and data flow between them. Node models in turn, consist of modules with process models specified by the *Process Editor*. The lowest level of the network hierarchy, the process models, describes the module's behaviour in terms of protocols, algorithms, and applications using finite state machines and a high-level language.

There are several other editors to define various data models referenced by process- or node-level models, e.g., packet formats and control information between processes. Additional editors create, edit, and view probability density functions (PDFs) to control certain events, such as the interarrival time of sending or receiving packets, etc. The model-specification editors provide a graphical interface for the user to manipulate objects representing the models and the corresponding processes. Each editor can specify objects and operations corresponding to the model's abstraction level. Therefore, the Project Editor specifies nodes and link objects of a network, the Node Editor specifies processors, queues, transmitters, and receivers in the network nodes, and the Process Editor specifies the states and transitions in the processes. Figure 14.12 depicts the abstraction level of each editor:

14.6.3. Data collection and simulation

OPNET can produce many types of output during simulation depending on how the modeller defined the types of output. In most cases, modellers use the built in types of data: output vectors, output scalars, and animation:

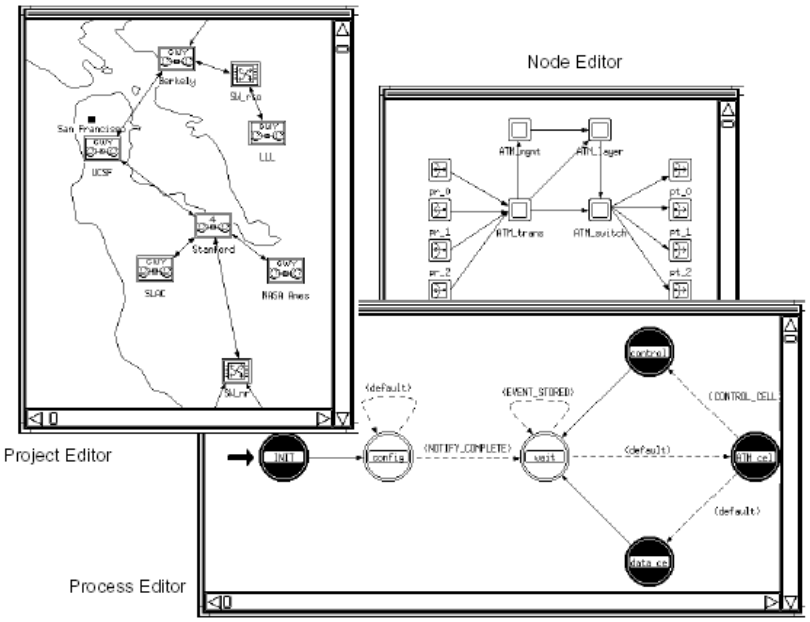


Figure 14.12. The three modelling abstraction levels specified by the Project, Node, and Process editors.

- Output vectors represent time-series simulation data consisting of list of entries, each of which is a time-value pair. The first value in the entries can be considered as the independent variable and the second as the dependent variable.
- Scalar statistics are individual values derived from statistics collected during simulation, e.g., average transmission rate, peak number of dropped cells, mean response time, or other statistics.
- OPNET can also generate animations that are viewed during simulation or replay after simulation. The modeller can define several forms of animations, for instance, packet flows, state transitions, and statistics.

14.6.4. Analysis

Typically, much of the data collected during simulations is stored in output scalar and output vector files. In order to analyse these data OPNET provides the *Analysis Tool* which is a collection of graphing and numerical processing functions. The Analysis Tool presents data in the form of graphs or traces. Each trace consists of a list of abscissa X and ordinate Y pairs. Traces are held and displayed in analysis panels. The Analysis Tool supports a variety of methods for processing simulation output data and computing new traces. Calculations, such as histograms, PDF,

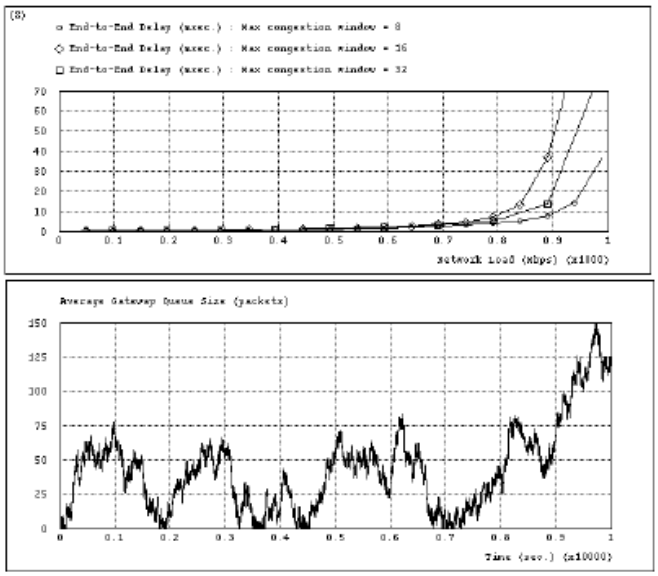


Figure 14.13. Example for graphical representation of scalar data (upper graph) and vector data (lower graph).

CDF, and confidence intervals are included. Analysis Tool also supports the use of mathematical filters to process vector or trace data. Mathematical filters are defined as hierarchical block diagrams based on a predefined set of calculus, statistical, and arithmetic operators. The example diagrams below (Figures 14.13 and 14.14) shows graphs generated by the Analysis Tool:

Figure 14.14 Analysis Tool Showing Four Graphs.

COMNET is another popular discrete-event simulation system. We will discuss it briefly and demonstrate its features in Section 14.9.

14.6.5. Network Analysers – Application Characterisation Environment (ACE)

There is an increasing interest in predicting, measuring, modelling, and diagnosing application performance across the application lifecycle from development through deployment to production. Characterising the application's performance is extremely important in critical application areas, like in eCommerce. In the increasingly competitive eCommerce, the application's performance is critical, especially where the competition is just "one click" away. Application performance affects revenue. When an application performs poorly it is always the network that is blamed rather than the application. These performance problems may result from several areas including application design or slow database servers. Using tools, like ACE and Network As-

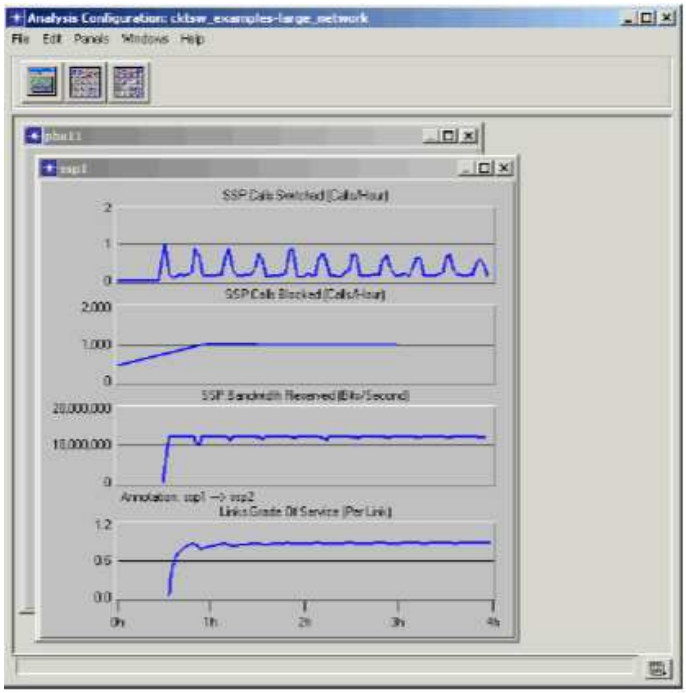


Figure 14.14. Figure 14.14 shows four graphs represented by the Analysis Tool.

sociates' Sniffer, network modellers can develop methodologies to identify the source of application slowdowns and resolve their causes. After analysing the applications, modellers can make recommendations for performance optimisation. The result is faster applications and better response times. The Application Characterisation Environment (ACE) is a tool for visualising, analysing, and troubleshooting network applications. Network managers and application developers can use ACE to

- Locate network and application bottlenecks.
- Diagnose network and application problems.
- Analyse the affect of anticipated network changes on the response time of existing applications.
- Predict application performance under varying configurations and network conditions.

The performance of an application is determined by network attributes that are affected by the various components of a communication network. The following list contains some example for these attributes and the related network elements:

- *Network media*
  - Bandwidth (Congestion, Burstiness)
  - Latency (TCP window size, High latency devices, Chatty applications)

- *Nodes*
- *Clients*
  - User time
  - Processing time
  - Starved for data
- *Servers*
  - Processing time
  - Multi-tier waiting data
  - Starved for data
- *Application*
  - Application turns (Too many turns – Chatty applications)
  - Threading (Single vs. multi-threaded)
  - Data profile (Bursty, Too much data processing)

Analysis of an application requires two phases:

- Capture packet traces while an application is running to build a baseline for modelling an application. We can use the ACE's capturing tool or any other network analysers to capture packet traces. The packet traces can be captured by strategically deployed capture agents.
- Import the capture file to create a representation of the application's transactions called an application task for further analysis of the messages and protocol data units generated by the application.

After creating the application task, we can perform the following operations over the captured traffic traces:

- View and edit the captured packet traces on different levels of the network protocol stack in different windows. We can also use these windows to remove or delete sections of an application task. In this way, we focus on transactions of our interest.
- Perform application level analysis by identifying and diagnosing bottlenecks. We can measure the components of the total response time in terms of application level time, processing time, and network time and view detailed statistics on the network and application. We can also decode and analyse the network and application protocol data units from the contents of the packet traces.
- Predict application performance in "what-if" scenarios and for testing projected changes.

Without going into specific details we illustrate some of the features above through a simple three-tier application. We want to determine the reason or reasons of the slow response time from a Client that remotely accesses an Application Server (App Server) to retrieve information from a Database Server (DB Server). The connection is over an ADSL line between the client and the Internet, and a 100Mbps

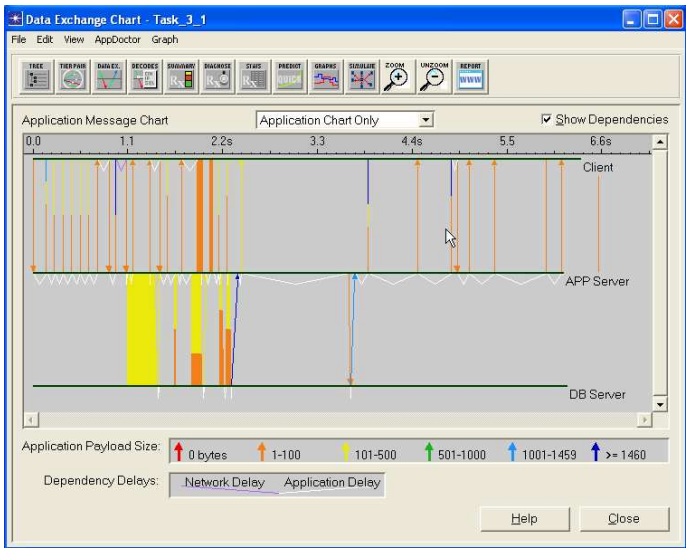


Figure 14.15. Data Exchange Chart.

Ethernet connection between the App Server and the DB Server. We want to identify the cause of the slow response time and recommend solutions. We deployed capture agents at the network segments between the client and the App Server and between the servers. The agents captured traffic traces simultaneously during a transaction between the client and the App Server and the App Server and the DB Server respectively. Then, the traces were merged and synchronised to obtain the best possible analysis of delays at each tier and in the network.

After importing the trace into ACE, we can analyse the transaction in the *Data Exchange Chart*, which depicts the flow of application messages among tiers over time.

The Data Exchange Chart shows packets of various sizes being transmitted between the Client and the servers. The overall transaction response time is approximately 6 seconds. When the "Show Dependencies" checkbox is checked, the white dependency lines indicate large processing delays on the Application Server and Client tiers. For further analysis, we generate the "Summary of Delays" window showing how the total response time of the application is divided into four general categories: Application delay, Propagation delay, Transmission delay and Protocol/Congestion delay. Based on this chart we can see the relation between application and network related delays during the transaction between the client and the servers. The chart clearly shows that the application delay far outweighs the Propagation, Transmission, and Protocol/Congestion delays slowing down the transaction.

The "Diagnosis" function (Figure 14.17) provides a more granular analysis of possible bottlenecks by analysing factors that often cause performance problems in networked applications. Values over a specified threshold are marked as bottlenecks or potential bottlenecks.

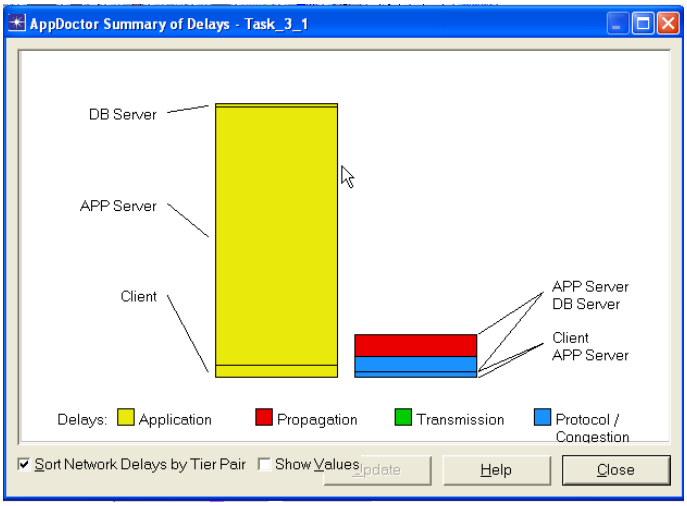


Figure 14.16. Summary of Delays.

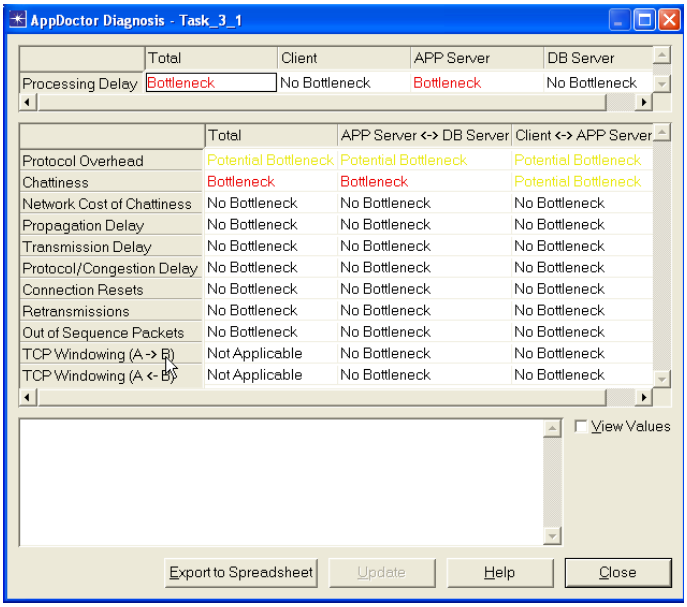


Figure 14.17. Diagnosis window.

The diagnosis of the transaction confirms that the primary bottleneck is due to Processing Delay on the Application Server. The processing delay is due to the file I/O, CPU processing, or memory access. It also reveals another bottleneck: the chattiness of the application that leads us to the next step. We investigate the

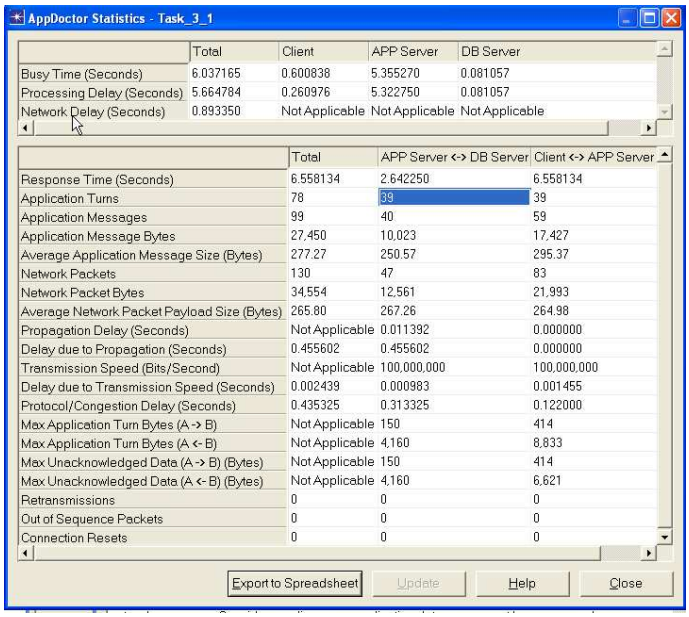


Figure 14.18. Statistics window.

application behaviour in terms of application turns that can be obtained from the transaction statistics. An application turn is a change in direction of the application-message flow.

The statistics of the transaction (Figure 14.18) disclose that the number of application turns is high, i.e., the data sent by the transaction at a time is small. This may cause significant application and network delays. Additionally, a significant portion of application processing time can be spent processing the many requests and responses. The Diagnosis window indicates a "Chattiness" bottleneck without a "Network Cost of Chattiness" bottleneck, which means the following:

- The application does not create significant network delays due to chattiness.
- The application creates significant processing delays due to overhead associated with handling many small application level requests and responses.
- The application's "Network Cost of Chattiness" could dramatically increase in a high-latency network.

The recommendation is that the application should send fewer, larger application messages. This will utilise network and tier resources more efficiently. For example, a database application should avoid sending a set of records one record at a time.

Would the response time decrease significantly if we added more bandwidth to the link between the client and the APP Server (Figure 14.19)? Answering this question is important because adding more bandwidth is expensive. Using the prediction feature we can answer the question. In the following chart we selected the bandwidth from 128K to 10Mbps. The chart shows that beyond approximately 827 Kbps there

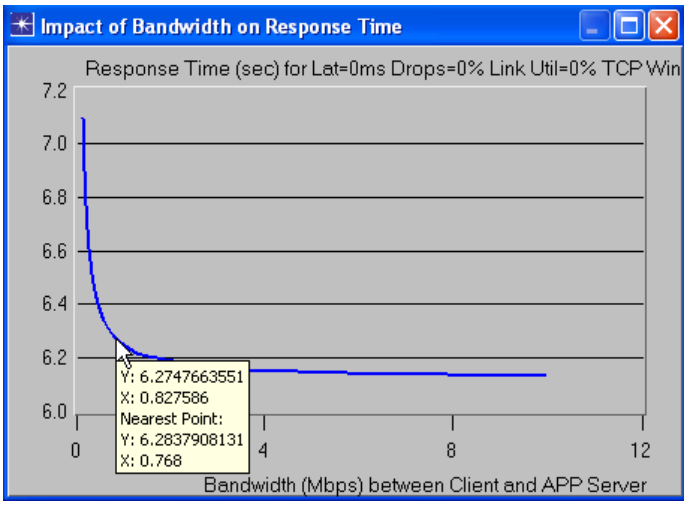


Figure 14.19. Impact of adding more bandwidth on the response time.

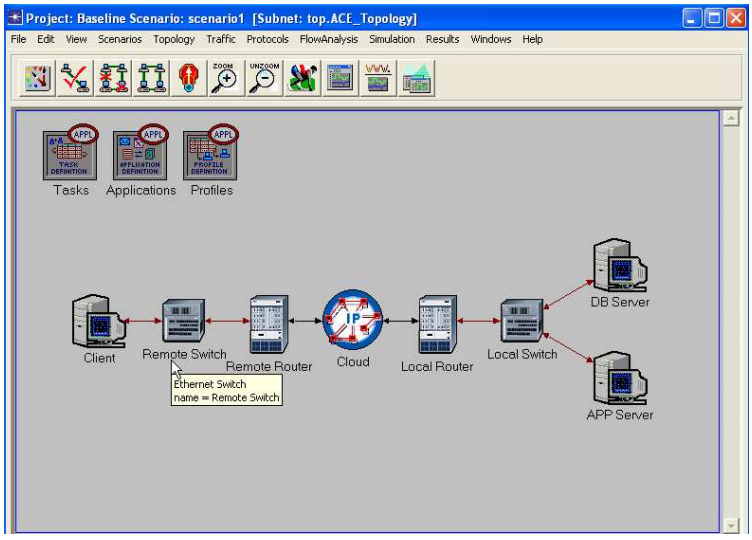


Figure 14.20. Baseline model for further simulation studies.

is no significant improvement in response time, i.e., for this application the recommended highest bandwidth is no more than 827Kbps, which can be provided by a higher speed DSL line.

After the analysis of the application's performance, we can immediately create the starting baseline model from the captured traffic traces for further simulation studies as illustrated in Figure 14.20.

14.6.6. Sniffer

Another popular network analyser is Network Associates' *Sniffer*. (Network Associates has recently renamed it to Netasyst.) It is a powerful network visualisation tool consisting of a set of functions to:

- Capture network traffic for detailed analysis.
- Diagnose problems using the *Expert Analyzer*.
- Monitor network activity in real time.
- Collect detailed utilisation and error statistics for individual stations, conversations, or any portion of your network.
- Save historical utilisation and error information for baseline analysis.
- Generate visible and audible real-time alarms and notify network administrators when troubles are detected.
- Probe the network with active tools to simulate traffic, measure response times, count hops, and troubleshoot problems.

For further details we refer the reader to the vendors' documentations on http://www.nai.com.

14.7. Model Development Life Cycle (MDLC)

There are several approaches for network modelling. One possible approach is the creation of a starting model that follows the network topology and approximates the assumed network traffic statistically. After some changes are made, the modeller can investigate the impact of the changes of some system parameters on the network or application performance. This is an approach when it is more important to investigate the performance difference between two scenarios rather than starting from a model based on real network traffic. For instance, assuming certain client/server transactions, we want to measure the change of the response time as the function of the link utilisation 20%, 40%, 60%, etc. In this case it is not extremely important to start from a model based on actual network traffic. It is enough to specify certain amount of data transmission estimated by a frequent user or designer. We investigate, for this amount of data, how much the response time will increase as the link utilisation increases relative to the starting scenario.

The most common approach for network modelling follows the methodologies of proactive network management. It implies the creation of a network model using actual network traffic as input to simulate current and future behaviour of the network and predict the impact of the addition of new applications on the network performance. By making use of modelling and simulation tools network managers can change the network model by adding new devices, workstations, servers, and applications. Or they can upgrade the links to higher speed network connections and perform"what-if" scenarios before the implementation of the actual changes. We follow this approach in our further discussions because this approach has been widely accepted in the academia, corporate world, and the industry. In the subsequent pa-

ragraphs we elaborate a sequence of modelling steps, called the **M**odel **D**evelopment **L**ife **C**ycle – MDLC that the author has applied in various real life scenarios of modelling large enterprise networks. The MDLC has the following steps:

- Identification of the topology and network components.
- Data collection.
- Construction and validation of the baseline model. Perform network simulation studies using the baseline.
- Creation of the application model using the details of the traffic generated by the applications.
- Integration of the application and baseline model and completion of simulation studies.
- Further data gathering as the network growths and changes and as we know more about the applications.
- Repeat the same sequence.

In the following, we expand the steps above:

Identification of the topology and network components
Topology data describes the physical network components (routers, circuits, and servers) and how they are connected. It includes the location and configuration description of each internetworking device, how those devices are connected (the circuit types and speeds), the type of LANs and WANs, the location of the servers, addressing schemes, a list of applications and protocols, etc.

Data collection
In order to build the baseline model we need to acquire topology and traffic data. Modellers can acquire topology data either by entering the data manually or by using network management tools and network devices' configuration files. Several performance management tools use the **S**imple **N**etwork **M**anagement **P**rotocol – SNMP to query the *Management Information Base (MIB)* maintained by SNMP agents running in the network's routers and other internetworking devices. This process is known as an SNMP discovery. We can import topology data from routers' configuration files to build a representation of the topology for the network in question. Some performance management tools can import data using the map file from a network management platform, such as HP OpenView or IBM NetView. Using the network management platform's export function, the map file can be imported by modelling.

The network traffic input to the baseline model can be derived from various sources: Traffic descriptions from interviews and network documents, design or maintenance documents, MIB/SNMP reports and network analyser and **R**emote **M**onitoring—traffic traces. RMON is a network management protocol that allows network information to be gathered at a single node. RMON traces are collected by RMON probes that collect data at different levels of the network architecture depending on the probe's standard. Figure 14.21 includes the most widely used standards and the level of data collection:

	RMON1	RMON2	Enterprise RMON
Ethernet/Token Ring	X	X	X
MAC Layer Monitoring	X	X	X
Network Layer Monitoring		X	X
Application Layer Monitoring		X	X
Switched LAN, Frame Relay, ATM			X
VLAN Support			X
Application response time			X

Figure 14.21. Comparison of RMON Standards.

Network traffic can be categorised as usage-based data and application-based data. The primary difference between usage- and application-based data is the degree of details that the data provides and the conclusions that can be made based on the data. The division can be clearly specified by two adjacent OSI layers, the Transport layer and the Session layer: usage-based data is for investigating the performance issues through the transport layer; application-based data is for analysing the rest of the network architecture above the Transport layer. (In Internet terminology this is equivalent to the cut between the TCP level and the applications above the TCP level.)

The goal of collecting usage-based data is to determine the total traffic volume before the applications are implemented on the network. Usage-based data can be gathered from SNMP agents in routers or other internetworking devices. SNMP queries sent to the routers or switches provide statistics about the exact number of bytes that have passed through each LAN interface, WAN circuit, or (**P**ermanent **V**irtual **C**ircuit – PVC) interfaces. We can use the data to calculate the percentage of utilisation of the available bandwidth for each circuit.

The purpose of gathering application-based data is to determine the amount of data generated by an application and the type of demand the application makes. It allows the modeller to understand the behaviour of the application and to characterise the application level traffic. Data from traffic analysers or from RMON2-compatible probes, Sniffer, NETScout Manager, etc., provide specifics about the application traffic on the network. Strategically placed data collection devices can gather enough data to provide clear insight into the traffic behaviour and flow patterns of the network applications. Typical application level data collected by traffic analysers:

- The type of applications.
- Hosts communicating by network layer addresses (i.e., IP addresses).
- The duration of the network conversation between any two hosts (start time and end time).
- The number of bytes in both the forward and return directions for each network conversation.
- The average size of the packets in the forward and return directions for each network conversation.
- Traffic burstiness.

- Packet size distributions.
- Packet interarrival distributions.
- Packet transport protocols.
- Traffic profile, i.e., message and packet sizes, interarrival times, and processing delays.
- Frequency of executing application for a typical user.
- Major interactions of participating nodes and sequences of events.

Construction and validation of the baseline model. Perform network simulation studies using the baseline

The goal of building a baseline model is to create an accurate model of the network as it exists today. The baseline model reflects the current "as is" state of the network. All studies will assess changes to the baseline model. This model can most easily be validated since its predictions should be consistent with current network measurements. The baseline model generally only predicts basic performance measures such as resource utilisation and response time.

The baseline model is a combination of the topology and usage-based traffic data that have been collected earlier. It has to be validated against the performance parameters of the current network, i.e., we have to prove that the model behaves similarly to the actual network activities. The baseline model can be used either for analysis of the current network or it can serve as the basis for further application and capacity planning. Using the import functions of a modelling tool, the baseline can be constructed by importing first the topology data gathered in the data collection phase of the modelling life cycle. Topology data is typically stored in topology files (.top or .csv) created by Network Management Systems, for instance HP OpenView or Network Associate's Sniffer. Traffic files can be categorised as follows:

- Conversation pair traffic files that contain aggregated end-to-end network load information, host names, packet counts, and byte counts for each conversation pair. The data sets allow the modelling tool to preserve the bursty nature of the traffic. These files can be captured by various data collection tools.
- Event trace traffic files that contain network load information in the form of individual conversations on the network rather than summarised information. During simulation the file can replay the captured network activity on an event by event basis.

Before simulation the modeller has to decide on the following simulation parameters:

- *Run length:* ***Runtime length*** must exceed the longest message delay in the network. During this time the simulation should produce sufficient number of events to allow the model to generate enough samples of every event.
- *Warm-up period:* The simulation ***warm-up period*** is the time needed to initialise packets, buffers, message queues, circuits, and the various elements of the model. The warm-up period is equal to a typical message delay between hosts. Simulation warm-up is required to ensure that the simulation has reached

steady-state before data collection begins.

- *Multiple replications:* There may be a need for multiple runs of the same model in cases when statistics are not sufficiently close to true values. We also need multiple runs prior to validation when we execute multiple replicates to determine variation of statistics between replications. A common cause of variation between replications is rare events.
- *Confidence interval:* A confidence interval is an interval used to estimate the likely size of a population parameter. It gives an estimated range of values that has a specified probability of containing the parameter being estimated. Most commonly used intervals are the 95% and 99% confidence intervals that have .95 and .99 probabilities respectively of containing the parameter. In simulation, confidence interval provides an indicator of the precision of the simulation results. Fewer replications result in a broader confidence interval and less precision.

In many modelling tools, after importing both the topology and traffic files, the baseline model is created automatically. It has to be checked for construction errors prior to any attempts at validation by performing the following steps:

- Execute a preliminary run to confirm that all source-destination pairs are present in the model.
- Execute a longer simulation with warm-up and measure the sent and received message counts and link utilisation to confirm that correct traffic volume is being transmitted.

Validating the baseline model is the proof that the simulation produces the same performance parameters that are confirmed by actual measurements on the physical network. The network parameters below can usually be measured in both the model and in the physical network:

- Number of packets sent and received
- Buffer usage
- Packet delays
- Link utilisation
- Node's CPU utilisation

Confidence intervals and the number of independent samples affect how close a match between the model and the real network is to be expected. In most cases, the best that we can expect is an overlap of the confidence interval of predicted values from the simulation and the confidence interval of the measured data. A very close match may require too many samples of the network and too many replications of the simulation to make it practical.

Creation of the application model using the details of the traffic generated by the applications

Application models are studied whenever there is a need to evaluate the impact of a networked application on the network performance or to evaluate the application's performance affected by the network. Application models provide traffic details between network nodes generated during the execution of the application. The steps of

building an application model are similar to the ones for baseline models.

- Gather data on application events and user profiles.
- Import application data into a simulation model manually or automatically.
- Identify and correct any modelling errors.
- Validate the model.

Integration of the application and baseline models and completion of simulation studies

The integration of the application model(s) and baseline model follows the following steps:

- Start with the baseline model created from usage-based data.
- Use the information from the application usage scenarios (locations of users, number of users, transaction frequencies) to determine where and how to load the application profiles onto the baseline model.
- Add the application profiles generated in the previous step to the baseline model to represent the additional traffic created by the applications under study.

Completion of Simulation studies consists of the following steps:

- Use a modelling tool to run the model or simulation to completion.
- Analyse the results: Look at the performance parameters of the target transactions in comparison to the goals established at the beginning of the simulation.
- Analyse the utilisation and performance of various network elements, especially where the goals are not being met.

Typical simulation studies include the following cases:

- Capacity analysis

  Capacity analysis studies the changes of network parameters, for instance:

  - Changes in the number and location of users.
  - Changes in network elements capacity.
  - Changes in network technologies.

  A modeller may be interested in the effect of the changes above on the following network parameters:

  - Switches and routers' utilisation
  - Communications link utilisation
  - Buffer utilisation
  - Retransmitted and lost packets

- Response time analysis

  The scope of response time analysis is the study of message and packet transmission delay:

  - Application and network level packet end-to-end delay.

  - Packet round trip delay.
  - Message/packet delays.
  - Application response time.

- Application Analysis

  The scope of application studies is the ratio of the total application response time relative to the individual components of network and application delay. Application's analysis provides statistics of various measures of network and application performance in addition to the items discussed in a previous section.

Further data gathering as the network growths and as we know more about the applications
The goal of this phase is to analyse or predict how a network will perform both under current conditions and when changes to traffic load (new applications, users, or network structure) are introduced:

- Identify modifications to the network infrastructure that will alter capacity usage of the network's resources.
- A redesign can include increasing or decreasing capacity, relocating network elements among existing network sites, or changing communications technology.
- Modify the models to reflect these changes.
- Assess known application development or deployment plans in terms of projected network impact.
- Assess business conditions and plans in terms of their impact on the network from projected additional users, new sites, and other effects of the plans.
- Use ongoing Baselining techniques to watch usage trends over time, especially related to Internet and intranet usage.

14.8. Modelling of traffic burstiness

Recent measurements of local area network traffic and wide-area network traffic have proved that the widely used Markovian process models cannot be applied for today's network traffic. If the traffic were a Markovian process, the traffic's burst length would be smoothed by averaging over a long time scale, contradicting the observations of today's traffic characteristics. Measurements of real traffic also prove that traffic burstiness is present on a wide range of time scales. Traffic that is bursty on many or all time scales can be characterised statistically using the concept of *self-similarity*. Selfsimilarity is often associated with objects in fractal geometry, objects that appear to look alike regardless of the scale at which they are viewed. In case of stochastic processes like time series, the term self-similarity refers to the process' distribution, which, when viewed at varying time scales, remains the same. Self-similar time series has noticeable bursts, which have long periods with extremely high values on all time scales. Characteristics of network traffic, such as packets/sec,

bytes/sec, or length of frames, can be considered as stochastic time series. Therefore, measuring traffic burstiness is the same as characterising the self-similarity of the corresponding time series.

The self-similarity of network traffic has also been observed in studies in numerous papers. These and other papers show that packet loss, buffer utilisation, and response time are totally different when simulations use either real traffic data or synthetic data that include self-similarity.

Background

Let $X = (X_t : t = 0, 1, 2, \ldots)$ be a covariance stationary stochastic process. Such a process has a constant mean $\mu = \mathrm{E}[X_t]$, finite variance $\sigma^2 = \mathrm{E}\left[(X_t - \mu)^2\right]$, and an autocorrelation function $r(k) = \mathrm{E}\left[(X_t - \mu)(X_{t+k} - \mu)\right] / \mathrm{E}\left[(X_t - \mu)^2\right]$ $(k = 0, 1, 2, \ldots)$, that depends only on k. It is assumed that X has an autocorrelation function of the form:

$$r(k) \sim \alpha k^{-\beta}, \quad k \to \infty \tag{14.1}$$

where $0 < \beta < 1$ and α is a positive constant. Let $X^{(m)} = (X_{(k)}^{(m)} : k = 1, 2, 3,\ m = 1, 2, 3, \ldots)$ represent a new time series obtained by averaging the original series X over nonoverlapping blocks of size m. For each $m = 1, 2, 3, \ldots, X^{(m)}$ is specified by $X_k^{(m)} = (X_{km-m+1} + \cdots + X_{km})/m, (k \geq 1)$. Let $r^{(m)}$ denote the autocorrelation function of the aggregated time series $X^{(m)}$.

Definition of self-similarity

The process X called exactly ***self-similar*** with self-similarity parameter $H = 1 - \beta/2$ if the corresponding aggregated processes $X^{(m)}$ have the same correlation structure as X, i.e. $r^{(m)}(k) = r(k)$ for all $m = 1, 2, \ldots$ $(k = 1, 2, 3, \ldots)$.

A covariance stationary process X is called ***asymptotically self-similar*** with self-similarity parameter $H = 1 - \beta/2$, if for all k large enough $r^{(m)}(k) \to r(k)$, as $m \to \infty$, $0.5 \leq H \leq 1$.

Definition of long-range dependency

A stationary process is called ***long-range dependent*** if the sum of the autocorrelation values approaches infinity: $\sum_k r(k) \to \infty$. Otherwise, it is called ***short-range dependent***. It can be derived from the definitions that while short-range dependent processes have exponentially decaying autocorrelations, the autocorrelations of long-range dependent processes decay hyperbolically; i.e., the related distribution is heavy-tailed. In practical terms, a random variable with heavy-tail distribution generates extremely large values with high probability. The degree of self-similarity is expressed by the parameter H or ***Hurst-parameter.*** The parameter represents the speed of decay of a process' autocorrelation function. As $H \to 1$ the extent of both self-similarity and long-range dependence increases. It can also be shown that for self-similar processes with long-range dependency $H > 0.5$.

Traffic models

Traffic modelling originates in traditional voice networks. Most of the models have relied on the assumption that the underlying processes are Markovian (or more ge-

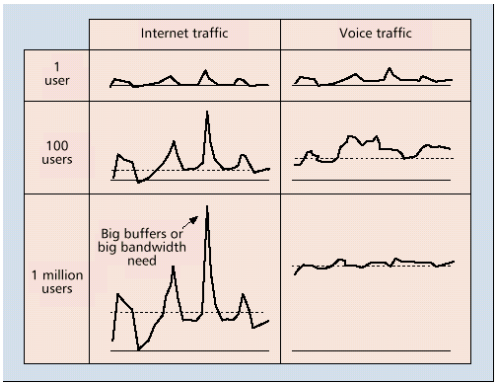


Figure 14.22. The self-similar nature of Internet network traffic.

neral, short-range dependent). However, today's high-speed digital packet networks are more complex and bursty than traditional voice traffic due to the diversity of network services and technologies.

Several sophisticated stochastic models have been developed as a reaction to new developments, such as Markov-modulated Poisson processes, fluid flow models, Markovian arrival processes, batched Markovian arrival process models, packet train models, and *Transform-Expand-Sample models* . These models mainly focus on the related queueing problem analytically. They are usually not compared to real traffic patterns and not proven to match the statistical property of actual traffic data.

Another category of models attempts to characterise the statistical properties of actual traffic data. For a long time, the area of networking research has lacked adequate traffic measurements. However, during the past years, large quantities of network traffic measurements have become available and collected in the Web and high-speed networks. Some of these data sets consist of high-resolution traffic measurements over hours, days, or weeks. Other data sets provide information over time periods ranging from weeks to months and years. Statistical analyses of these high time-resolution traffic measurements have proved that actual traffic data from packet networks reveal self-similarity. These results point out the difference between traditional models and measured traffic data. While the assumed processes in traditional packet traffic models are short-range dependent, measured packet traffic data show evidence of long-range dependency. Figure 14.22 illustrates the difference between Internet traffic and voice traffic for different numbers of aggregated users. As the number of voice flows increases, the traffic becomes more and more smoothed contrary to the Internet traffic.

Quite the opposite to the well developed field of short-range dependent queueing models, fewer theoretical results exist for queueing systems with long-range dependence. For some of the results. In terms of modelling, the two major groups of self-similar models are fractional Gaussian noises and fractional ARIMA processes. The Gaussian models accurately represent aggregation of many traffic streams. Another well-known model, the M/Pareto model has been used in modelling network

traffic that is not sufficiently aggregated for the Gaussian model to apply.

Black box vs. structural models

We share the opinion calling the approach of traditional time series analysis as black box modelling as opposite to the structural modelling that concentrates on the environment in which the models' data was collected; i.e., the complex hierarchies of network components that make up today's communications systems. While the authors admit that black box models can be and are useful in other contexts, they argue that black box models are of no use for understanding the dynamic and complex nature of the traffic in modern packet networks. Black box models have not much use in designing, managing and controlling today's networks either. In order to provide physical explanations for empirically observed phenomena such as long-range dependency, we need to replace black box models with structural models. The attractive feature of structural traffic models is that they take into account the details of the layered architecture of today's networks and can analyse the interrelated network parameters that ultimately determine the performance and operation of a network. Time series models usually handle these details as black boxes. Because actual networks are complex systems, in many cases, black box models assume numerous parameters to represent a real system accurately. For network designers, who are important users of traffic modelling, black box models are not very useful. It is rarely possible to measure or estimate the model's numerous parameters in a complex network environment. For a network designer, a model ought to be simple, meaningful in a particular network. It can relay on actual network measurements, and the result ought to be relevant to the performance and the operation of a real network.

For a long time, traffic models were developed independently of traffic data collected in real networks. These models could not be applied in practical network design. Today the availability of huge data sets of measured network traffic and the increasing complexity of the underlying network structure emphasise the application of the Ockham' Razer in network modelling. (Ockham's Razor is a principle of the mediaeval philosopher William Ockham. According to his principle, modellers should not make more assumptions than the minimum needed. This principle is also called the Principle of Parsimony and motivates all scientific modelling and theory building. It states that modellers should choose the simplest model among a set of otherwise equivalent models of a given phenomenon. In any given model, Ockham's Razor helps modellers include only those variables that are really needed to explain the phenomenon. Following the principle, model development will become easier, reducing the possibilities for inconsistencies, ambiguities and redundancies.)

Structural models are presented, for instance in different papers, which demonstrate how the self-similar nature of aggregated network traffic of all conversations between hosts explains the details of the traffic dynamics at the level generated by the individual hosts. The papers introduce structural traffic models that have a physical meaning in the network context and underline the predominance of long-range dependence in the packet arrival patterns generated by the individual conversations between hosts. The models provide insight into how individual network connections behave in local and wide area networks. Although the models go beyond the

black box modelling methodology by taking into account the physical structure of the aggregated traffic patterns, they do not include the physical structure of the intertwined structure of links, routers, switches, and their finite capacities along the traffic paths.

Crovella and Stavros demonstrated that World Wide Web traffic shows characteristics that are consistent with self-similarity. They show that transmission times may be heavy tailed, due to the distribution of available file sizes in the Web. It is also shown that silent times may also be heavy-tailed; primarily due to the effect of user "think time". Similarly to the structural models due to Willinger at al., their paper lacks of analysing the impact of selfsimilar traffic on the parameters of the links and the routers' buffers that ultimately determine a network's performance.

This chapter describes a traffic model that belongs to the structural model category above. We implement the M/Pareto model within the discrete event simulation package COMNET that allows the analysis of the negative impact of self-similar traffic on not just one single queue, but on the overall performance of various interrelated network components, such as link, buffers, response time, etc. The commercially available package does not readily provide tools for modelling self-similar, long-range dependent network traffic. The model-generated traffic is based on measurements collected from a real ATM network. The choice of the package emphasises the need for integrated tools that could be useful not just for theoreticians, but also for network engineers and designers. Our paper intends to narrow the gap between existing, well-known theoretical results and their applicability in everyday, practical network analysis and modelling. It is highly desirable that appropriate traffic models should be accessible from measuring, monitoring, and controlling tools. Our model can help network designers and engineers, the ultimate users of traffic modelling, understand the dynamic nature of network traffic and assist them to design, measure, monitor, and control today's complex, high-speed networks in their everyday's practice.

Implications of burstiness on high-speed networks

Various papers discuss the impact of burstiness on network congestion. Their conclusions are:

- Congested periods can be quite long with losses that are heavily concentrated.
- Linear increases in buffer size do not result in large decreases in packet drop rates.
- A slight increase in the number of active connections can result in a large increase in the packet loss rate.

Results show that packet traffic "spikes" (which cause actual losses) ride on longerterm "ripples", which in turn ride on still longer-term "swells".

Another area where burstiness can affect network performance is a link with priority scheduling between classes of traffic. In an environment, where the higher priority class has no enforced bandwidth limitations (other than the physical bandwidth), interactive traffic might be given priority over bulk-data traffic. If the higher priority class is bursty over long time scales, then the bursts from the higher priority traffic could obstruct the lower priority traffic for long periods of time.

The burstiness may also have an impact on networks where the admission control mechanism is based on measurements of recent traffic, rather than on policed traffic parameters of individual connections. Admission control that considers only recent traffic patterns can be misled following a long period of fairly low traffic rates.

14.8.1. Model parameters

Each transaction between a client and a server consists of active periods followed by inactive periods. Transactions consist of groups of packets sent in each direction. Each group of packets is called a burst. The burstiness of the traffic can be characterised by the following time parameters:

- ***Transaction Interarrival Time*** (TIAT): The time between the first packet in a transaction and the first packet of the next immediate transaction.
- ***Burst Interarrival Time,*** $1/\lambda$***,*** λ ***arrival rate of bursts:*** The time between bursts.
- ***Packet Interarrival Time,*** $1/r$***,*** r***: arrival rate of packets:*** The time between packets in a burst.

The Hurst parameter

It is anticipated that the rapid and ongoing aggregation of more and more traffic onto integrated multiservice networks will eventually result in traffic smoothing. Once the degree of aggregation is sufficient, the process can be modelled by Gaussian process. Currently, network traffic does not show characteristics that close to Gaussian. In many networks the degree of aggregation is not enough to balance the negative impact of bursty traffic. However, before traffic becomes Gaussian, existing methods can still provide accurate measurement and prediction of bursty traffic.

Most of the methods are based on the estimate of the Hurst parameter H - the higher the value of H, the higher the burstiness, and consequently, the worse the queueing performance of switches and routers along the traffic path. Some are more reliable than others. The reliability depends on several factors; e.g., the estimation technique, sample size, time scale, traffic shaping or policing, etc. Based on published measurements we investigated methods with the smallest estimation error*. [1] Among those, we chose the *Rescaled Adjusted Range* (R/S) method because we found it implemented in the Benoit package. The Hurst parameter calculated by the package is input to our method.

The M/Pareto traffic model and the Hurst parameter

Recent results have proven that the M/Pareto model is appropriate for modelling long-range dependent traffic flow characterised by long bursts. Originally, the model was introduced and applied in the analysis of ATM buffer levels. The M/Pareto model was also used to predict the queueing performance of Ethernet, VBR video, and IP packet streams in a single server queue. We apply the M/Pareto model not just for a single queue, but also for predicting the performance of an interconnec-

[1] Variance, Aggregated Variance, Higuchi, Variance of Residuals, Rescaled Adjusted Range (R/S), Whittle Estimator, Periodogram, Residuals of Regression.

ted system of links, switches and routers affecting the individual network elements' performance.

The M/Pareto model is a Poisson process of overlapping bursts with arrival rate λ. A burst generates packets with arrival rate r. Each burst, from the time of its interval, will continue for a Pareto-distributed time period. The use of Pareto distribution results in generating extremely long bursts that characterise long-range dependent traffic.

The probability that a Pareto-distributed random variable X exceeds threshold x is:

$$\Pr\{X > x\} = \begin{cases} \left(\frac{x}{\delta}\right)^{\gamma}, & x \geq \delta \\ 1, & \text{otherwise} \end{cases}, \tag{14.2}$$

$$1 < \gamma < 2, \delta > 0 .$$

The mean of X, the mean duration of a burst $\mu = \delta\gamma/(\gamma - 1)$ and its variance is infinite. Assuming a t time interval, the mean number of packets M in the time interval t is:

$$M = \lambda t r \delta \gamma / (\gamma - 1) , \tag{14.3}$$

where

$$\lambda = \frac{M(\gamma - 1)}{t r \delta \gamma} . \tag{14.4}$$

The M/Pareto model is asymptotically self-similar and it is shown that for the Hurst parameter the following equation holds:

$$H = \frac{3 - \gamma}{2} . \tag{14.5}$$

14.8.2. Implementation of the Hurst parameter in the COMNET modelling tool

We implemented the Hurst parameter and a modified version of the M/Pareto model in the discrete event simulation system COMNET. By using discrete event simulation methodology, we can get realistic results in measuring network parameters, such as utilisation of links and the queueing performance of switches and routers. Our method can model and measure the harmful consequences of aggregated bursty traffic and predict its impact on the overall network's performance.

Traffic measurements

In order to build the baseline model, we collected traffic traces in a large corporate network by the Concord Network Health network analyser system. We took measurements from various broadband and narrow band links including 45Mbps ATM, 56Kbps, and 128 Kbps frame relay connections. The Concord Network Health system can measure the traffic in certain time intervals at network nodes, such as routers and switches. We set the time intervals to 6000 seconds and measured the number of bytes and packets sent and received per second, packet latency, dropped packets, discard eligible packets, etc. Concord Network Health cannot measure the number of packets in a burst and the duration of the bursts as it is assumed in the M/Pareto

Delta Time (sec)	Average Bandwidth utilisation %	Bytes Total/ sec	Bytes in/ sec	Bytes out/ sec
299	2.1	297.0	159.2	137.8
300	2.2	310.3	157.3	153.0
301	2.1	296.8	164.4	132.4
302	2.7	373.2	204.7	168.5
...	...	...	...	...

Figure 14.23. Traffic traces.

Bytes average number	Message delay (ms)	Buffer level (byte)	Dropped packets number	Links' Mean Bandwidth utilisation (%)		
				56 Kbps Frame Relay	ATM DS-3 segment	100 Mbps Ethernet
440.4279	78.687	0.04	0	3.14603	0.06	0.0031

Figure 14.24. Measured network parameters.

model above. Due to this limitation of our measuring tool, we slightly modify our traffic model according to the data available. We took snapshots of the traffic in every five minutes from a narrow band frame relay connection between a remote client workstation and a server at the corporate headquarters as traffic destination in the following format:

The mean number of bytes, the message delay from the client to server, the input buffer level at the client's local router, the number of blocked packets, the mean utilisations of the 56Kbps frame relay, the DS-3 segment of the ATM network, and the 100Mbps Ethernet link at the destination are summarised in Figure 14.24.

COMNET represents a transaction by a message source, a destination, the size of the message, communication devices, and links along the path. The rate at which messages are sent is specified by an interarrival time distribution, the time between two consecutive packets. The Poisson distribution in the M/Pareto model generates bursts or messages with arrival rate λ, the number of arrivals, which are likely to occur in a certain time interval. In simulation, this information is expressed by the time interval between successive arrivals $1/\lambda$. For this purpose, we use the Exponential distribution. Using the Exponential distribution for interarrival time will result in an arrival pattern characterised by the Poisson distribution. In COMNET, we implemented the interarrival time with the function $\mathrm{Exp}(1/\lambda)$. The interarrival time in the model is set to one second matching the sampling time interval set in Concord Network Health and corresponding to an arrival rate $\lambda = 1/\mathrm{sec}$.

In the M/Pareto model, each burst continues for a Pareto-distributed time period. The Concord Network Health cannot measure the duration of a burst; hence, we assume that a burst is characterised by the number of bytes in a message sent or received in a second. Since the ATM cell rate algorithm ensures that equal length messages are processed in equal time, then longer messages require longer processing time. So we can say that the distribution of the duration of bursts is the same as the distribution of the length of bursts. Hence, we can modify the M/Pareto model by substituting the Pareto-distributed duration of bursts with the Pareto-distributed length of bursts. We derive δ of the Pareto distribution not from the mean duration of bursts, but from the mean length of bursts.

The Pareto distributed length of bursts is defined in COMNET by two

parameters- the location and the shape. The location parameter corresponds to the δ, the shape parameter corresponds to the δ parameter of the M/Pareto model in (1) and can be calculated from the relation (4) as

$$\gamma = 3 - 2H \ . \tag{14.6}$$

The Pareto distribution can have infinite mean and variance. If the shape parameter is greater than 2, both the mean and variance are finite. If the shape parameter is greater than 1, but less than or equal to 2, the mean is finite, but then the variance is infinite. If the shape parameter is less than or equal to 1, both the mean and variance are infinite.

From the mean of the Pareto distribution we get:

$$\delta = \frac{\mu \cdot (\gamma - 1)}{\gamma} \ . \tag{14.7}$$

The relations (5) and (6) allow us to model bursty traffic based on real traffic traces by performing the following steps:

a. Collect traffic traces using the Concord Network Health network analyser.

b. Compute the Hurst parameter H by making use of the Benoit package with the traffic trace as input.

c. Use the Exponential and Pareto distributions in the COMNET modelling tool with the parameters calculated above to specify the distribution of the interarrival time and length of messages.

d. Generate traffic according to the modified M/Pareto model and measure network performance parameters.

The traffic generated according to the steps above is bursty with parameter H calculated from real network traffic.

14.8.3. Validation of the baseline model

We validate our baseline model by comparing various model parameters of a 56Kbps frame relay and a 6Mbps ATM connection with the same parameters of a real network as the Concord Network Health network analyser traced it. For simplicity, we use only the "Bytes Total/sec" column of the trace, i.e., the total number of bytes in the "Bytes Total/sec" column is sent in one direction only from the client to the server. The Hurst parameter of the real traffic trace is $H = 0.55$ calculated by the Benoit package. The topology is as follows:

The "Message sources" icon is a subnetwork that represents a site with a token ring network, a local router, and a client A sending messages to the server B in the "Destination" subnetwork:

The interarrival time and the length of messages are defined by the Exponential and Pareto functions Exp (1) and Par (208.42, 1.9) respectively. The Pareto distribution's location (208.42) and shape (1.9) are calculated from formulas (5) and (6) by substituting the mean length of bursts (440 bytes from Table 2.) and $H = 0.55$.

The corresponding heavy-tailed Pareto probability distribution and cumulative

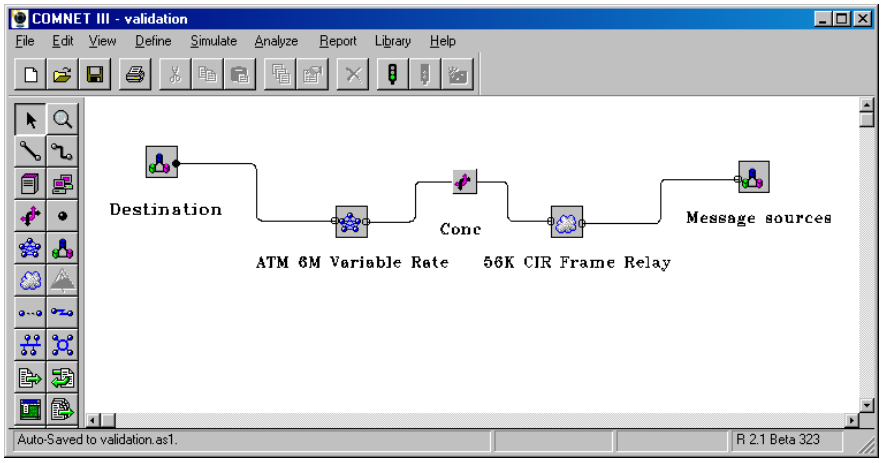


Figure 14.25. Part of the real network topology where the measurements were taken.

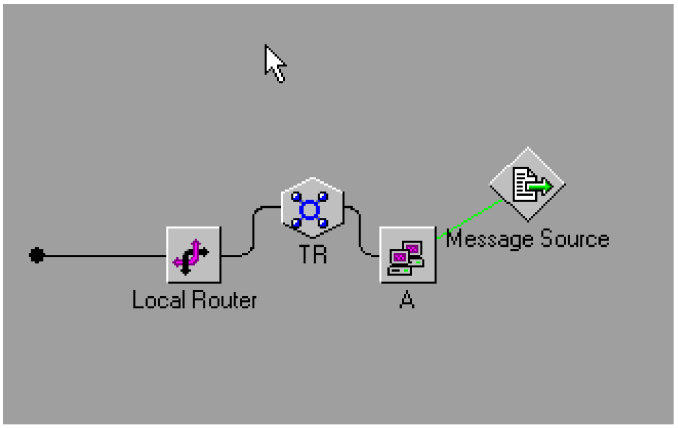


Figure 14.26. "Message Source" remote client.

distribution functions are illustrated in Figure 14.28 (The $X-axis$ represents the number of bytes):

The "Frame Relay" icon represents a frame relay cloud with 56K committed information rate (CIR). The "Conc" router connects the frame relay network to a 6Mbps ATM network with variable rate control (VBR) as shown in Figures 14.29 and 14.30:

The "Destination" icon denotes a subnetwork with server B:

The results of the model show almost identical average for the utilisation of the frame relay link (0.035 ~3.5%) and the utilisation of the real measurements (3.1%):

The message delay in the model is also very close to the measured delay between

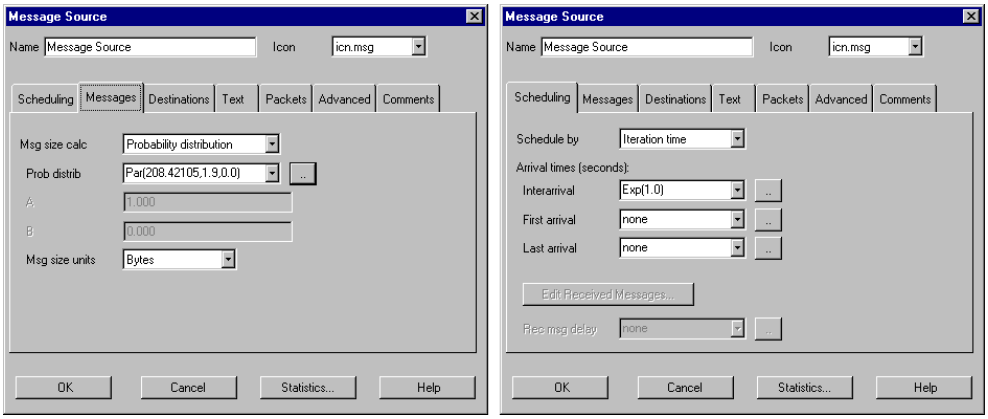


Figure 14.27. Interarrival time and length of messages sent by the remote client.

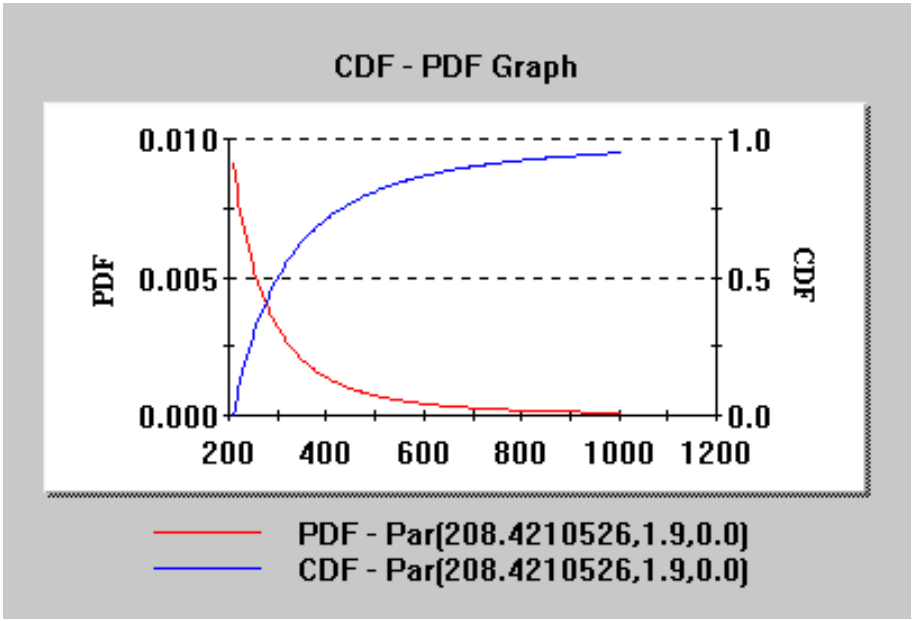


Figure 14.28. The Pareto probability distribution for mean 440 bytes and Hurst parameter $H = 0.55$.

the client and the server (78 msec):

The input buffer level of the remote client's router in the model is almost identical with the measured buffer level of the corresponding router:

Similarly, the utilisations of the model's DS-3 link segment of the ATM network and the Ethernet link in the destination network closely match with the measure-

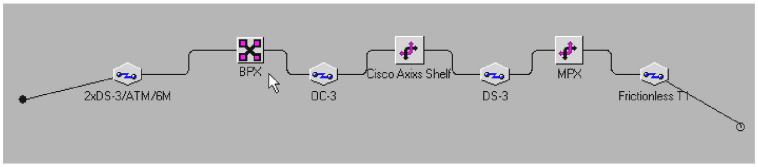


Figure 14.29. The internal links of the 6Mbps ATM network with variable rate control (VBR).

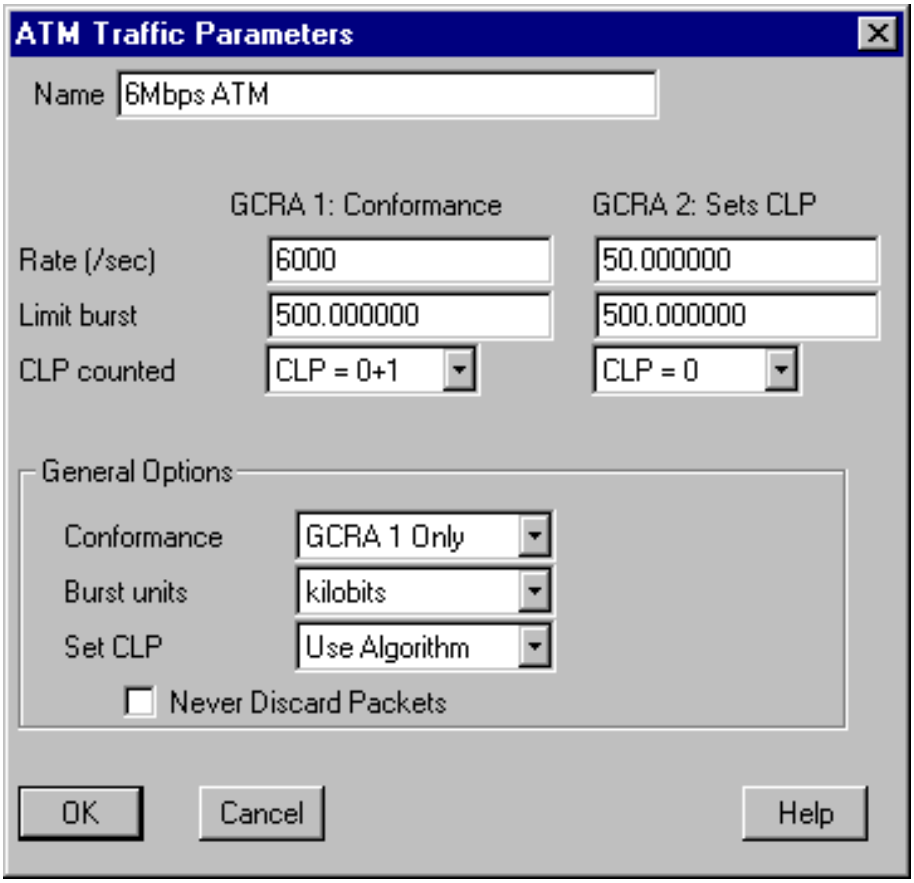


Figure 14.30. Parameters of the 6Mbps ATM connection.

ments of the real network:

It can also be shown from the model's traffic trace that for the model generated messages the Hurst parameter $H = 0.56$, i.e., the model generates almost the same bursty traffic as the real network. Furthermore, the number of dropped packets in the model was zero similarly to the number of dropped packets in the real measurements.

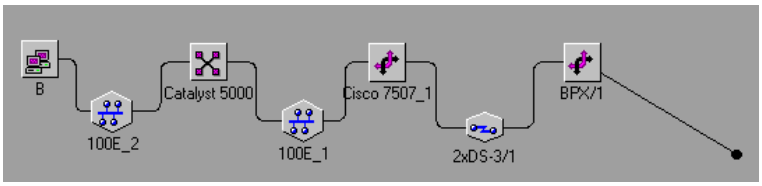


Figure 14.31. The "Destination" subnetwork.

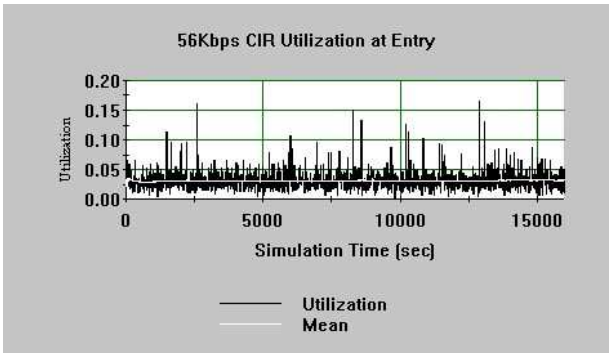


Figure 14.32. utilisation of the frame relay link in the baseline model.

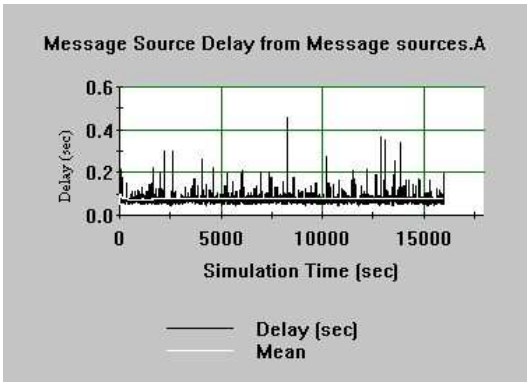


Figure 14.33. Baseline message delay between the remote client and the server.

Therefore, we start from a model that closely represents the real network.

14.8.4. Consequences of traffic burstiness

In order to illustrate our method, we developed a COMNET simulation model to measure the consequences of bursty traffic on network links, message delays, routers'

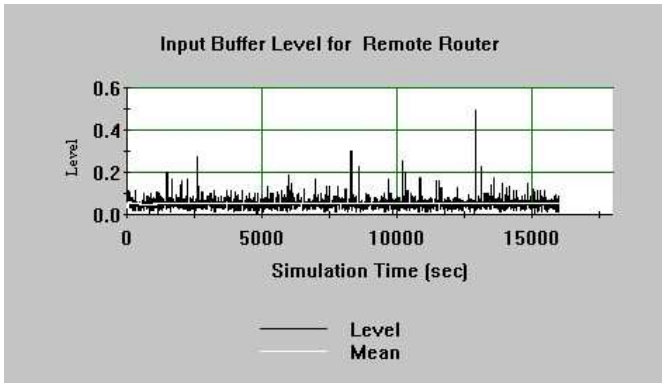


Figure 14.34. Input buffer level of remote router.

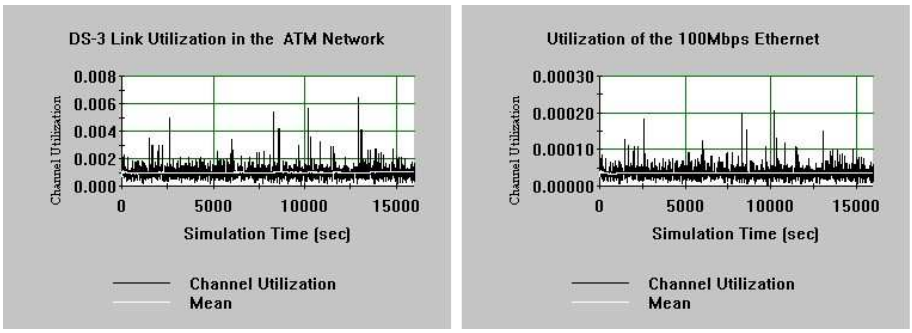


Figure 14.35. Baseline utilisations of the DS-3 link and Ethernet link in the destination.

input buffers, and the number of dropped packets due to the aggregated traffic of large number of users. The model implements the Hurst parameter as it has been described in Section 3. We repeated the simulation for 6000 sec, 16000 sec and 18000 sec to allow infrequent events to occur a reasonable number of times. We found that the results are very similar in each simulation.

Topology of bursty traffic sources
The "Message Source" subnetworks transmit messages as in the baseline model above, but with different burstiness: $H = 0.95, H = 0.75, H = 0.55$, and with fixed size. Initially, we simulate four subnetworks and four users per subnetwork each sending the same volume of data (mean 440 bytes per second) as in the validating model above:

Link utilisation and message delay
First, we are going to measure and illustrate the extremely high peaks in frame relay link utilisation and message delay. The model traffic is generated with message

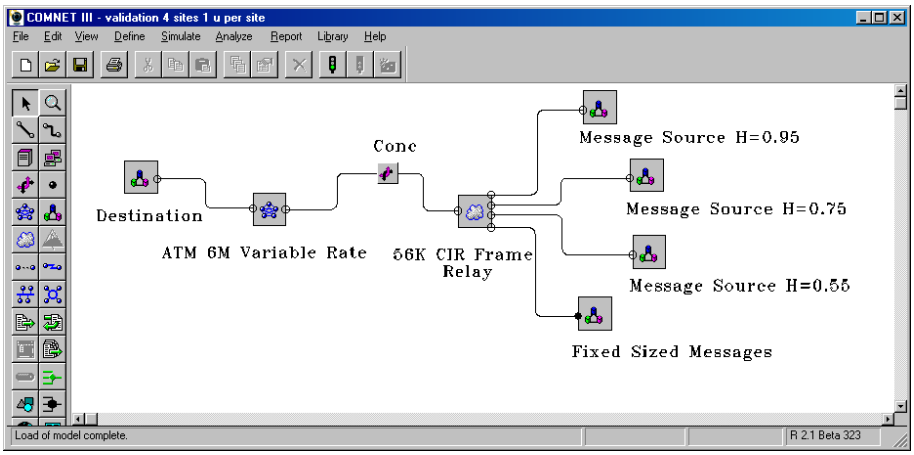


Figure 14.36. Network topology of bursty traffic sources with various Hurst parameters.

	Fixed size messages	$H = 0.55$	$H = 0.75$	$H = 0.95$
Average utilisation	0.12	0.13	0.13	0.14
Peak utilisation	0.18	0.48	1	1

Figure 14.37. Simulated average and peak link utilisation.

	Fixed size messages	$H = 0.55$	$H = 0.75$	$H = 0.95$
Average response time (ms)	75.960	65.61	87.880	311.553
Peak response time (ms)	110.06	3510.9	32418.7	112458.08
Standard deviation	0.470	75.471	716.080	4341.24

Figure 14.38. Response time and burstiness.

sizes determined by various Hurst parameters and fixed size messages for comparison. The COMNET modelling tool has a trace option to capture its own model generated traffic. It has been verified that for the model-generated traffic flows with various Hurst parameters the Benoit package computed similar Hurst parameters for the captured traces.

The following table shows the simulated average and peak link utilisation of the different cases. The utilisation is expressed in the [0, 1] scale not in percentages:

The enclosed charts in Appendix A clearly demonstrate that even though the average link utilisation is almost identical, the frequency and the size of the peaks increase with the burstiness, causing cell drops in routers and switches. We received the following results for response time measurements:

The charts in the Appendix A graphically illustrate the relation between response times and various Hurst parameters.

Input buffer level for large number of users

We also measured the number of cells dropped at a router's input buffer in the ATM network due to surge of bursty cells. We simulated the aggregated traffic of

	Fixed size message	$H = 0.55$	$H = 0.75$	$H = 0.95$
Packets accepted	13282	12038	12068	12622
Packets blocked	1687	3146	3369	7250
Average buffer use in bytes	56000858	61001835	62058222	763510495

Figure 14.39. Relation between the number of cells dropped and burstiness.

approximately 600 users each sending the same number of bytes in a second as in the measured real network. The number of blocked packets is summarised in the following table:

14.8.5. Conclusion

Theis chapter presented a discrete event simulation methodology to measure various network performance parameters while transmitting bursty traffic. It has been proved in recent studies that combining bursty data streams will also produce bursty combined data flow. The studies imply that the methods and models used in traditional network design require modifications. We categorise our modelling methodology as a structural model contrary to a black box model. Structural models focus on the environment in which the models' data was collected; i.e., the complex hierarchies of network components that make up today's communications systems. Although black box models are useful in other contexts, they are not easy to use in designing, managing and controlling today's networks. We implemented a well-known model, the M/Pareto model within the discrete event simulation package COMNET that allows the analysis of the negative impact of self-similar traffic on not just one single queue, but on the overall performance of various interrelated network components as well. Using real network traces, we built and validated a model by which we could measure and graphically illustrate the impact of bursty traffic on link utilisation, message delays, and buffer performance of Frame Relay and ATM networks. We illustrated that increasing burstiness results in extremely high link utilisation, response time, and dropped packets, and measured the various performance parameters by simulation.

The choice of the package emphasises the need for integrated tools that could be useful not just for theoreticians, but also for network engineers and designers. Our paper intends to narrow the gap between existing, well-known theoretical results and their applicability in everyday, practical network analysis and modelling. It is highly desirable that appropriate traffic models should be accessible from measuring, monitoring, and controlling tools. Our model can help network designers and engineers, the ultimate users of traffic modelling, understand the dynamic nature of network traffic and assist them in their everyday practice.

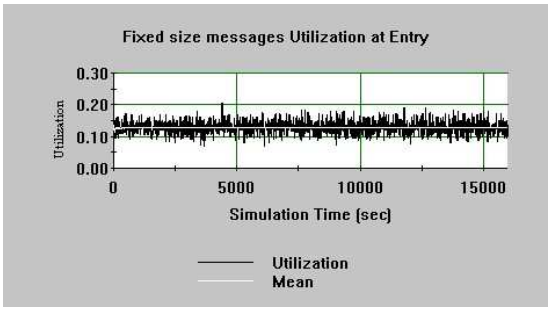


Figure 14.40. Utilisation of the frame relay link for fixed size messages.

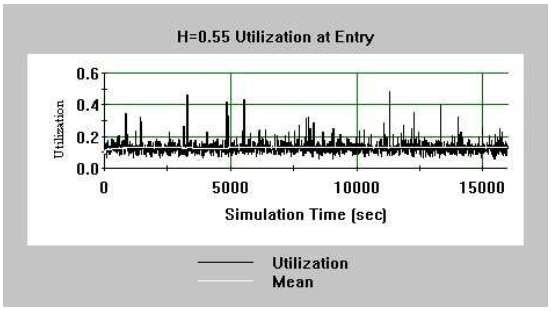


Figure 14.41. Utilisation of the frame relay link for Hurst parameter $H = 0.55$.

14.9. Appendix A

14.9.1. Measurements for link utilisation

The following charts demonstrate that even though the average link utilisation for the various Hurst parameters is almost identical, the frequency and the size of the peaks increase with the burstiness, causing cell drops in routers and switches. The utilisation is expressed in the [0, 1] scale not in percentages:

14.9.2. Measurements for message delays

Figures 14.43–14.45 illustrate the relation between response time and various Hurst parameters:

Exercises

14.9-1 Name some attributes, events, activities and state variables that belong to the following concepts:

- Server
- Client

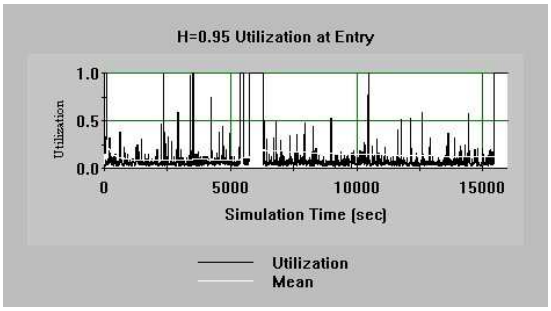


Figure 14.42. Utilisation of the frame relay link for Hurst parameter $H = 0.95$ (many high peaks).

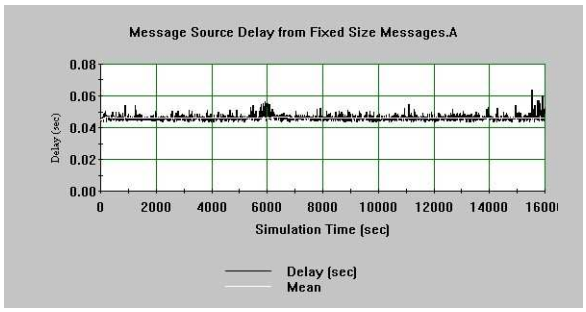


Figure 14.43. Message delay for fixed size message.

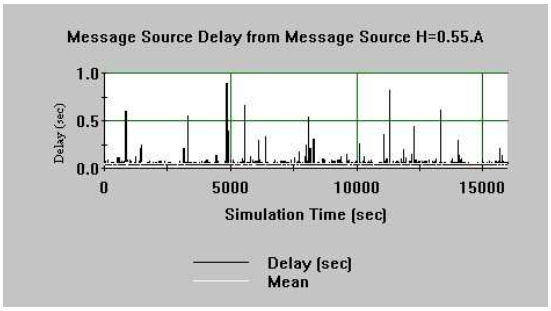


Figure 14.44. Message delay for $H = 0.55$ (longer response time peaks).

- Ethernet
- Packet switched network
- Call set up in cellular mobile network
- TCP *Slow start algorithm*

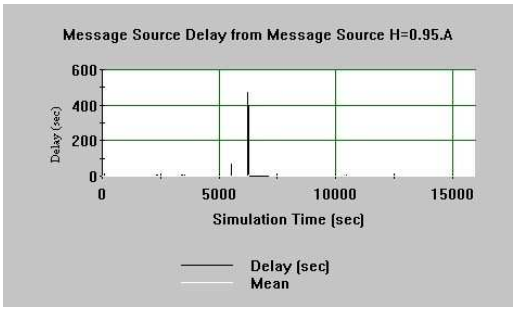


Figure 14.45. Message delay for $H = 0.95$ (extremely long response time peak).

14.9-2 Read the article about the application of the network simulation and write a report about how the article approaches the validation of the model.

14.9-3 For this exercise it is presupposed that there is a network analyser software (e.g., LAN Analyzer for Windows or any similar) available to analyse the network traffic. We use the mentioned software thereinafter.

- Let's begin to transfer a file between a client and a server on the LAN. Observe the detailed statistics of the utilisation of the datalink and the number of packets per second then save the diagram.
- Read the *Capturing and analysing Packet* chapter in the Help of LAN Analyzer.
- Examine the packets between the client and the server during the file transfer.
- Save the captured trace information about the packets in .csv format. Analyse this file using spreadsheet manager. Note if there are any too long time intervals between two packets, too many bad packets, etc. in the unusual protocol events.

14.9-4 In this exercise we examine the network analysing and baseline maker functions of the Sniffer. The baseline defines the activities that characterise the network. By being familiar with this we can recognise the non-normal operation. This can be caused by a problem or the growing of the network. Baseline data has to be collected in case of typical network operation. For statistics like bandwidth utilization and number of packets per second we need to make a chart that illustrates the information in a given time interval. This chart is needed because sampled data of a too short time interval can be false. After adding one or more network component a new baseline should be made, so that later the activities before and after the expansion can be compared. The collected data can be exported to be used in spreadsheet managers and modelling tools, that provides further analysing possibilities and is helpful in handling gathered data.

Sniffer is a very effective network analysing tool. It has several integrated functions.

- Gathering traffic-trace information for detailed analysis.

- Problem diagnosis with Expert Analyzer.
- Real-time monitoring of the network activities.
- Collecting detailed error and utilization statistics of nodes, dialogues or any parts of the network.
- Storing the previous utilization and fault information for baseline analysis.
- When a problem occurs it creates visible or audible alert notifications for the administrators.
- For traffic simulation monitoring of the network with active devices, measuring the response time, hop counting and faults detection.
- The Histroy Samples option of the Monitor menu allows us to record the network activities within a given time interval. This data can be applied for baseline creation that helps to set some thresholds. In case of non-normal operation by exceeding these thresholds some alerts are triggered. Furthermore this data is useful to determine the long-period alteration of the network load, therefore network expansions can be planned forward.
- Maximum 10 of network activities can be monitored simultaneously. Multiple statistics can be started for a given activity, accordingly short-period and long-period tendencies can be analysed concurrently. Network activities that are available for previous statistics depends on the adapter selected in the Adapter dialogue box. For example in case of a token ring network the samples of different token ring frame types (e.g, Beacon frames), in Frame Relay networks the samples of different Frame Relay frame types (e.g, LMI frames) can be observed. The available events depend on the adapter.

Practices:

- Set up a filter (Capture/Define filter) between your PC and a remote Workstation to sample the IP traffic.
- Set up the following at the Monitor/History Samples/Multiple History: Octets/sec, utilization, Packets/sec, Collisions/sec and Broadcasts/sec.
- Configure sample interval for 1 sec. (right click on the Multiple icon and Properties/Sample).
- Start network monitoring (right click on the Multiple icon and Start Sample).
- Simulate a typical network traffic, e.g, download a large file from a server.
- Record the "Multiple History" during this period of time. This can be considered as baseline.
- Set the value of the Octets/sec tenfold of the baseline value at the Tools/Options/MAC/Threshold. Define an alert for the Octets/sec: When this threshold exceeded, a message will be sent to our email address. On Figure 14.46 we suppose that this threshold is 1,000.
- Alerts can be defined as shown in Figure 14.47.
- Set the SMTP server to its own local mail server (Figure 14.48).
- Set the Severity of the problem to Critical (Figure 14.49).

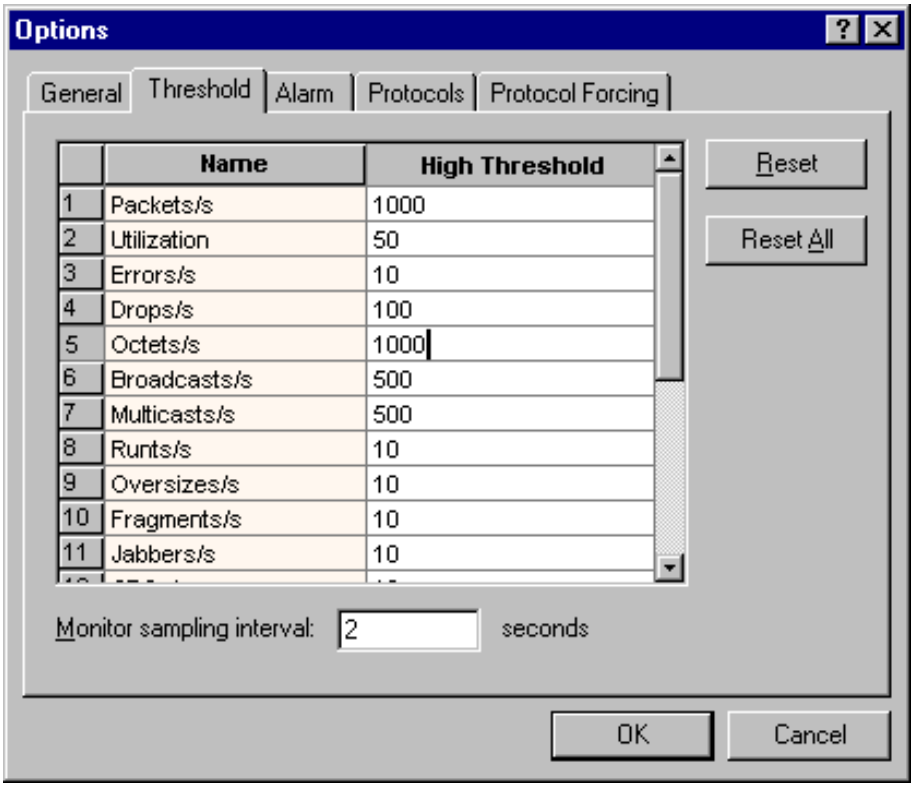


Figure 14.46. Settings.

- Collect tracing information (Capture/Start) about network traffic during file download.
- Stop capture after finished downloading (Capture/Stop then Display).
- Analyse the packets' TCP/IP layers with the Expert Decode option.
- Check the "Alert message" received from Sniffer Pro. Probably a similar message will be arrived that includes the octets/sec threshold exceeded:

```
From: ...
Subject: Octets/s: current value = 22086, High Threshold = 9000
To: ...

This event occurred on ...
```

Save the following files:

- The "Baseline screens"
- The Baseline Multiple History.csv file
- The "alarm e-mail".

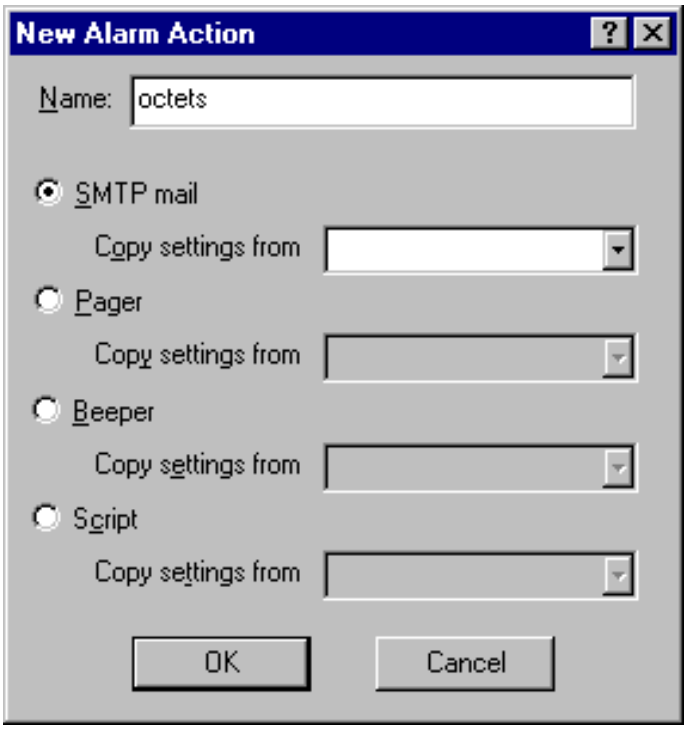


Figure 14.47. New alert action.

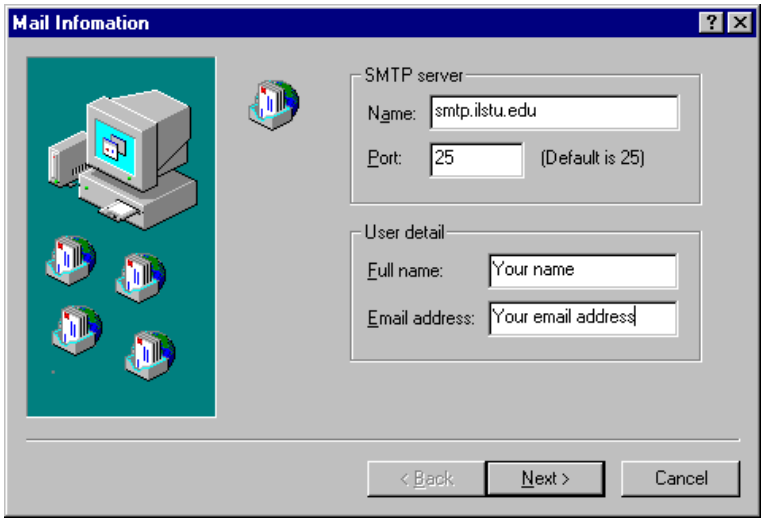


Figure 14.48. Mailing information.

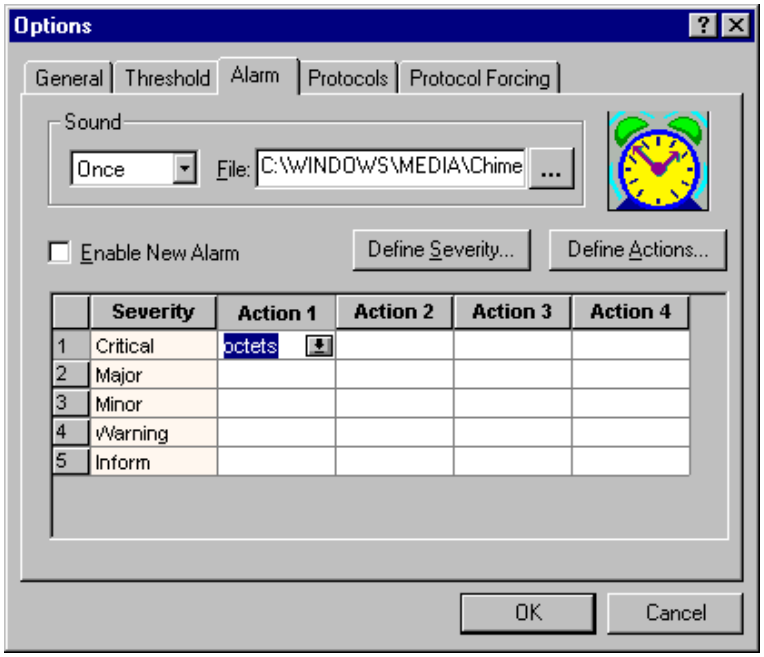


Figure 14.49. Settings.

14.9-5 The goal of this practice is to build and validate a baseline model using a network modelling tool. It's supposed that a modelling tool such as COMNET or OPNET is available for the modeller.

First collect response time statistics by pinging a remote computer. The ping command measures the time required for a packet to take a round trip between the client and the server. A possible format of the command is the following: ping hostname -n x -l y -w z > filename where "x" is the number of packet to be sent, "y" is the packet length in bytes, "z" is the time value and "filename" is the name of the file that includes the collected statistics.

For example the ping 138.87.169.13 -n 5 -l 64 > c:
ping.txt command results the following file:

```
Pinging 138.87.169.13 with 64 bytes of data:
Reply from 138.87.169.13: bytes=64 time=178ms TTL=124
Reply from 138.87.169.13: bytes=64 time=133ms TTL=124
Reply from 138.87.169.13: bytes=64 time=130ms TTL=124
Reply from 138.87.169.13: bytes=64 time=127ms TTL=124
Reply from 138.87.169.13: bytes=64 time=127ms TTL=124
```

- Create a histogram for these time values and the sequence number of the packets by using a spreadsheet manager.

- Create a histogram about the number of responses and the response times.
- Create the cumulative density function of the response times indicating the details at the tail of the distribution.
- Create the baseline model of the transfers. Define the traffic attributes by the density function created in the previous step.
- Validate the model.
- How much is the link utilization in case of messages with length of 32 and 64 bytes?

14.9-6 It is supposed that a modelling tool (e.g., COMNET, OPNET, etc.) is available for the modeller. In this practice we intend to determine the place of some frequently accessed image file in a lab. The prognosis says that the addition of clients next year will triple the usage of these image files. These files can be stored on the server or on the client workstation. We prefer storing them on a server for easier administration. We will create a baseline model of the current network, we measure the link-utilization caused by the file transfers. Furthermore we validate the model with the correct traffic attributes. By scaling the traffic we can create a forecast about the link- utilization in case of trippled traffic after the addition of the new clients.

- Create the topology of the baseline model.
- Capture traffic trace information during the transfer and import them.
- Run and validate the model (The number of transferred messages in the model must be equal to the number in the trace file, the time of simulation must be equal to the sum of the **I**nterpacket **T**imes and the link utilization must be equal to the average utilization during capture).
- Print reports about the number of transferred messages, the message delays, the link utilization of the protocols and the total utilization of the link.
- Let's triple the traffic.
- Print reports about the number of transferred messages, the message delay, the link utilization of the protocols and the total utilization of the link.
- If the link-utilization is under the baseline threshold then we leave the images on the server otherwise we move them to the workstations.
- What is your recommendation: Where is better place to store the image files, the client or the server?

14.9-7 The aim of this practice to compare the performance of the shared and the switched Ethernet. It can be shown that transformation of the shared Ethernet to switched one is only reasonable if the number of collisions exceeds a given threshold.

a. Create the model of a client/server application that uses shared Ethernet LAN. The model includes 10Base5 Ethernet that connects one Web server and three group of workstations. Each group has three PCs, furthermore each group has a source that generates "Web Request" messages. The Web server application of the server

responds to them. Each "Web Request" generates traffic toward the server. When the "Web Request" message is received by the server a "Web Response" message is generated and sent to the appropriate client.

- Each "Web Request" means a message with 10,000 bytes of length sent by the source to the Web Server every Exp(5) second. Set the text of the message to "Web Request".
- The Web server sends back a message with the "Web Response" text. The size of the message varies between 10,000 and 100,000 bytes that determined by the Geo(10000, 100000) distribution. The server responds only to the received "Web Request" messages. Set the reply message to "Web Response".
- For the rest of the parameters use the default values.
- Select the "Channel Utilization" and the ("Collision Stats") at the ("Links Reports").
- Select the "Message Delay" at the ("Message + Response Source Report").
- Run the simulation for 100 seconds. Animation option can be set.
- Print the report that shows the "Link Utilization", the "Collision Statistics" and the report about the message delays between the sources of the traffic.

b. In order to reduce the response time transform the shared LAN to switched LAN. By keeping the clien/server parameters unchanged, deploy an Ethernet switch between the clients and the server. (The server is connected to the switch with full duplex 10Base5 connection.)

- Print the report of "Link Utilization" and "Collision Statistics", furthermore the report about the message delays between the sources of the traffic.

c. For all of the two models change the 10Base5 connections to 10BaseT. Unlike the previous situations we will experience a non-equivalent improvement of the response times. We have to give explanation.

14.9-8 A part of a corporate LAN consists of two subnets. Each of them serves a department. One operates according to IEEE 802.3 CSMA/CD 10BaseT Ethernet standard, while the other communicates with IEEE 802.5 16Mbps Token Ring standard. The two subnets are connected with a Cisco 2500 series router. The Ethernet LAN includes 10 PCs, one of them functions as a dedicated mail server for all the two departments. The Token Ring LAN includes 10 PC's as well, one of them operates as a file server for the departments.

The corporation plans to engage employees for both departments. Although the current network configuration won't be able to serve the new employees, the corporation has no method to measure the network utilization and its latency. Before engaging the new employees the corporation would like to estimate these current baseline levels. Employees have already complained about the slowness of download from the file server.

According to a survey, most of the shared traffic flown through the LAN originates from the following sources: electronic mailing, file transfers of applications and voice based messaging systems (Leaders can send voice messages to their employees). The conversations with the employees and the estimate of the average size of the

messages provides the base for the statistical description of the message parameters.

E-mailing is used by all employees in both departments. The interviews revealed that the time interval of the mail sending can be characterised with an Exponential distribution. The size of the mails can be described with an Uniform distribution accordingly the mail size is between 500 and 2,000 bytes. All of the emails are transferred to the email server located in the Ethernet LAN, where they are be stored in the appropriate user's mailbox.

The users are able to read messages by requesting them from the email server. The checking of the mailbox can be characterised with a Poisson distribution whose mean value is 900 seconds. The size of the messages used for this transaction is 60 bytes. When a user wants to download an email, the server reads the mailbox file that belongs to the user and transfers the requested mail to the user's PC. The time required to read the files and to process the messages inside can be described with an Uniform distribution that gathers its value from the interval of 3 and 5 seconds. The size of the mails can be described with a normal distribution whose mean value is 40,000 bytes and standard deviation is 10,000 bytes.

Both departments have 8 employees, each of them has their own computer, furthermore they download files from the file server. Arrival interval of these requests can be described as an Exponential distribution with a mean value of 900 ms. The requests' size follows Uniform distribution, with a minimum of 10 bytes minimum and a maximum of 20 bytes. The requests are only sent to the file server located in the Token Ring network. When a request arrives to the server, it read the requested file and send to the PC. This processing results in a very low latency. The size of the files can be described with a normal distribution whose mean value is 20,000 bytes and standard deviation is 25,000 bytes.

Voice-based messaging used only by the heads of the two departments, sending such messages only to theirs employees located in the same department. The sender application makes connection to the employee's PC. After successful connection the message will be transferred. The size of these messages can be described by normal distribution with a mean value of 50,000 bytes and a standard deviation of 1,200 bytes. Arrival interval can be described with a Normal distribution whose mean value is 1,000 seconds and standard deviation is 10 bytes.

TCP/IP is used by all message sources, and the estimated time of packet construction is 0.01 ms.

The topology of the network must be similar to the one in COMNET, Figure 14.50.

The following reports can be used for the simulation:

- Link Reports: Channel Utilization and Collision Statistics for each link.
- Node Reports: Number of incoming messages for the node.
- Message and Response Reports: The delay of the messages for each node.
- Session Source Reports: Message delay for each node.

By running the model, a much higher response time will be observed at the file server. What type of solution can be proposed to reduce the response time when the quality of service level requires a lower response time? Is it a good idea to set up a

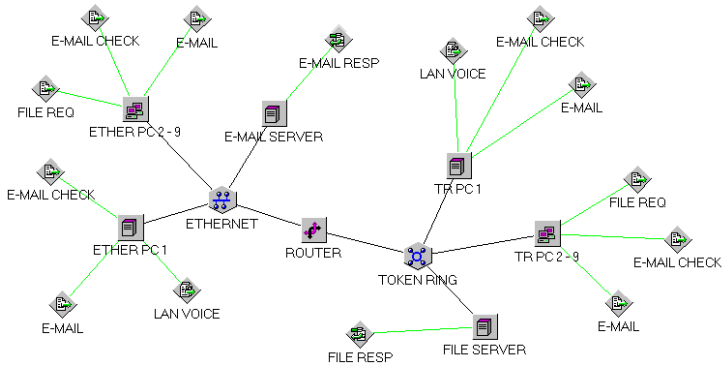


Figure 14.50. Network topology.

second file server on the LAN? What else can be modified?

Chapter notes

Law and Kelton's monography [214] provides a good overview about the network systems e.g. we definition of the networks in Section 14.1 is taken from it.

About the classification of computer networks we propose two monography, whose authors are Sima, Fountain és Kacsuk [303], and Tanenbaum [314]

Concerning the basis of probability the book of Alfréd, Rényi [286] is recommended. We have summarised the most common statistical distribution by the book of Banks et al. [30]. The review of COMNET simulation modelling tool used to depict the density functions can be found in two publications of CACI (**C**onsolidated **A**nalysis **C**enters, **I**nc.) [54, 185].

Concerning the background of mathematical simulation the monography of Ross [290], and concerning the queueing theory the book of Kleinrock [198] are useful.

The definition of channel capacity can be found in the dictionaries that are available on the Internet [172, 340]. Information and code theory related details can be found in Jones and Jones' book [184].

Taqqu and Co. [219, 317] deal with long-range dependency.

Figure 14.1 that describes the estimations of the most common distributions in network modelling is taken from the book of Banks, Carson és Nelson könyvéből [30].

The OPNET software and its documentation can be downloaded from the add-

ress found in [258]. Each phase of simulation is discussed fully in this document.

The effect of traffic burstiness is analysed on the basis of Tibor Gyires's and H. Joseph Wenn's articles [150, 151].

Leland and Co. [217, 218], Crovella and Bestavros [77] report measurements about network traffic.

The self-similarity of networks is dealt by Erramilli, Narayan and Willinger [99], Willinger and Co. [344], and Beran [38]. Mandelbrot [233], Paxson és Floyd [266], furthermore the long-range dependent processes was studied by Mandelbrot and van Ness [234].

Traffic routing models can be found in the following publications: [17, 159, 178, 240, 252, 253, 265, 344]. Neuts, Marcel F.

Figure 14.22 is from the article of Listanti, Eramo and Sabella [223] The papers [39, 92, 146, 266] contains data on traffic.

Long-range dependency was analysed by Addie, Zukerman and Neame [5], Duffield and O'Connell [91], and Narayan and Willinger [99].

The expression of *black box* modelling was introduced by Willinger and Paxson [342] in 1997.

Information about the principle of *Ockham's Razor* can be found on the web page of Francis Heylighen [164]. More information about Sniffer is on Network Associates' web site [238].

Willinger, Taqqu, Sherman and Wilson [343] analyse a structural model. Crovella and Bestavros [77] analysed the traffic of World Wide Web.

The effect of burstiness to network congestion is dealt by Neuts [252], and Molnár, Vidács, and Nilsson [247].

The pareto-model and the effect of the Hurst parameter is studied by Addie, Zukerman and Neame [5].

The Benoit-package can be downloaded from the Internet [325].

15. Parallel Computations

Parallel computating is concerned with solving a problem faster by using multiple processors in parallel. These processors may belong to a single machine, or to different machines that communicate through a network. In either case, the use of parallelism requires to split the problem into tasks that can be solved simultaneously.

In the following, we will take a brief look at the history of parallel computing, and then discuss reasons why parallel computing is harder than sequential computing. We explain differences from the related subjects of distributed and concurrent computing, and mention typical application areas. Finally, we outline the rest of this chapter.

Although the history of parallel computing can be followed back even longer, the first parallel computer is commonly said to be Illiac IV, an experimental 64-processor machine that became operational in 1972. The parallel computing area boomed in the late 80s and early 90s when several new companies were founded to build parallel machines of various types. Unfortunately, software was difficult to develop and non-portable at that time. Therefore, the machines were only adopted in the most compute-intensive areas of science and engineering, a market too small to commence for the high development costs. Thus many of the companies had to give up.

On the positive side, people soon discovered that cheap parallel computers can be built by interconnecting standard PCs and workstations. As networks became faster, these so-called *clusters* soon achieved speeds of the same order as the special-purpose machines. At present, the Top 500 list, a regularly updated survey of the most powerful computers worldwide, contains 42% clusters. Parallel computing also profits from the increasing use of multiprocessor machines which, while designed as servers for web etc., can as well be deployed in parallel computing. Finally, software portability problems have been solved by establishing widely used standards for parallel programming. The most important standards, MPI and OpenMP, will be explained in Subsections 15.3.1 and 15.3.2 of this book.

In summary, there is now an affordable hardware basis for parallel computing. Nevertheless, the area has not yet entered the mainstream, which is largely due to difficulties in developing parallel software. Whereas writing a sequential program

requires to find an algorithm, that is, a sequence of elementary operations that solves the problem, and to formulate the algorithm in a programming language, parallel computing poses additional challenges:

- Elementary operations must be grouped into tasks that can be solved concurrently.
- The tasks must be scheduled onto processors.
- Depending on the architecture, data must be distributed to memory modules.
- Processes and threads must be managed, i.e., started, stopped and so on.
- Communication and synchronisation must be organised.

Of course, it is not sufficient to find any grouping, schedule etc. that work, but it is necessary to find solutions that lead to fast programs. Performance measures and general approaches to performance optimisation will be discussed in Section 15.2, where we will also elaborate on the items above. Unlike in sequential computing, different parallel architectures and programming models favour different algorithms.

In consequence, the design of parallel algorithms is more complex than the design of sequential algorithms. To cope with this complexity, algorithm designers often use simplified models. For instance, the Parallel Random Access Machine (see Subsection 15.4.1) provides a model in which opportunities and limitations of parallelisation can be studied, but it ignores communication and synchronisation costs.

We will now contrast parallel computing with the related fields of distributed and concurrent computing. Like parallel computing, *distributed computing* uses interconnected processors and divides a problem into tasks, but the purpose of division is different. Whereas in parallel computing, tasks are executed *at the same time*, in distributed computing tasks are executed *at different locations*, using *different resources*. These goals overlap, and many applications can be classified as both parallel and distributed, but the focus is different. Parallel computing emphasises homogeneous architectures, and aims at speeding up applications, whereas distributed computing deals with heterogeneity and openness, so that applications profit from the inclusion of different kinds of resources. Parallel applications are typically standalone and predictable, whereas distributed applications consist of components that are brought together at runtime.

Concurrent computing is not bound to the existence of multiple processors, but emphasises the fact that several sub-computations are in progress at the same time. The most important issue is guaranteeing correctness for any execution order, which can be parallel or interleaved. Thus, the relation between concurrency and parallelism is comparable to the situation of reading several books at a time. Reading the books concurrently corresponds to having a bookmark in each of them and to keep track of all stories while switching between books. Reading the books in parallel, in contrast, requires to look into all books at the same time (which is probably impossible in practice). Thus, a concurrent computation may or may not be parallel, but a parallel computation is almost always concurrent. An exception is data parallelism, in which the instructions of a single program are applied to different data in parallel. This approach is followed by SIMD architectures, as described below.

For the emphasis on speed, typical application areas of parallel computing are science and engineering, especially numerical solvers and simulations. These applica-

tions tend to have high and increasing computational demands, since more computing power allows one to work with more detailed models that yield more accurate results. A second reason for using parallel machines is their higher memory capacity, due to which more data fit into a fast memory level such as cache.

The rest of this chapter is organised as follows: In Section 15.1, we give a brief overview and classification of current parallel architectures. Then, we introduce basic concepts such as task and process, and discuss performance measures and general approaches to the improvement of efficiency in Section 15.2. Next, Section 15.3 describes parallel programming models, with focus on the popular MPI and OpenMP standards. After having given this general background, the rest of the chapter delves into the subject of parallel algorithms from a more theoretical perspective. Based on example algorithms, techniques for parallel algorithm design are introduced. Unlike in sequential computing, there is no universally accepted model for parallel algorithm design and analysis, but various models are used depending on purpose. Each of the models represents a different compromise between the conflicting goals of accurately reflecting the structure of real architectures on one hand, and keeping algorithm design and analysis simple on the other. Section 15.4 gives an overview of the models, Section 15.5 introduces the basic concepts of parallel algorithmics, Sections 15.6 and 15.7 explain deterministic example algorithms for PRAM and mesh computational model.

15.1. Parallel architectures

A simple, but well-known classification of parallel architectures has been given in 1972 by Michael Flynn. He distinguishes computers into four classes: SISD, SIMD, MISD, and MIMD architectures, as follows:

- SI stands for "single instruction", that is, the machine carries out a single instruction at a time.
- MI stands for "multiple instruction", that is, different processors may carry out different instructions at a time.
- SD stands for "single data", that is, only one data item is processed at a time.
- MD stands for "multiple data", that is, multiple data items may be processed at a time.

SISD computers are von-Neumann machines. MISD computers have probably never been built. Early parallel computers were SIMD, but today most parallel computers are MIMD. Although the scheme is of limited classification power, the abbreviations are widely used.

The following more detailed classification distinguishes parallel machines into SIMD, SMP, ccNUMA, nccNUMA, NORMA, clusters, and grids.

15.1.1. SIMD architectures

As depicted in Figure 15.1, a SIMD computer is composed of a powerful control processor and several less powerful processing elements (PEs). The PEs are typically

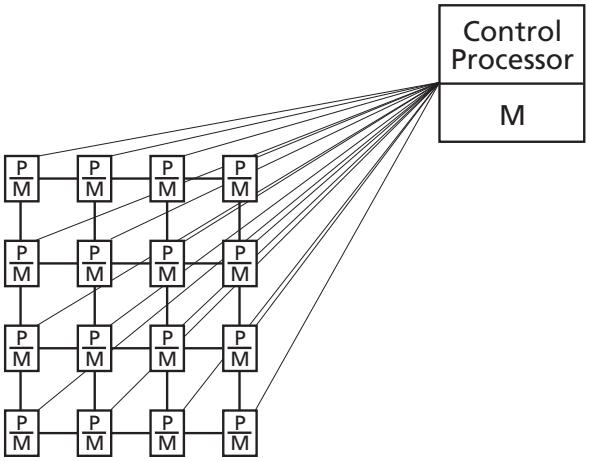


Figure 15.1. SIMD architecture.

arranged as a mesh so that each PE can communicate with its immediate neighbours. A program is a single thread of instructions. The control processor, like the processor of a sequential machine, repeatedly reads a next instruction and decodes it. If the instruction is sequential, the control processor carries out the instruction on data in its own memory. If the instruction is parallel, the control processor broadcasts the instruction to the various PEs, and these simultaneously apply the instruction to different data in their respective memories. As an example, let the instruction be LD reg, 100. Then, all processors load the contents of memory address 100 to reg, but memory address 100 is physically different for each of them. Thus, all processors carry out the same instruction, but read different values (therefore "SIMD"). For a statement of the form if *test* then *if_branch* else *else_branch*, first all processors carry out the test simultaneously, then some carry out *if_branch* while the rest sits idle, and finally the rest carries out *else_branch* while the formers sit idle. In consequence, SIMD computers are only suited for applications with a regular structure. The architectures have been important historically, but nowadays have almost disappeared.

15.1.2. Symmetric multiprocessors

Symmetric multiprocessors (SMP) contain multiple processors that are connected to a single memory. Each processor may access each memory location through standard load/store operations of the hardware. Therefore, programs, including the operating system, must only be stored once. The memory can be physically divided into modules, but the access time is the same for each pair of a processor and a memory module (therefore "symmetric"). The processors are connected to the memory by a bus (see Figure 15.2), by a crossbar, or by a network of switches. In either case, there is a delay for memory accesses which, partially due to competition for network resources, grows with the number of processors.

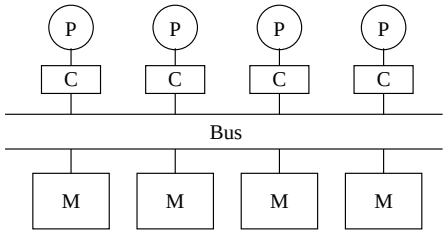


Figure 15.2. Bus-based SMP architecture.

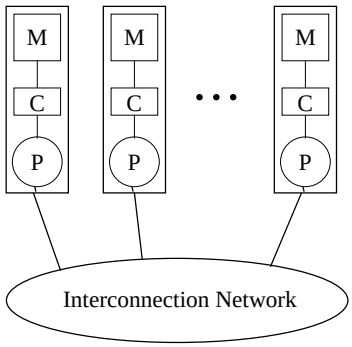


Figure 15.3. ccNUMA architecture.

In addition to main memory, each processor has one or several levels of cache with faster access. Between memory and cache, data are moved in units of cache lines. Storing a data item in multiple caches (and writing to it) gives rise to coherency problems. In particular, we speak of *false sharing* if several processors access the same cache line, but use different portions of it. Since coherency mechanisms work at the granularity of cache lines, each processor assumes that the other would have updated its data, and therefore the cache line is sent back and forth.

15.1.3. Cache-coherent NUMA architectures:

NUMA stands for **N**on-**U**uniform **M**emory **A**ccess, and contrasts with the symmetry property of the previous class. The general structure of ccNUMA architectures is depicted in Figure 15.3. As shown in the figure, each processor owns a *local memory*, which can be accessed faster than the rest called *remote memory*. All memory is accessed through standard load/store operations, and hence programs, including the operating system, must only be stored once. As in SMPs, each processor owns one or several levels of cache; cache coherency is taken care of by the hardware.

15.1.4. Non-cache-coherent NUMA architectures:

nccNUMA (**n**on **c**ache **c**oherent **N**on-**U**uniform **M**emory **A**ccess) architectures differ from ccNUMA architectures in that the hardware puts into a processor's cache only data from local memory. Access to remote memory can still be accomplished through standard load/store operations, but it is now up to the operating system to first move the corresponding page to local memory. This difference simplifies hardware design, and thus nccNUMA machines scale to higher processor numbers. On the backside, the operating system gets more complicated, and the access time to remote memory grows. The overall structure of Figure 15.3 applies to nccNUMA architectures as well.

15.1.5. No remote memory access architectures

NORMA (**NO R**emote **M**emory **A**ccess) architectures differ from the previous class in that the remote memory must be accessed through slower I/O operations as opposed to load/store operations. Each node, consisting of processor, cache and local memory, as depicted in Figure 15.3, holds an own copy of the operating system, or at least of central parts thereof. Whereas SMP, ccNUMA, and nccNUMA architectures are commonly classified as shared memory machines, SIMD architectures, NORMA architectures, clusters, and grids (see below) fall under the heading of distributed memory.

15.1.6. Clusters

According to Pfister, a cluster is a type of parallel or distributed system that consists of a collection of interconnected whole computers that are used as a single, unified computing resource. Here, the term "whole computer" denotes a PC, workstation or, increasingly important, SMP, that is, a node that consists of processor(s), memory, possibly peripheries, and operating system. The use as a single, unified computing resource is also denoted as single system image SSI. For instance, we speak of SSI if it is possible to login into the system instead of into individual nodes, or if there is a single file system. Obviously, the SSI property is gradual, and hence the borderline to distributed systems is fuzzy. The borderline to NORMA architectures is fuzzy as well, where the classification depends on the degree to which the system is designed as a whole instead of built from individual components.

Clusters can be classified according to their use for parallel computing, high throughput computing, or high availability. Parallel computing clusters can be further divided into *dedicated clusters*, which are solely built for the use as parallel machines, and *campus-wide clusters*, which are distributed systems with part-time use as a cluster. Dedicated clusters typically do not contain peripheries in their nodes, and are interconnected through a high-speed network. Nodes of campus-wide clusters, in contrast, are often desktop PCs, and the standard network is used for intra-cluster communication.

15.1.7. Grids

A grid is a hardware/software infrastructure for shared usage of resources and problem solution. Grids enable coordinated access to resources such as processors, memories, data, devices, and so on. Parallel computing is one out of several emerging application areas. Grids differ from other parallel architectures in that they are large, heterogeneous, and dynamic. Management is complicated by the fact that grids cross organisational boundaries.

15.2. Performance in practice

As explained in the introduction, parallel computing splits a problem into *tasks* that are solved independently. The tasks are implemented as either *processes* or *threads*. A detailed discussion of these concepts can be found in operating system textbooks such as Tanenbaum. Briefly stated, processes are programs in execution. For each process, information about resources such as memory segments, files, and signals is stored, whereas threads exist within processes such that multiple threads share resources. In particular, threads of a process have access to shared memory, while processes (usually) communicate through explicit message exchange. Each thread owns a separate PC and other register values, as well as a stack for local variables. Processes can be considered as units for resource usage, whereas threads are units for execution on the CPU. As less information needs to be stored, it is faster to create, destroy and switch between threads than it is for processes.

Whether threads or processes are used, depends on the architecture. On shared-memory machines, threads are usually faster, although processes may be used for program portability. On distributed memory machines, only processes are a priori available. Threads can be used if there is a software layer (distributed shared memory) that implements a shared memory abstraction, but these threads have higher communication costs.

Whereas the notion of tasks is problem-related, the notions of processes and threads refer to implementation. When designing an algorithm, one typically identifies a large number of tasks that can potentially be run in parallel, and then maps several of them onto the same process or thread.

Parallel programs can be written in two styles that can also be mixed: With ***data parallelism***, the same operation is applied to different data at a time. The operation may be a machine instruction, as in SIMD architectures, or a complex operation such as a function application. In the latter case, different processors carry out different instructions at a time. With ***task parallelism***, in contrast, the processes/threads carry out different tasks. Since a function may have an `if` or `case` statement as the outermost construct, the borderline between data parallelism and task parallelism is fuzzy.

Parallel programs that are implemented with processes can be further classified as using *Single Program Multiple Data* (SPMD) or *Multiple Program Multiple Data* (MPMD) coding styles. With SPMD, all processes run the same program, whereas with MPMD they run different programs. MPMD programs are task-parallel, whe-

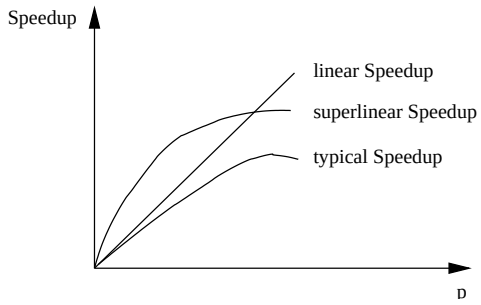


Figure 15.4. Ideal, typical, and super-linear speedup curves.

reas SPMD programs may be either task-parallel or data-parallel. In SPMD mode, task parallelism is expressed through conditional statements.

As the central goal of parallel computing is to run programs faster, performance measures play an important role in the field. An obvious measure is execution time, yet more frequently the derived measure of speedup is used. For a given problem, speedup is defined by

$$\mathit{speedup}(p) = \frac{T_1}{T_p} \ ,$$

where T_1 denotes the running time of the fastest sequential algorithm, and T_p denotes the running time of the parallel algorithm on p processors. Depending on context, speedup may alternatively refer to using p processes or threads instead of p processors. A related, but less frequently used measure is efficiency, defined by

$$\mathit{efficiency}(p) = \frac{\mathit{speedup}(p)}{p} \ .$$

Unrelated to this definition, the term efficiency is also used informally as a synonym for good performance.

Figure 15.4 shows ideal, typical, and super-linear speedup curves. The ideal curve reflects the assumption that an execution that uses twice as many processors requires half of the time. Hence, ideal speedup corresponds to an efficiency of one. Super-linear speedup may arise due to cache effects, that is, the use of multiple processors increases the total cache size, and thus more data accesses can be served from cache instead of from slower main memory.

Typical speedup stays below ideal speedup, and grows up to some number of processors. Beyond that, use of more processors slows down the program. The difference between typical and ideal speedups has several reasons:

- *Amdahl's law* states that each program contains a serial portion s that is not amenable to parallelisation. Hence, $T_p > s$, and thus $\mathit{speedup}(p) < T_1/s$, that is, the speedup is bounded from above by a constant. Fortunately, another observation, called *Gustafson-Barsis law* reduces the practical impact of Amdahl's law. It states that in typical applications, the parallel variant does not speed up a fixed problem, but runs larger instances thereof. In this case, s may grow slower

than T_1, so that T_1/s is no longer constant.

- Task management, that is, the starting, stopping, interrupting and scheduling of processes and threads, induces a certain overhead. Moreover, it is usually impossible, to evenly balance the load among the processes/threads.
- Communication and synchronisation slow down the program. Communication denotes the exchange of data, and synchronisation denotes other types of coordination such as the guarantee of mutual exclusion. Even with high-speed networks, communication and synchronisation costs are orders of magnitude higher than computation costs. Apart from physical transmission costs, this is due to protocol overhead and delays from competition for network resources.

Performance can be improved by minimising the impact of the factors listed above. Amdahl's law is hard to circumvent, except that a different algorithm with smaller s may be devised, possibly at the price of larger T_1. Algorithmic techniques will be covered in later sections; for the moment, we concentrate on the other performance factors.

As explained in the previous section, tasks are implemented as processes or threads such that a process/thread typically carries out multiple tasks. For high performance, the granularity of processes/threads should be chosen in relation to the architecture. Too many processes/threads unnecessarily increase the costs of task management, whereas too few processes/threads lead to poor machine usage. It is useful to map several processes/threads onto the same processor, since the processor can switch when it has to wait for I/O or other delays. Large-granularity processes/threads have the additional advantage of a better communication-to-computation ratio, whereas fine-granularity processes/threads are more amenable to load balancing.

Load balancing can be accomplished with static or dynamic schemes. If the running time of the tasks can be estimated in advance, static schemes are preferable. In these schemes, the programmer assigns to each process/thread some number of tasks with about the same total costs. An example of a dynamic scheme is master/slave. In this scheme, first a master process assigns one task to each slave process. Then, repeatedly, whenever a slave finishes a task, it reports to the master and is assigned a next task, until all tasks have been processed. This scheme achieves good load balancing at the price of overhead for task management.

The highest impact on performance usually comes from reducing communication/synchronisation costs. Obvious improvements result from changes in the architecture or system software, in particular from reducing ***latency***, that is, the delay for accessing a remote data item, and ***bandwidth***, that is, the amount of data that can be transferred per unit of time.

The algorithm designer or application programmer can reduce communication/synchronisation costs by minimising the number of interactions. An important approach to achieve this minimisation is locality optimisation. ***Locality***, a property of (sequential or parallel) programs, reflects the degree of temporal and spatial concentration of accesses to the same data. In distributed-memory architectures, for instance, data should be stored at the processor that uses the data. Locality can be improved by code transformations, data transformations, or a combination thereof.

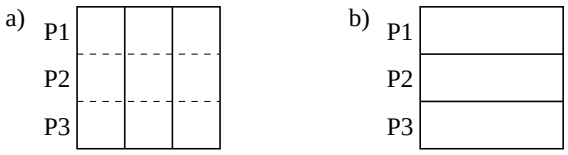


Figure 15.5. Locality optimisation by data transformation.

As an example, consider the following program fragment to be executed on three processors:

```
for (i=0; i<N; i++) in parallel
  for (j=0; j<N; j++)
    f(A[i][j]);
```

Here, the keyword "in parallel" means that the iterations are evenly distributed among the processors so that P_0 runs iterations $i = 0, \ldots, i/3$, P_1 runs iterations $i = i/3 + 1, \ldots, 2i/3$, and P_2 runs iterations $i = 2i/3 + 1, \ldots, N - 1$ (rounded if necessary). The function f is supposed to be free of side effects.

With the data distribution of Figure 15.5a), locality is poor, since many accesses refer to remote memory. Locality can be improved by changing the data distribution to that of Figure 15.5b) or, alternatively, by changing the program into

```
for (j=0; j<N; j++) in parallel
  for (i=0; i<N; i++)
    f(A[i][j]);
```

The second alternative, code transformations, has the advantage of being applicable selectively to a portion of code, whereas data transformations influence the whole program so that an improvement in one part may slow down another. Data distributions are always correct, whereas code transformations must respect *data dependencies*, which are ordering constraints between statements. For instance, in

```
a = 3;          (1)
b = a;          (2)
```

a data dependence occurs between statements (1) and (2). Exchanging the statements would lead to an incorrect program.

On shared-memory architectures, a programmer does not the specify data distribution, but locality has a high impact on performance, as well. Programs run faster if data that are used together are stored in the same cache line. On shared-memory architectures, the data layout is chosen by the compiler, e.g. row-wise in C. The programmer has only indirect influence through the manner in which he or she declares data structures.

Another opportunity to reduce communication costs is replication. For instance, it pays off to store frequently used data at multiple processors, or to repeat short computations instead of communicating the result.

Synchronisations are necessary for correctness, but they slow down program execution, first because of their own execution costs, and second because they cause processes to wait for each other. Therefore, excessive use of synchronisation should be avoided. In particular, critical sections (in which processes/threads require exclusive access to some resource) should be kept at a minimum. We speak of ***sequentialisation*** if only one process is active at a time while the others are waiting.

Finally, performance can be improved by latency hiding, that is, parallelism between computation and communication. For instance, a process can start a remote read some time before it needs the result (prefetching), or write data to remote memory in parallel to the following computations.

Exercises

15.2-1 For standard matrix multiplication, identify tasks that can be solved in parallel. Try to identify as many tasks as possible. Then, suggest different opportunities for mapping the tasks onto (a smaller number of) threads, and compare these mappings with respect to their efficiency on a shared-memory architecture.

15.2-2 Consider a parallel program that takes as input a number n and computes as output the number of primes in range $2 \leq p \leq n$. Task T_i of the program should determine whether i is a prime, by systematically trying out all potential factors, that is, dividing by $2, \ldots, \sqrt{i}$. The program is to be implemented with a fixed number of processes or threads. Suggest different opportunities for this implementation and discuss their pros and cons. Take into account both static and dynamic load balancing schemes.

15.2-3 Determine the data dependencies of the following stencil code:

```
for (t=0; t<tmax; t++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      a[i][j] += a[i-1][j] + a[i][j-1]
```

Restructure the code so that it can be parallelised.

15.2-4 Formulate and prove the bounds of the speedup known as Amdahl law and Gustafson-Barsis law. Explain the virtual contradiction between these laws. What can you say on the practical speedup?

15.3. Parallel programming

Partly due to the use of different architectures and the novelty of the field, a large number of parallel programming models has been proposed. The most popular models today are message passing as specified in the Message Passing Interface standard (MPI), and structured shared-memory programming as specified in the OpenMP standard. These programming models are discussed in Subsections 15.3.1 and 15.3.2, respectively. Other important models such as threads programming, data parallelism, and automatic parallelisation are outlined in Subsection 15.3.3.

15.3.1. MPI programming

As the name says, MPI is based on the programming model of message passing. In this model, several processes run in parallel and communicate with each other by sending and receiving messages. The processes do not have access to a shared memory, but accomplish all communication through explicit message exchange. A communication involves exactly two processes: one that executes a send operation, and another that executes a receive operation. Beyond message passing, MPI includes collective operations and other communication mechanisms.

Message passing is asymmetric in that the sender must state the identity of the receiver, whereas the receiver may either state the identity of the sender, or declare its willingness to receive data from any source. As both sender and receiver must actively take part in a communication, the programmer must plan in advance when a particular pair of processes will communicate. Messages can be exchanged for several purposes:

- exchange of data with details such as the size and types of data having been planned in advance by the programmer
- exchange of control information that concerns a subsequent message exchange, and
- synchronisation that is achieved since an incoming message informs the receiver about the sender's progress. Additionally, the sender may be informed about the receiver's progress, as will be seen later. Note that synchronisation is a special case of communication.

The MPI standard has been introduced in 1994 by the MPI forum, a group of hardware and software vendors, research laboratories, and universities. A significantly extended version, MPI-2, appeared in 1997. MPI-2 has about the same core functionality as MPI-1, but introduces additional classes of functions.

MPI describes a set of library functions with language binding to C, C++, and Fortran. With notable exceptions in MPI-2, most MPI functions deal with interprocess communication, leaving issues of process management such as facilities to start and stop processes, open. Such facilities must be added outside the standard, and are consequently not portable. For this and other reasons, MPI programs typically use a fixed set of processes that are started together at the beginning of a program run. Programs can be coded in SPMD or MPMD styles. It is possible to write parallel programs using only six base functions:

- MPI_Init must be called before any other MPI function.
- MPI_Finalize must be called after the last MPI function.
- MPI_Comm_size yields the total number of processes in the program.
- MPI_Comm_rank yields the number of the calling process, with processes being numbered starting from 0.

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main (int argc, char **argv) {
  char msg[20];
  int me, total, tag=99;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);
  MPI_Comm_size(MPI_COMM_WORLD, &total);

  if (me==0) {
    strcpy(msg, "Hello");
    MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag,
             MPI_COMM_WORLD);
  }
  else if (me==1) {
    MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD,
             &status);
    printf("Received: %s \n", msg);
  }
  MPI_Finalize();
  return 0;
}
```

Figure 15.6. A simple MPI program.

- MPI_Send sends a message. The function has the following parameters:
  - address, size, and data type of the message,
  - number of the receiver,
  - message tag, which is a number that characterises the message in a similar way like the subject characterises an email,
  - communicator, which is a group of processes as explained below.
- MPI_Recv receives a message. The function has the same parameters as MPI_Send, except that only an upper bound is required for the message size, a wildcard may be used for the sender, and an additional parameter called status returns information about the received message, e.g. sender, size, and tag.

Figure 15.6 depicts an example MPI program.

Although the above functions are sufficient to write simple programs, many more functions help to improve the efficiency and/or structure MPI programs. In particular, MPI-1 supports the following classes of functions:

- *Alternative functions for pairwise communication:* The base MPI_Send function, also called standard mode send, returns if either the message has been delivered to the receiver, or the message has been buffered by the system. This decision is left to MPI. Variants of MPI_Send enforce one of the alternatives: In synchronous mode, the send function only returns when the receiver has started receiving the message, thus synchronising in both directions. In buffered mode, the system is required to store the message if the receiver has not yet issued MPI_Recv.

  On both the sender and receiver sides, the functions for standard, synchronous, and buffered modes each come in blocking and nonblocking variants. Blocking variants have been described above. Nonblocking variants return immediately after having been called, to let the sender/receiver continue with program execution while the system accomplishes communication in the background. Nonblocking communications must be completed by a call to MPI_Wait or MPI_Test to make sure the communication has finished and the buffer may be reused. Variants of the completion functions allow to wait for multiple outstanding requests.

  MPI programs can deadlock, for instance if a process P_0 first issues a send to process P_1 and then a receive from P_1; and P_1 does the same with respect to P_0. As a possible way-out, MPI supports a combined send/receive function.

  In many programs, a pair of processes repeatedly exchanges data with the same buffers. To reduce communication overhead in these cases, a kind of address labels can be used, called persistent communication. Finally, MPI functions MPI_Probe and MPI_Iprobe allow to first inspect the size and other characteristics of a message before receiving it.

- *Functions for Datatype Handling:* In simple forms of message passing, an array of equally-typed data (e.g. float) is exchanged. Beyond that, MPI allows to combine data of different types in a single message, and to send data from non-contiguous buffers such as every second element of an array. For these purposes, MPI defines two alternative classes of functions: user-defined data types describe a pattern of data positions/types, whereas packaging functions help to put several data into a single buffer. MPI supports heterogeneity by automatically converting data if necessary.

- *Collective communication functions:* These functions support frequent patterns of communication such as broadcast (one process sends a data item to all other processes). Although any pattern can be implemented by a sequence of sends/receives, collective functions should be preferred since they improve program compactness/understandability, and often have an optimised implementation. Moreover, implementations can exploit specifics of an architecture, and so a program that is ported to another machine may run efficiently on the new machine as well, by using the optimised implementation of that machine.

- *Group and communicator management functions:* As mentioned above, the send and receive functions contain a communicator argument that describes a group of processes. Technically, a communicator is a distributed data structure that tells each process how to reach the other processes of its group, and contains additional information called attributes. The same group may be described

by different communicators. A message exchange only takes place if the communicator arguments of MPI_Send and MPI_Recv match. Hence, the use of communicators partitions the messages of a program into disjoint sets that do not influence each other. This way, communicators help structuring programs, and contribute to correctness. For libraries that are implemented with MPI, communicators allow to separate library traffic from traffic of the application program. Groups/communicators are necessary to express collective communications. The attributes in the data structure may contain application-specific information such as an error handler. In addition to the (intra)communicators described so far, MPI supports intercommunicators for communication between different process groups.

MPI-2 adds four major groups of functions:

- *Dynamic process management functions:* With these functions, new MPI processes can be started during a program run. Additionally, independently started MPI programs (each consisting of multiple processes) can get into contact with each other through a client/server mechanism.
- *One-sided communication functions:* One-sided communication is a type of shared-memory communication in which a group of processes agrees to use part of their private address spaces as a common resource. Communication is accomplished by writing into and reading from that shared memory. One-sided communication differs from other shared-memory programming models such as OpenMP in that explicit function calls are required for the memory access.
- *Parallel I/O functions:* A large set of functions allows multiple processes to collectively read from or write to the same file.
- *Collective communication functions for intercommunicators:* These functions generalise the concept of collective communication to intercommunicators. For instance, a process of one group may broadcast a message to all processes of another group.

15.3.2. OpenMP programming

OpenMP derives its name from being an open standard for multiprocessing, that is for architectures with a shared memory. Because of the shared memory, we speak of threads (as opposed to processes) in this section.

Shared-memory communication is fundamentally different from message passing: Whereas message passing immediately involves two processes, shared-memory communication uncouples the processes by inserting a medium in-between. We speak of read/write instead of send/receive, that is, a thread writes into memory, and another thread later reads from it. The threads need not know each other, and a written value may be read by several threads. Reading and writing may be separated by an arbitrary amount of time. Unlike in message passing, synchronisation must be organised explicitly, to let a reader know when the writing has finished, and to avoid concurrent manipulation of the same data by different threads.

OpenMP is one type of shared-memory programming, while others include one-sided communication as outlined in Subsection 15.3.1, and threads programming as

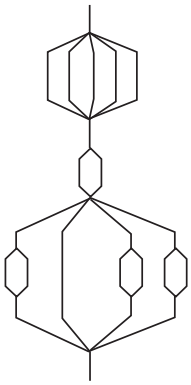


Figure 15.7. Structure of an OpenMP program.

outlined in Subsection 15.3.3. OpenMP differs from other models in that it enforces a fork-join structure, which is depicted in Figure 15.7. A program starts execution as a single thread, called master thread, and later creates a team of threads in a so-called parallel region. The master thread is part of the team. Parallel regions may be nested, but the threads of a team must finish together. As shown in the figure, a program may contain several parallel regions in sequence, with possibly different numbers of threads.

As another characteristic, OpenMP uses compiler directives as opposed to library functions. Compiler directives are hints that a compiler may or may not take into account. In particular, a sequential compiler ignores the directives. OpenMP supports incremental parallelisation, in which one starts from a sequential program, inserts directives at the most performance-critical sections of code, later inserts more directives if necessary, and so on.

OpenMP has been introduced in 1998, version 2.0 appeared in 2002. In addition to compiler directives, OpenMP uses a few library functions and environment variables. The standard is available for C, C++, and Fortran.

Programming OpenMP is easier than programming MPI since the compiler does part of the work. An OpenMP programmer chooses the number of threads, and then specifies work sharing in one of the following ways:

- *Explicitly:* A thread can request its own number by calling the library function omp_get_thread_num. Then, a conditional statement evaluating this number explicitly assigns tasks to the threads, similar as in SPMD-style MPI programs.

- *Parallel loop:* The compiler directive #pragma omp parallel for indicates that the following for loop may be executed in parallel so that each thread carries out several iterations (tasks). An example is given in Figure 15.8. The programmer can influence the work sharing by specifying parameters such as schedule(static) or schedule(dynamic). Static scheduling means that each thread gets an about equal-sized block of consecutive iterations. Dynamic scheduling means that first each thread is assigned one iteration, and then, repeatedly, a thread that has

```
#include <omp.h>
#define N 100
double a[N][N], b[N], c[N];
int main() {
  int i, j;
  double h;
  /* initialisation omitted */
  omp_set_num_threads(4);
  #pragma omp parallel for shared(a,b,c) private(j)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      c[i] += a[i][j] * b[j];
  /* output omitted */
}
```

Figure 15.8. Matrix-vector multiply in OpenMP using a parallel loop.

finished an iteration gets the next one, as in the master/slave paradigma described before for MPI. Different from master/slave, the compiler decides which thread carries out which tasks, and inserts the necessary communications.

- *Task-parallel sections:* The directive #pragma omp parallel sections allows to specify a list of tasks that are assigned to the available threads.

Threads communicate through shared memory, that is, they write to or read from shared variables. Only part of the variables are shared, while others are private to a particular thread. Whether a variable is private or shared is determined by rules that the programmer can overwrite.

Many OpenMP directives deal with synchronisation that is necessary for mutual exclusion, and to provide a consistent view of shared memory. Some synchronisations are inserted implicitly by the compiler. For instance, at the end of a parallel loop all threads wait for each other before proceeding with a next loop.

15.3.3. Other programming models

While MPI and OpenMP are the most popular models, other approaches have practical importance as well. Here, we outline threads programming, High Performance Fortran, and automatic parallelisation.

Like OpenMP, *threads programming* or by Java threads uses shared memory. Threads operate on a lower abstraction level than OpenMP in that the programmer is responsible for all details of thread management and work sharing. In particular, threads are created explicitly, one at a time, and each thread is assigned a function to be carried out. Threads programming focuses on task parallelism, whereas OpenMP programming focuses on data parallelism. Thread programs may be unstructured, that is, any thread may create and stop any other. OpenMP programs are often compiled into thread programs.

Data parallelism provides for a different programming style that is explicitly supported by languages such as *High Performance Fortran* (HPF). While data pa-

rallelism can be expressed in MPI, OpenMP etc., data-parallel languages center on the approach. As one of its major constructs, HPF has a parallel loop whose iterations are carried out independently, that is, without communication. The data-parallel style makes programs easier to understand since there is no need to take care of concurrent activities. On the backside, it may be difficult to force applications into this structure. HPF is targeted at single address space distributed memory architectures, and much of the language deals with expressing data distributions. Whereas MPI programmers distribute data by explicitly sending them to the right place, HPF programmers specify the data distribution on a similar level of abstraction as OpenMP programmers specify the scheduling of parallel loops. Details are left to the compiler. An important concept of OpenMP is the owner-computes rule, according to which the owner of the left-hand side variable of an assignment carries out an operation. Thus, data distribution implies the distribution of computations.

Especially for programs from scientific computing, a significant performance potential comes from parallelising loops. This parallelisation can often be accomplished automatically, by *parallelising compilers*. In particular, these compilers check for data dependencies. that prevent parallelisation. Many programs can be restructured to circumvent the dependence, for instance by exchanging outer and inner loops. Parallelising compilers find these restructuring for important classes of programs.

Exercises

15.3-1 Sketch an MPI program for the prime number problem of Exercise 15.2-3. The program should deploy the master/slave paradigma. Does your program use SPMD style or MPMD style?

15.3-2 Modify your program from Exercise 15.3-1 so that it uses collective communication.

15.3-3 Compare MPI and OpenMP with respect to programmability, that is, give arguments why or to which extent it is easier to program in either MPI or OpenMP.

15.3-4 Sketch an OpenMP program that implements the stencil code example of Exercise 15.2-3.

15.4. Computational models

15.4.1. PRAM

The most popular computational model is the **P**arallel **R**andom **A**ccess **M**achine (PRAM) which is a natural generalisation of the **R**andom **A**ccess **M**achine (RAM).

The PRAM model consists of p synchronised processors $P_1, P_2, \ldots, P_p$, a shared memory with memory cells $M[1]$, $M[2]$, $\ldots$, $M[m]$ and memories of the processors. Figure 15.9. shows processors and the shared random access memory

There are variants of this model. They differ in whether multiple processors are allowed to access the same memory cell in a step, and in how the resulting conflicts are resolved. In particular the following variants are distinguished:

Types on the base of the properties of read/write operations are

- EREW (Exclusive-Read Exclusive-Write) PRAM,

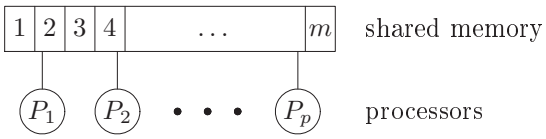


Figure 15.9. Parallel random access machine.

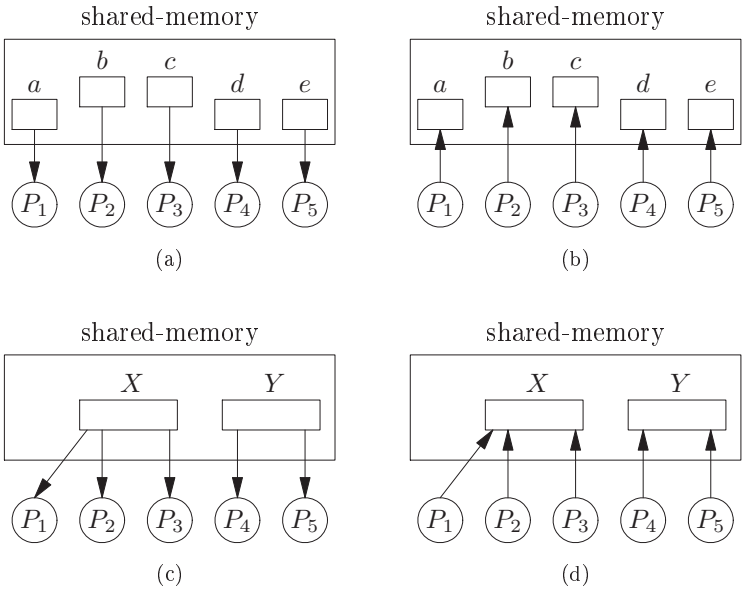


Figure 15.10. Types of parallel random access machines.

- ERCW (Exclusive-Read Concurrent-Write) PRAM,
- CREW (Concurrent-Read Exclusive-Write) PRAM,
- CRCW (Concurrent-Read Concurrent-Write) PRAM.

Figure 15.10(a) shows the case when at most one processor has access a memory cell (ER), and Figure 15.10(d) shows, when multiple processors have access the same cell (CW).

Types of concurrent writing are *common, priority, arbitrary, combined.*

15.4.2. BSP, LogP and QSM

Here we consider the models BSP, LogP and QSM.

Bulk-synchronous Parallel Model (BSP) describes a computer as a collection of nodes, each consisting of a processor and memory. BSP supposes the existence of a router and a barrier synchronisation facility. The router transfers messages between the nodes, the barrier synchronises all or a subset of nodes. According to BSP compu-

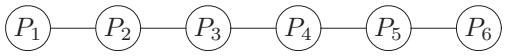


Figure 15.11. A chain consisting of six processors.

tation is partitioned into ***supersteps***. In a superstep each processor independently performs computations on data in its own memory, and initiates communications with other processors. The communication is guaranteed to complete until the beginning of the next superstep.

g is defined such that gh is the time that is takes to route an h-relation under continuous traffic conditions. An h-relation is a communication pattern in which each processor sends and receives up to h messages.

The cost of a superstep is determined as $x + gh + l$, where x is the maximum number of communications initiated by any processor. The cost of a program is the sum of the costs of the individual supersteps.

BSP contains a cost model that involves three parameters: the number of processors (p), the cost of a barrier synchronisation (l) and a characteristics of the available bandwidth (g).

LogP model was motivated by inaccuracies of BSP and the restrictive requirement to follow the superstep structure.

While LogP improves on BSP with respect to reflectivity, QSM improves on it with respect to simplicity. In contrast to BSP, QSM is a shared-memory model. As in BSP, the computation is structured into supersteps, and each processor has its own local memory. In a superstep, a processor performs computations on values in the local memory, and initiates read/write operations to the shared memory. All shared-memory accesses complete until the beginning of the next superstep. QSM allows for concurrent reads and writes. Let the maximum number of accesses to any cell in a superstep be k. Then QSM charges costs $\max(x, gh, k)$, with x, g, and h being defined in BSP.

15.4.3. Mesh, hypercube and butterfly

Mesh also is a popular computational model. A d-dimensional mesh is an $a_1 \times a_2 \times \cdots \times a_d$ sized grid having a processor in each grid point. The edges are the communication lines, working in two directions. Processors are labelled by d-tuples, as $P_{i_1,i_2,\ldots,i_d}$.

Each processor is a RAM, having a local memory. The local memory of the processor $P_{i_1,i_2,\ldots,i_d}$ is $M[i_1,\ldots,i_d,1],\ldots,M[i_1,\ldots,i_d,m]$. Each processor can execute in one step such basic operations as adding, subtraction, multiplication, division, comparison, read and write from/into the local memory, etc. Processors work in synchronised way, according to a global clock.

The simplest mesh is the ***chain***, belonging to the value $d = 1$. Figure 15.11 shows a chain consisting of 6 processors.

The processors of a chain are $P_1,\ldots,P_p$. P_1 is connected with P_{p-1}, P_p is connected with P_{p-1}, the remaining processors P_i are connected with P_{i-1} and P_{i+1}.

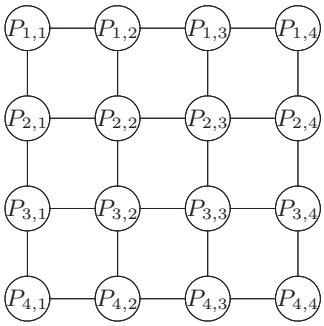


Figure 15.12. A square of size 4×4.

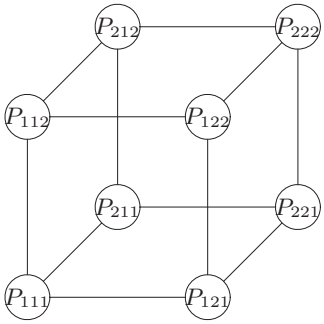


Figure 15.13. A 3-dimensional cube of size $2 \times 2 \times 2$.

If $d = 2$, then we get a rectangle. If now $a_1 = a_2 = \sqrt{p}$, then we get a ***square***. Figure 15.12 shows a square of size 4×4.

A square contains several chains consisting of a processors. The processors having identical first index, form a ***row of processors***, and the processors having the same second index form a ***column of processors***. Algorithms running on a square often consists of such operations, executed only by processors of some rows or columns.

If $d = 3$, then the corresponding mesh is a brick. In the special case $a_1 = a_2 = a_3 = \sqrt[3]{p}$ the mesh is called ***cube***. Figure 15.13 shows a cube of size $2 \times 2 \times 2$.

The next model of computation is the ***d-dimensional hypercube*** $\mathcal{H}_d$. This model can be considered as the generalisation of the square and cube: the square represented on Figure 15.12 is a 2-dimensional, and the cube, represented on Figure 15.13 is a 3-dimensional hypercube. The processors of $\mathcal{H}_d$ can be labelled by a binary number consisting of d bits. Two processors of $\mathcal{H}_d$ are connected iff the Hamming-distance of their labels equals to 1. Therefore each processors of $\mathcal{H}_d$ has d neighbours, and the of $\mathcal{H}_d$ is d. Figure 15.14 represents $\mathcal{H}_4$.

The butterfly model $\mathcal{B}_d$ consists of $p = (d+1)2^d$ processors and $2d^{d+1}$ edges. The processors can be labelled by a pair $\langle r, l \rangle$, where r is the ***columnindex*** and l is

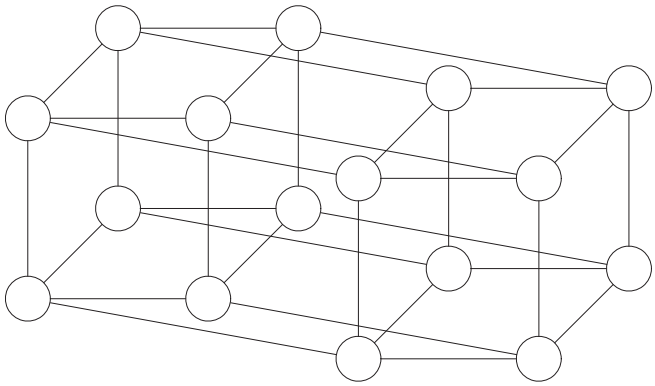


Figure 15.14. A 4-dimensional hypercube $\mathcal{H}_4$.

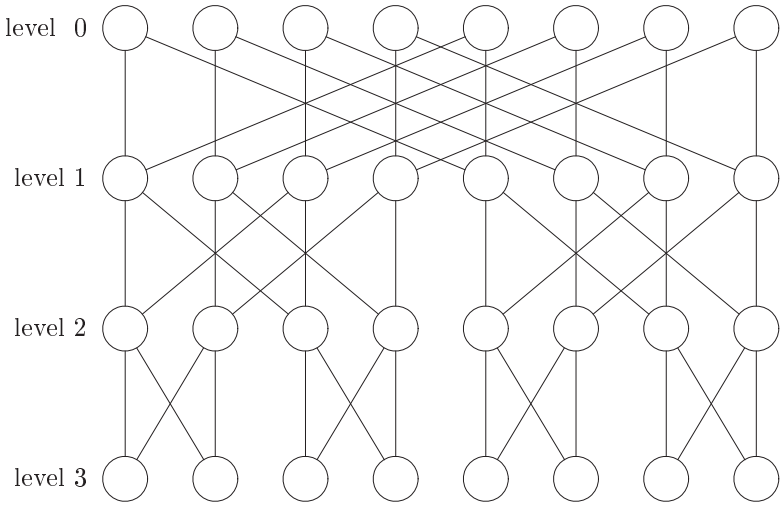


Figure 15.15. A butterfly model.

the ***level*** of the given processor. Figure 15.15 shows a ***butterfly*** model $\mathcal{B}_3$ containing 32 processors in 8 columns and in 4 levels.

Finally Figure 15.16 shows a ***ring*** containing 6 processors.

15.5. Performance in theory

In the previous section we considered the performance measures used in the practice.

In the theoretical investigations the algorithms are tested using abstract computers called computation models.

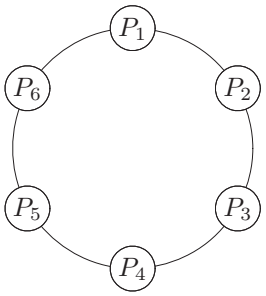


Figure 15.16. A ring consisting of 6 processors.

The required quantity of resources can be characterised using *absolute* and *relative* measures.

Let $W(n, \pi, \mathrm{A})$, resp. $W(n, \pi, p, \mathrm{P})$ denote the time necessary in *worst case* to solve the problem π of size n by the sequential algorithm A, resp. parallel algorithm P (using p processors).

In a similar way let $B(n, \pi, \mathrm{A})$, resp. $B(n, \pi, p, \mathrm{P})$ the time necessary for algorithm A, resp. P in *best case* to solve the problem π of size n (algorithm P can use p processors).

Let $N(n, \pi)$, resp. $N(n, \pi, p)$ the time needed by any sequential, resp. parallel algorithm to solve problem π of size n (algorithm P can use p processors). These times represent a ***lower bound*** of the corresponding running time.

Let suppose the distribution function $D(n, \pi)$ of the problem π of size n is given. Then let $E(n, \pi, \mathrm{A})$, resp. $E(n, \pi, p, \mathrm{P})$ the expected value of the time necessary for algorithm A, resp. P to solve problem π of size n (algorithm P uses p processors).

In the analysis it is often supposed that the input data of equal size have equal probability. For such cases we use the notation $A(n, \mathrm{A})$, resp. $A(n, \mathrm{P}, p)$ and termin ***average running time***.

The value of the performance measures W, B, N, E and A depend on the used computation model too. For the simplicity of notations we suppose that the algorithms determine the computation model.

Usually the context shows in a unique way the investigated problem. If so, then the parameter π is omitted.

Among these performance measures hold the following inequalities:

$$\begin{aligned} N(n) &\leq B(n, \mathrm{A}) && (15.1)\\ &\leq E(n, \mathrm{A}) && (15.2)\\ &\leq W(n, \mathrm{A})\ . && (15.3) \end{aligned}$$

In a similar way for the characteristic data of the parallel algorithms the following inequalities are true:

$$\begin{aligned} N(n,p) &\leq B(n,\mathrm{P},p) && (15.4)\\ &\leq E(n,\mathrm{P},p) && (15.5)\\ &\leq W(n,\mathrm{P},p)\ . && (15.6) \end{aligned}$$

For the expected running time we have

$$\begin{aligned} B(n,\mathrm{A}) &\leq A(n,\mathrm{A}) && (15.7)\\ &\leq W(n,\mathrm{A})\ , && (15.8) \end{aligned}$$

and

$$\begin{aligned} B(n,\mathrm{P},p) &\leq A(n,\mathrm{P},p) && (15.9)\\ &\leq W(n,\mathrm{P},p)\ . && (15.10) \end{aligned}$$

These notations can be used not only for the running time, but also for any other resource, as memory requirement, number of messages, etc.

Now we define some ***relative performance measures***.

Speedup shows, how many times is smaller the running time of a parallel algorithm, than the running time of the parallel algorithm solving the same problem.

The ***speedup*** (or *relative number of steps* or *relative speed*) of a given parallel algorithm P, comparing it with a given sequential algorithm A, is defined as

$$g(n,\mathrm{A},\mathrm{P}) = \frac{W(n,\mathrm{A})}{W(n,\mathrm{P},p)}\ . \tag{15.11}$$

If for a sequential algorithm A and a parallel algorithm P holds

$$\frac{W(n,\mathrm{A})}{W(n,p,\mathrm{P})} = \Theta(p)\ , \tag{15.12}$$

then the speedup of P comparing with A is ***linear***, if

$$\frac{W(n,\mathrm{A})}{W(n,\mathrm{P},p)} = o(p)\ , \tag{15.13}$$

then the speedup of P comparing with A is ***sublinear***, and if

$$\frac{W(n,\mathrm{A})}{W(n,\mathrm{P},p)} = \omega(p)\ , \tag{15.14}$$

then the speedup of P comparing with A is ***superlinear***.

In the case of parallel algorithms it is a very important performance measure the ***work*** $w(n,p,\mathrm{P})$, defined by the product of the running time and the number of the used processors:

$$w(n,p,\mathrm{P}) = pW(n,\mathrm{P},p)\ . \tag{15.15}$$

This definition is used even then if some processors work only in a small fraction of the running time. Therefore the real work can be much smaller, then given by the formula 15.15).

The ***efficiency*** $h(n, p, \mathrm{P}, \mathrm{A})$ is a measure of the fraction of time for which the processors are usefully employed; it is defined as the ratio of the work of the sequential algorithm to the work of the parallel algorithm P:

$$e(n, p, \mathrm{P}, \mathrm{A}) = \frac{W(n, A)}{pW(n, \mathrm{P}, p)} \ . \tag{15.16}$$

One can observe, that the ratio of the speedup and the number of the used parallel processors results the same value. If the parallel work is not less than the sequential one, then efficiency is between zero and one, and the relatively large values are beneficial.

In connection with the analysis of the parallel algorithms the work-efficiency is a central concept. If for a parallel algorithm P and sequential algorithm A holds

$$pW(n, \mathrm{P}, p) = O(W(n, \mathrm{A})) \ , \tag{15.17}$$

then algorithm P ***work-optimal*** comparing with A.

This definition is equivalent with the equality

$$\frac{pW(n, \mathrm{P}, p)}{W(n, \mathrm{A})} = O(1). \tag{15.18}$$

According to this definition a parallel algorithm is work-optimal only if the order of its total work is not greater, than the order of the total work of the considered sequential algorithm.

A weaker requirement is the following. If there exists a finite positive integer k such that

$$pW(n, \mathrm{P}, p) = O(W(n, \mathrm{A}(\lg n)^k) \ , \tag{15.19}$$

then algorithm P is ***work-efficient*** comparing with A.

If a sequential algorithm A, resp. a parallel algorithm P uses only $O(N(n))$, resp. $O(N(n, p))$ units of a given resource, then A, resp. P is called—for the given resource and the considered model of computation—***asymptotically optimal.***

If an A sequential or a P parallel algorithm uses only the necessary amount of some resource for all possible size $n \geq 1$ of the input, that is $N(n, \mathrm{A})$, resp. $N(n, p, \mathrm{A})$ units, and so we have

$$W(n, \mathrm{A}) = N(n, \mathrm{A}) \ , \tag{15.20}$$

for A and

$$W(n, \mathrm{P}, p) = N(n, \mathrm{P}, p) \ , \tag{15.21}$$

for P, then we say, that the given algorithm is ***absolute optimal*** for the given resource and the given computation model. In this case we say, that $W(n, \mathrm{P}, p) = N(n, \mathrm{P}, p)$ is the ***accurate complexity*** of the given problem.

Comparing two algorithms and having

$$W(n, \mathrm{A}) = \Theta(W(n, \mathrm{B})) \tag{15.22}$$

we say, that the speeds of the growths of algorithms A and B ***asymptotically have the same order***.

Comparing the running times of two algorithms A and B (e.g. in worst case) sometime the estimation depends on n: for some values of n algorithm A, while for other values of n algorithm B is the better. A possible formal definition is as follows. If the functions $f(n)$ and $g(n)$ are defined for all positive integer n, and for some positive integer v hold

1. $f(v) = g(v)$;
2. $(f(v-1) - g(v-1))(f(v+1) - g(v+1)) < 0$,

then the number v is called ***crossover point*** of the functions $f(n)$ and $g(n)$.

For example multiplying two matrices according to the definition and algorithm of Strassen we get one crossover point, whose value is about 20.

Exercises

15.5-1 Suppose that the parallel algorithms P and Q solve the selection problem. Algorithm P uses $n^{0.5}$ processors and its running time is $W(n, \mathrm{P}, p) = \Theta(n^{0.5})$. Algorithm Q uses n processors and its running time is $W(n, \mathrm{P}, p) = \Theta(\lg n)$. Determine the work, speedup and efficiency for both algorithms. Are these algorithms work-optimal or at least work-efficient?

15.5-2 Analyse the following two assertions.

a) Running time of algorithm P is at least $O(n^2)$.

b) Since the running time of algorithm P is $O(n^2)$, and the running time of algorithm B is $O(n \lg n)$, therefore algorithm B is more efficient.

15.5-3 Extend the definition of the crossover point to noninteger v values and parallel algorithms.

15.6. PRAM algorithms

In this section we consider parallel algorithms solving simple problems as prefix calculation, ranking of the elements of an array, merging, selection and sorting.

In the analysis of the algorithms we try to give the accurate order of the running time in the worst case and try to decide whether the presented algorithm is work-optimal or at least work-efficient or not. When parallel algorithms are compared with sequential algorithms, always the best known sequential algorithm is chosen.

To describe these algorithms we use the following pseudocode conventions.

```
P_i  in   parallel for i ← 1 to p
     do   ⟨ command 1 ⟩
          ⟨ command 2 ⟩
          .
          .
          .
          ⟨ command u ⟩
```

For m^2 PRAM ordered into a square grid of size $m \times m$ the instruction begin with

$P_{i,j}$ **in parallel for** $i \leftarrow 1$ **to** m, $j \leftarrow 1$ **to** m
do

For a k-dimensional mesh of size $m_1 \times \cdots m_k$ the similar instruction begins with

$P_{i_1,i_2,\ldots,i_k}$ **in parallel for** $i_1 \leftarrow 1$ **to** m_1, $\ldots$, $i_k \leftarrow 1$ **to** m_k
do

It is allowed that in this commands P_i represents a group of processors.

15.6.1. Prefix

Let $\oplus$ be a binary associative operator defined over a set Σ. We suppose that the operator needs only one set and the set is closed for this operation.

A binary operation $\oplus$ is *associative* on a Σ set, if for all $x, y, z \in \Sigma$ holds

$$((x \oplus y) \oplus z) = (x \oplus (y \oplus z)) \; . \tag{15.23}$$

Let the elements of the sequence $X = x_1, x_2, \ldots, x_p$ be elements of the set Σ. Then the input data are the elements of the sequence X, and the ***prefix problem*** is the computation of the elements x_1, $x_1 \oplus x_2$, $\ldots, x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_p$. These elements are called ***prefixes***.

It is worth to remark that in other topics of parallel computations the starting sequences $x_1, x_2, \ldots, x_k$ of the sequence X are called prefixes.

Example 15.1 *Associative operations.* If Σ is the set of integer numbers, $\oplus$ means addition and the sequence of the input data is $X = 3, -5, 8, 2, 5, 4$, then the sequence of the prefixes is $Y = 3, -2, 6, 8, 13, 17$. If the alphabet and the input data are the same, but the operation is the multiplication, then $Y = 3, -15, -120, -240, -1200, -4800$. If the operation is the minimum (it is also an associative operation), then $Y = 3, -5, -5, -5, -5, -5$. In this case the last prefix is the minimum of the input data.

The prefix problem can be solved by sequential algorithms in $O(p)$ time. Any sequential algorithm A requires $\Omega(p)$ time to solve the prefix problem. There are parallel algorithms for different models of computation resulting a work-optimal solution of the prefix problem.

In this subsection at first the algorithm CREW-PREFIX is introduced, which solves the prefix problem in $\Theta(\lg p)$ time, using p CREW PRAM processors.

Next is algorithm EREW-PREFIX, having similar quantitative characteristics, but requiring only EREW PRAM processors.

These algorithms solve the prefix problem quicker, then the sequential algorithms, but the order of the necessary work is larger.

Therefore interesting is algorithm OPTIMAL-PREFIX, which uses only $\lceil p/\lg p \rceil$ CREW PRAM processors, and makes only $\Theta(\lg p)$ steps. The work of this algorithm is only $\Theta(p)$, therefore its efficiency is $\Theta(1)$, and so it is work-optimal. The speedup of this algorithm equals to $\Theta(n/\lg n)$.

For the sake of simplicity in the further we write usually $p/\lg p$ instead of $\lceil p/\lg p \rceil$.

A CREW PRAM algorithm

As first parallel algorithm a recursive algorithm is presented, which runs on CREW PRAM model of computation, uses p processors and $\Theta(\lg p)$ time. Designing parallel algorithm it is often used the principle *divide-and-conquer*, as we we will see in the case of the next algorithm too

Input is the number of processors (p) and the array $X[1\,..\,p]$, output data are the array $Y[1\,..\,p]$. We suppose p is a power of 2. Since we use the algorithms always with the same number of processors, therefore we omit the number of processors from the list of input parameters. In the mathematical descriptions we prefer to consider X and Y as sequences, while in the pseudocodes sometimes as arrays.

CREW-PREFIX(X)

1 **if** $p = 1$
2 **then** $y_1 \leftarrow x_1$
3 **return** Y
4 **if** $p > 1$
5 **then** P_i **in parallel for** $i \leftarrow 1$ **to** $p/2$
do compute recursive $y_1, y_2, \ldots, y_{p/2}$,
the prefixes, belonging to $x_1, x_2, \ldots, x_{p/2}$
P_i **in parallel for** $i \leftarrow p/2 + 1$ **to** p
do compute recursive $y_{p/2+1}, y_{p/2+2}, \ldots, y_p$
the prefixes, belonging to $x_{p/2+1}, x_{p/2+2}, \ldots, x_p$
6 P_i **in parallel for** $p/2 + 1 \leq i \leq p$
do read $y_{p/2}$ from the global memory and compute $y_{p/2} \oplus y_{p/2+i}$
7 **return** Y

Example 15.2 *Calculation of prefixes of 8 elements on 8 processors.* Let $n = 8$ and $p = 8$. The input data of the prefix calculation are 12, 3, 6, 8, 11, 4, 5 and 7, the associative operation is the addition.

The run of the *recursive algorithm* consists of *rounds*. In the first round (step 4) the first four processors get the input data 12, 3, 6, 8, and compute recursively the prefixes 12, 15, 21, 29 as output. At the same time the other four processors get the input data 11, 4, 5, 7, and compute the prefixes 11, 15, 20, 27.

According to the recursive structure $P_1, P_2,\ P_3$ and P_4 work as follows. P_1 and P_2 get x_1 and x_2, resp. P_3 and P_4 get x_3 and x_4 as input. Recursivity mean for P_1 and P_2, that P_1 gets x_1 and P_2 gets x_2, computing at first $y_1 = x_1$ and $y_2 = x_2$, then P_2 updates $y_2 = y_1 \oplus y_2$. After this P_3 computes $y_3 = y_2 \oplus y_3$ and $y_4 = y_4 \oplus y_4$.

While $P_1, P_2,\ P_3$ and P_4, according to step 4, compute the final values $y_1,\ y_2,\ y_3$ and y_4, $P_5, P_6,\ P_7$ and P_8 compute the local provisional values of $y_5,\ y_6,\ y_7$ and y_8.

In the second round (step 5) the first four processors stay, the second four processors compute the final values of $y_5,\ y_6,\ y_7$ and y_8, adding $y_4 = 29$ to the provisional values 11, 15, 20 and 27 and receiving 40, 44, 49 and 56.

In the remaining part of the section we use the notation $W(n)$ instead of $W(n,p)$ and give the number of used processors in verbal form. If $p = n$, then we usually prefer to use p.

Theorem 15.1 *Algorithm* CREW-PREFIX *uses* $\Theta(\lg p)$ *time on* p CREW PRAM *processors to compute the prefixes of* p *elements.*

Proof The lines 4–6 require $W(p/2)$ steps, the line 7 does $\Theta(1)$ steps. So we get the following recurrence:

$$W(p) = W(p/2) + \Theta(1). \tag{15.24}$$

Solution of this recursive equation is $W(p) = \Theta(\lg p)$. ∎

CREW-PREFIX is not work-optimal, since its work is $\Theta(p \lg p)$ and we know sequential algorithm requiring only $O(p)$ time, but it is work-effective, since all sequential prefix algorithms require $\Omega(p)$ time.

An EREW PRAM algorithm

In the following algorithm we use exclusive write instead of the parallel one, therefore it can be implemented on the EREW PRAM model. Its input is the number of processors p and the sequence $X = x_1,\ x_2,\ \ldots,\ x_p$, and its output is the sequence $Y = y_1,\ y_2,\ \ldots,\ y_p$ containing the prefixes.

EREW-PREFIX(X)

```
1  Y[1] ← X[1]
2  P_i in parallel for i ← 2 to p
3     do Y[i] ← X[i − 1] ⊕ X[i]
4  k ← 2
5  while k < p
6        do P_i in parallel for i ← k + 1 to p
7           do Y[i] ← Y[i − k] ⊕ Y[i]
8              k ← k + k
9  return Y
```

Theorem 15.2 *Algorithm* EREW-PREFIX *computes the prefixes of* p *elements on* p EREW PRAM *processors in* $\Theta(\lg p)$ *time.*

Proof The commands in lines 1–3 and 9 are executed in $O(1)$ time. Lines 4–7 are executed so many times as the assignment in line 8, that is $\Theta(p)$ times. ∎

A work-optimal algorithm

Next we consider a recursive work-optimal algorithm, which uses $p/\lg p$ CREW PRAM processors. Input is the length of the input sequence (p) and the sequence $X = x_1,\ x_2,\ \ldots,\ x_p$, output is the sequence $Y = y_1,\ y_2,\ \ldots,\ y_p$, containing the computed prefixes.

OPTIMAL-PREFIX(p, X)

1 P_i **in parallel for** $i \leftarrow 1$ **to** $p/\lg p$
2 **do** compute recursive $z_{(i-1)\lg p+1}, z_{(i-1)\lg p+2}, \ldots, z_{i\lg p}$,
the prefixes of the following $\lg p$ input data
$x_{(i-1)\lg p+1}, x_{(i-1)\lg p+2}, \ldots, x_{i\lg p}$
3 P_i **in parallel for** $i \leftarrow 1$ **to** $p/\lg p$
4 **do** using CREW-PREFIX compute $w_{\lg p}, w_{2\lg p}, w_{3\lg p}, \ldots, w_p$,
the prefixes of the following $p/\lg p$ elements:
$z_{\lg p}, z_{2\lg p}, z_{3\lg p}, \ldots, , z_p$
5 P_i **in parallel for** $i \leftarrow 2$ **to** $p/\lg p$
6 **do for** $j \leftarrow 1$ **to** p
7 **do** $Y[(i-1)\lg p + j] \leftarrow w_{(i-1)\lg p} \oplus z_{(i-1)\lg p+j}$
8 P_1 **for** $j \leftarrow 1$ **to** p
9 **do** $Y[j] \leftarrow z_j$
10 **return** Y

This algorithm runs in logarithmic time. The following two formulas help to show it:

$$z_{(i-1)\lg p+k} = \sum_{j=(i-1)\lg p+1}^{i\lg p} x_j \quad (k = 1,\ 2,\ \ldots,\ \lg p) \tag{15.25}$$

and

$$w_{i\lg p} = \sum_{j=1}^{i} z_{j\lg p}\ (i = 1,\ 2,\ \ldots,), \tag{15.26}$$

where summing goes using the corresponding associative operation.

Theorem 15.3 (parallel prefix computation in $\Theta(\lg p)$ time). *Algorithm* OPTIMAL-PREFIX *computes the prefixes of p elements on $p/\lg p$* CREW PRAM *processors in $\Theta(\lg p)$ time.*

Proof Line 1 runs in $\Theta(\lg p)$ time, line 2 runs $O(\lg(p/\lg p)) = O(\lg p)$ time, line 3 runs $\Theta(\lg p)$ time. ■

This theorem imply that the work of OPTIMAL-PREFIX is $\Theta(p)$, therefore OPTIMAL-PREFIX is a work-optimal algorithm.

Let the elements of the sequence $X = x_1, x_2, \ldots, x_p$ be the elements of the alphabet Σ. Then the input data of the prefix computation are the elements of the sequence X, and the ***prefix problem*** is the computation of the elements $x_1,\ x_1 \oplus x_2,\ \ldots, x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_p$. These computable elements are called ***prefixes***.

We remark, that in some books on parallel programming often the elements of the sequence X are called prefixes.

Example 15.3 Associative operations. If Σ is the set of integers, $\oplus$ denotes the addition and the sequence of the input data is 3, -5, 8, 2, 5, 4, then the prefixes are 3, -2, 6, 8, 13, 17. If the alphabet and the input data are the same, the operation is the multiplication, then the output data (prefixes) are 3, -15, -120, -240, -1200, -4800. If the operation is the minimum (it is also associative), then the prefixes are 3, -5, -5, -5, -5, -5. The last prefix

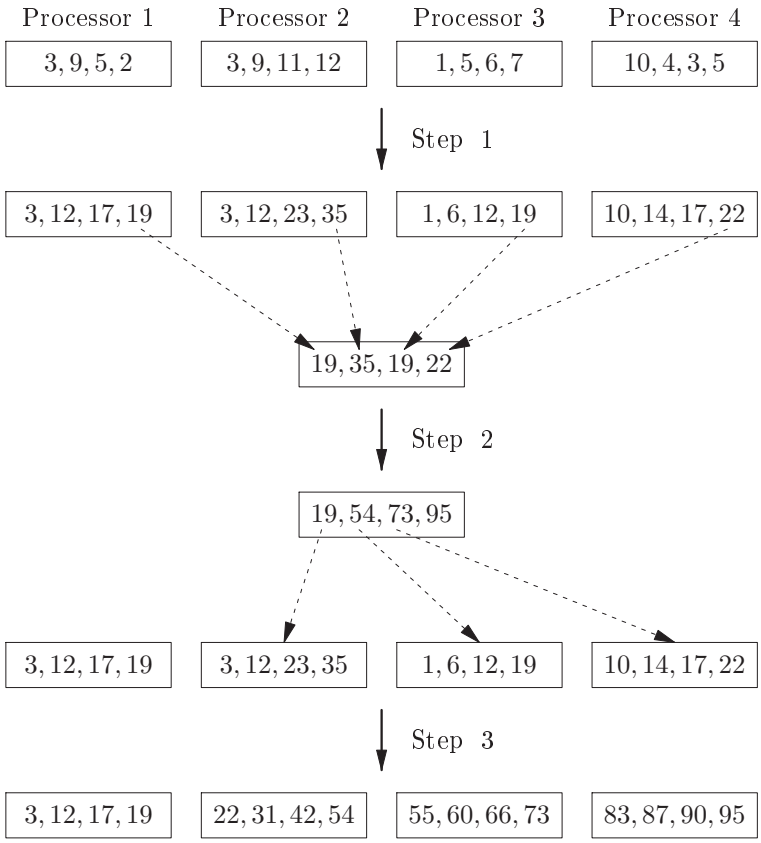


Figure 15.17. Computation of prefixes of 16 elements using OPTIMAL-PREFIX.

equals to the smallest input data.

Sequential prefix calculation can be solved in $O(p)$ time. Any A sequential algorithm needs $N(p, \mathrm{A}) = \Omega(\mathrm{n})$ time. There exist work-effective parallel algorithms solving the prefix problem.

Our first parallel algorithm is CREW-PREFIX, which uses p CREW PRAM processors and requires $\Theta(\lg p)$ time. Then we continue with algorithm EREW-PREFIX, having similar qualitative characteristics, but running on EREW PRAM model too.

These algorithms solve the prefix problem quicker, than the sequential algorithms, but the order of their work is larger.

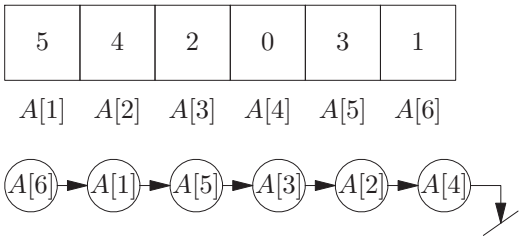


Figure 15.18. Input data of array ranking and the the result of the ranking.

Algorithm Optimal-Prefix requires only $\lceil p/\lg p \rceil$ CREW PRAM processors and in spite of the reduced numbers of processors requires only $O(\lg p)$ time. So its work is $O(p)$, therefore its efficiency is $\Theta(1)$ and is work-effective. The speedup of the algorithm is $\Theta(n/\lg n)$.

15.6.2. Ranking

The input of the ***list ranking problem*** is a list represented by an array $A[1\,..\,p]$: each element contains the index of its ***right neighbour*** (and maybe further data). The task is to determine the rank of the elements. The ***rank*** is defined as the number of the right neighbours of the given element.

Since the further data are not necessary to find the solution, for the simplicity we suppose that the elements of the array contain only the index of the right neighbour. This index is called ***pointer***. The pointer of the rightmost element equals to zero.

Example 15.4 *Input of list ranking.* Let $A[1\,..\,6]$ be the array represented in the first row of Figure 15.18. Then the right neighbour of the element $A[1]$ is $A[5]$, the right neighbour of $A[2]$ is $A[4]$. $A[4]$ is the last element, therefore its rank is 0. The rank of $A[2]$ is 1, since only one element, $A[4]$ is to right from it. The rank of $A[1]$ is 4, since the elements $A[5], A[3], A[2]$ and $A[4]$ are right from it. The second row of Figure 15.18 shows the elements of A in decreasing order of their ranks.

The list ranking problem can be solved in linear time using a sequential algorithm. At first we determine ***the head of the list*** which is the unique $A[i]$ having the property that does not exist an index j $(1 \leq j \leq p)$ with $A[j] = i$. In our case the head of A is $A[6]$. The head of the list has the rank $p-1$, its right neighbour has a rank $p-2, \ldots$ and finally the rank of the last element is zero.

In this subsection we present a deterministic list ranking algorithm, which uses p EREW PRAM processors and in worst case $\Theta(\lg p)$ time. The pseudocode of algorithm Det-Ranking is as follows.

The input of the algorithm is the number of the elements to be ranked (p), the array $N[1\,..\,p]$ containing the index of the right neighbour of the elements of A, output is the array $R[1\,..\,p]$ containing the computed ranks.

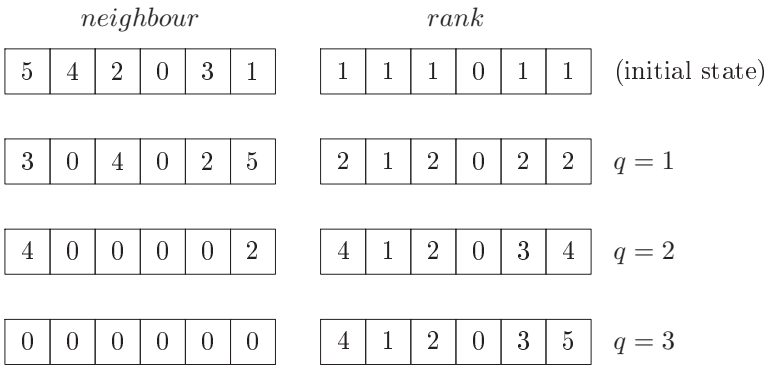


Figure 15.19. Work of algorithm DET-RANKING on the data of Example 15.4.

DET-RANKING(p, N)

```
1  P_i in parallel for i ← 1 to p
2     do if N[i] = 0
3        then R[i] ← 0
4        else  R[i] ← 1
5  for j ← 1  to  ⌈lg p⌉
6     do P_i in parallel for i ← 1 to p
7           do if N[i] ≠ 0
8                 then R[i] ← R[i] + R[N[i]]
9                      N[i] ← N[N[i]]
10 return R
```

The basic idea behind the algorithm DET-RANKING is the ***pointer jumping***. According to this algorithm at the beginning each element contains the index of its right neighbour, and accordingly its provisional rank equal to 1 (with exception of the last element of the list, whose rank equals to zero). This initial state is represented in the first row of Figure 15.19.

Then the algorithm modifies the element so, that each element points to the right neighbour of its right neighbour (if it exist, otherwise to the end of the list). This state is represented in the second row of Figure 15.19.

If we have p processors, then it can be done in $O(1)$ time.

After this each element (with exception of the last one) shows to the element whose distance was originally two. In the next step of the pointer jumping the elements will show to such other element whose distance was originally 4 (if there is no such element, then to the last one), as it is shown in the third row of Figure 15.19.

In the next step the pointer part of the elements points to the neighbour of distance 8 (or to the last element, if there is no element of distance 8), according to the last row of Figure 15.19.

In each step of the algorithm each element updates the information on the num-

ber of elements between itself and the element pointed by the pointer. Let $R[i]$, resp. $N[i]$ the rank, resp. neighbour field of the element $A[i]$. The initial value of $R[i]$ is 1 for the majority of the elements, but is 0 for the rightmost element ($R(4) = 0$ in the first line of Figure 15.19). During the pointer jumping $R[i]$ gets the new value (if $N[i] \neq 0$) gets the new value $R[i] + R[N[i]]$, if $N[i] \neq 0$. E.g. in the second row of Figure 15.19) $R[1] = 1 + 1 = 2$, since its previous rank is 1, and the rank of its right neighbour is also 1. After this $N[i]$ will be modified to point to $N[N[i]]$. E.g. in the second row of Figure 15.19 $N[1] = 3$, since the right neighbour of the right neighbour of $A[1]$ is $A[3]$.

Theorem 15.4 *Algorithm* DET-RANKING *computes the ranks of an array consisting of p elements on p* EREW PRAM *processors in $\Theta(\lg p)$ time.*

Since the work of DET-RANKING is $\Theta(p \lg p)$, this algorithm is not work-optimal, but it is work-efficient.

The list ranking problem corresponds to a list prefix problem, where each element is 1, but the last element of the list is 0. One can easily modify DET-RANKING to get a prefix algorithm.

15.6.3. Merge

The ***merging problem*** The input of the ***merging problem*** is two sorted sequences X_1 and X_2 and the output is one sorted sequence Y containing the elements of the input.

If the length of the input sequences is p, then the merging problem can be solved in $O(p)$ time using a sequential processor. Since we have to investigate all elements and write them into the corresponding element of Y, the running time of any algorithm is $\Omega(p)$. We get this lower bound even in the case when we count only the number of necessary comparisons.

Merge in logarithmic time

Let $X_1 = x_1, x_2, \ldots, x_m$ and $X_2 = x_{m+1}, x_{m+2}, \ldots, x_{2m}$ be the input sequences. For the shake of simplicity let m be the power of two and let the elements be different.

To merge two sequences of length $m$it is enough to know the ranks of the keys, since then we can write the keys—using $p = 2m$ processors—into the corresponding memory locations with one parallel write operation. The running time of the following algorithm is a logarithmic, therefore it is called LOGARITHMIC-MERGE.

Theorem 15.5 *Algorithm* LOGARITHMIC-MERGE *merges two sequences of length m on $2m$* CREW PRAM *processors in $\Theta(\lg m)$ time.*

Proof Let the rank of element x be r_1 (r_2) in X_1 (in X_2). If $x = x_j \in X_1$, then let $r_1 = j$. If we assign a single processor P to the element x, then it can determine, using binary search, the number q of elements in X_2, which are smaller than x. If q is known, then P computes the rank r_j in the union of X_1 and X_2, as $j + q$. If x belongs to X_2, the method is the same.

Summarising the time requirements we get, that using one CREW PRAM processor per element, that is totally $2m$ processors the running time is $\Theta(\lg m)$. ■

This algorithm is not work-optimal, only work-efficient.

Odd-even merging algorithm

This following recursive algorithm ODD-EVEN-MERGE follows the classical *divide-and-conquer* principle.

Let $X_1 = x_1, x_2, \ldots, x_m$ and $X_2 = x_{m+1}, x_{m+2}, \ldots, x_{2m}$ be the two input sequences. We suppose that m is a power of 2 and the elements of the arrays are different. The output of the algorithm is the sequence $Y = y_1, \ldots, y_{2m}$, containing the merged elements. This algorithm requires $2m$ EREW PRAM processors.

ODD-EVEN-MERGE(X_1, X_2)

```
1  if m = 1
2     then get Y by merging x_1 and x_2 with one comparison
3  return Y
4  if m > 1
5     then P_i in parallel for i ← 1 to m
6             do merge recursively X_1^odd = x_1, x_3, ..., x_{m-1} and
7                X_2^odd = x_{m+1}, x_{m+3}, ..., x_{2m-1} to get L_1 = l_1, l_2, ..., l_m
8          P_i in parallel for 1 ← m + 1 to 2m
9             do merge recursively X_1^even = x_2, x_4, ..., x_m and
10               X_2^even = x_{m+2}, x_{m+4}, ..., x_{2m} to get L_2 = l_{m+1}, l_{m+2}, ..., l_{2m}
11         P_i in parallel for i ← 1 to m
12            do y_{2i-1} ← l_i
13               y_{2i} ← l_{m+i}
14               if y[2i] > y[2i + 1]
13                  then z ← y[2i]
14                       y[2i] ← y[2i + 1]
15                       y[2i + 1] ← z
15 return Y
```

Example 15.5 *Merge of twice eight numbers.* Let X_1 = 1, 5, 8, 11, 13, 16, 21, 26 and X_2 = 3, 9, 12, 18, 23, 27, 31, 65. Figure 15.20 shows the sort of 16 numbers.

At first elements of X_1 with odd indices form the sequence X_1^{odd} and elements with even indices form the sequence X_1^{even}, and in the same way we get the sequences X_2^{odd} and X_2^{even}. Then comes the recursive merge of the two odd sequences resulting L_1 and the recursive merge of the even sequences resulting L_2.

After this ODD-EVEN-MERGE shuffles L_1 and L_2, resulting the sequence $Y = y_1, \ldots, y_{2m}$: the elements of Y with odd indices come from L_1 and the elements with even indices come from L_2.

Finally we compare the elements of Y with even index and the next element (that is $Y[2]$ with $Y[3]$, $Y[4]$ with $Y[5]$ etc.) and if necessary (that is they are not in the good order) they are changed.

Theorem 15.6 (merging in $\Theta(\lg m)$ time). *Algorithm* ODD-EVEN-MERGE *merges two sequences of length m elements in $\Theta(\lg m)$ time using $2m$* EREW PRAM *processors.*

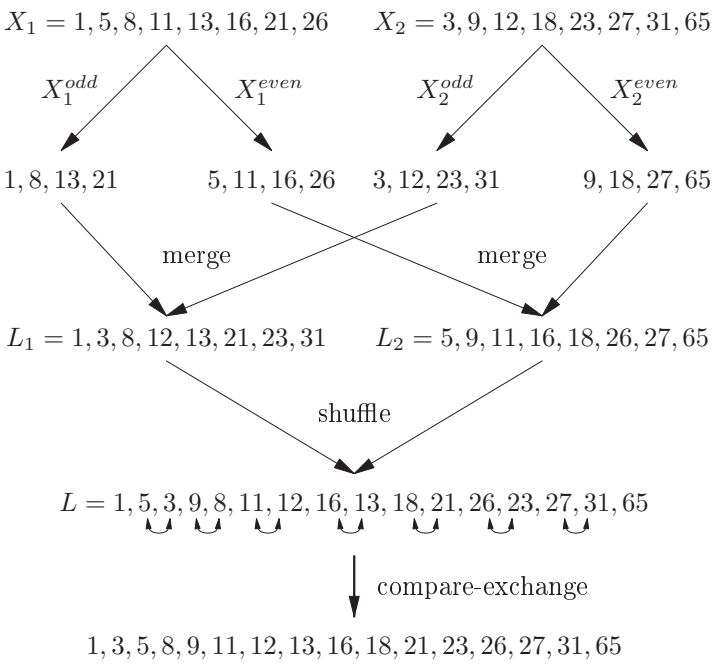


Figure 15.20. Sorting of 16 numbers by algorithm ODD-EVEN-MERGE.

Proof Let denote the running time of the algorithm by $W(m)$. Step 1 requires $\Theta(1)$ time, Step 2 $m/2$ time. Therefore we get the recursive equation

$$W(m) = W(m/2) + \Theta(1), \tag{15.27}$$

having the solution $W(m) = \Theta(\lg m)$. ■

We prove the correctness of this algorithm using the ***zero-one principle.***

A comparison-based sorting algorithm is ***oblivious,*** if the sequence of comparisons is fixed (elements of the comparison do not depend on the results of the earlier comparisons). This definition means, that the sequence of the pairs of elements to be compared $(i_1, j_1),\ (i_2, j_2), \ldots,\ (i_m, j_m)$ is given.

Theorem 15.7 (zero-one principle). *If a simple comparison-based sorting algorithm correctly sorts an arbitrary 0-1 sequence of length n, then it sorts also correctly any sequence of length n consisting of arbitrary keys.*

Proof Let A be a comparison-based oblivious sorting algorithm and let S be such a sequence of elements, sorted incorrectly by A. Let suppose A sorts in increasing order the elements of S. Then the incorrectly sorted sequence S' contains an element x on the i-th $(1 \leq i \leq n-1)$ position in spite of the fact that S contains at least i keys smaller than x.

Let x be the first (having the smallest index) such element of S. Substitute in the input sequence the elements smaller than x by 0's and the remaining elements

by 1's. This modified sequence is a 0-1 sequence therefore A sorts it correctly. This observation implies that in the sorted 0-1 sequence at least i 0's precede the 1, written on the place of x.

Now denote the elements of the input sequence smaller than x by red colour, and the remaining elements by blue colour (in the original and the transformed sequence too). We can show by induction, that the coloured sequences are identical at the start and remain identical after each comparison. According to colours we have three types of comparisons: blue-blue, red-red and blue-red. If the compared elements have the same colour, in both cases (after a change or not-change) the colours remain unchanged. If we compare elements of different colours, then in both sequences the red element occupy the position with smaller index. So finally we get a contradiction, proving the assertion of the theorem. ■

Example 15.6 *A non comparison-based sorting algorithm.* Let $x_1, x_2, \ldots, x_n$ be a bit sequence. We can sort this sequence simply counting the zeros, and if we count z zeros, then write z zeros, then $n - z$ ones. Of course, the general correctness of this algorithm is not guaranteed. Since this algorithm is not comparison-based, therefore this fact does not contradict to the zero-one principle.

But merge is sorting, and ODD-EVEN-MERGE is an oblivious sorting algorithm.

Theorem 15.8 *Algorithm* ODD-EVEN-MERGE *sorts correctly sequences consisting of arbitrary numbers.*

Proof Let X_1 and X_2 sorted 0-1 sequences of length m. Let q_1 (q_2) the number of zeros at the beginning of X_1 (X_2). Then the number of zeros in X_1^{odd} equals to $\lceil q_1/2 \rceil$, while the number of zeros in X_1^{even} is $\lfloor q_1/2 \rfloor$. Therefore the number of zeros in L_1 equals to $z_1 = \lceil q_1/2 \rceil + \lceil q_2/2 \rceil$ and the number of zeros in L_2 equals to $z_2 = \lfloor q_1/2 \rfloor + \lfloor q_2/2 \rfloor$.

The difference of z_1 and z_2 is at most 2. This difference is exactly then 2, if q_1 and q_2 are both odd numbers. Otherwise the difference is at most 1. Let suppose, that $|z_1 - z_2| = 2$ (the proof in the other cases is similar). In this cases L_1 contains two additional zeros. When the algorithm shuffles L_1 and L_2, L begins with an even number of zeros, end an even number of ones, and between the zeros and ones is a short "dirty" part, 0, 1. After the comparison and change in the last step of the algorithm the whole sequence become sorted. ■

A work-optimal merge algorithm

Algorithm WORK-OPTIMAL-MERGE uses only $\lceil 2m/\lg m \rceil$ processors, but solves the merging in logarithmic time. This algorithm divides the original problem into $m/\lg m$ parts so, that each part contains approximately $\lg m$ elements.

Let $X_1 = x_1, x_2, \ldots, x_m$ and $X_2 = x_{m+1}, x_{m+2}, \ldots, x_{m+m}$ be the input sequences. Divide X_1 into $M = \lceil m/\lg m \rceil$ parts so, that each part contain at most $\lceil \lg m \rceil$ elements. Let the parts be denoted by $A_1, A_2, \ldots, A_M$. Let the largest element in A_1 be l_i ($i = 1, 2, \ldots, M$).

Assign a processor to each l_i element. These processors determine (by binary search) the correct place (according to the sorting) of l_i in X_2. These places divide

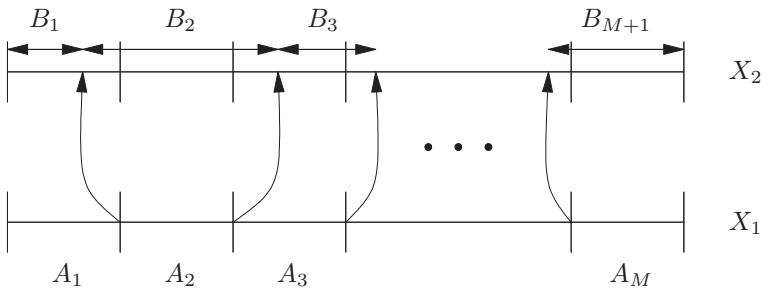


Figure 15.21. A work-optimal merge algorithm OPTIMAL-MERGE.

X_2 to $M+1$ parts (some of these parts can be empty). Let denote these parts by B_1, B_2, ..., B_{M+1}. We call B_i the subset corresponding to A_i in X_2 (see Figure 15.21).

The algorithm gets the merged sequence merging at first A_1 with B_1, A_2 with B_2 and so on, and then joining these merged sequences.

Theorem 15.9 *Algorithm* OPTIMAL-MERGING *merges two sorted sequences of length m in $O(\lg m)$ time on $\lceil 2m/\lg m \rceil$* CREW PRAM *processors.*

Proof We use the previous algorithm.

The length of the parts A_i is $\lg m$, but the length of the parts B_i can be much larger. Therefore we repeat the partition. Let A_i, B_i an arbitrary pair. If $|B_i| = O(\lg m)$, then A_i and B_i can be merged using one processor in $O(\lg m)$ time. But if $|B_i| = \omega(\lg m)$, then divide B_i into $|B_i|/\lg m$ parts—then each part contains at most $\lg m$ keys. Assign a processor to each part. This assigned processor finds the subset corresponding to this subsequence in A_i: $O(\lg \lg m)$ time is sufficient to do this. So the merge of A_i and B_i can be reduced to $|B_i|/\lg m$ subproblems, where each subproblem is the merge of two sequences of $O(\lg m)$ length.

The number of the used processors is $\sum_{i=1}^{M} \lceil |B_i|/\lg m \rceil$, and this is at most $m/\lg m + M$, what is not larger then $2M$. ■

This theorem imply, that OPTIMAL-MERGING is work-optimal.

Corollary 15.10 OPTIMAL-MERGING *is work-optimal.*

15.6.4. Selection

In the ***selection problem*** $n \geq 2$ elements and a positive integer i $(1 \leq i \leq n)$ are given and the i-th smallest element is to be selected.

Since selection requires the investigation of all elements, and our operations can handle at most two elements, so $N(n) = \Omega(n)$.

Since it is known sequential algorithm A requiring only $W(n, \mathcal{A}) = O(n)$ time, so A is asymptotically optimal.

The ***search problem*** is similar: in that problem the algorithm has to decide, whether a given element appears in the given sequence, and if yes, then where. Here

negative answer is also possible and the features of any element decide, whether it corresponds the requirements or not.

We investigate three special cases and work-efficient algorithms to solve them.

Selection in constant time using n^2 processors
Let $i = n$, that is we wish to select the largest key. Algorithm QUADRATIC-SELECT solves this task in $\Theta(1)$ time using n^2 CRCW processors.

The input (n different keys) is the sequence $X = x_1, x_2, \ldots, x_n$, and the selected largest element is returned as y.

QUADRATIC-SELECT(X)

```
 1  if n = 1
 2    then y ← x_1
 3         return y
 4  P_ij in parallel for i ← 1 to n, j ← 1 to n
       do if k_i < k_j
 5           then x_{i,j} ← FALSE
 6           else  x_{i,j} ← TRUE
 7  P_i1 in parallel for i ← 1 to n
 8       do L_i ← TRUE
 9  P_ij in parallel for i ← 1 to n, j ← 1 to n
10       if x_{i,j} = FALSE
11         then L_i ← FALSE
12  P_i1 in parallel for i ← 1 to n
13       do if L_i = TRUE
14            then y ← x_i
15  return y
```

In the first round (lines 4–6) the keys are compared in parallel manner, using all the n^2 processors. P_{ij} $(1 \le i, j \le n)$ so, that processor P_{ij} computes the logical value $x_{i,j} = x_i < x_j$. We suppose that the keys are different. If the elements are not different, then we can use instead of x_i the pair (x_i, i) (this solution requires an additional number of length $(\lg n)$ bits. Since there is a unique key for which all comparison result FALSE, this unique key can be found with a logical OR operation is lines 7–11.

Theorem 15.11 (selection in $\Theta(1)$ time). *Algorithm* QUADRATIC-SELECT *determines the largest key of n different keys in $\Theta(1)$ time using n^2* CRCW *common* PRAM *processors.*

Proof First and third rounds require unit time, the second round requires $\Theta(1)$ time, so the total running time is $\Theta(1)$. ∎

The speedup of this algorithm is $\Theta(n)$. The work of the algorithm is $w = \Theta(n^2)$. So the efficiency is $E = \Theta(n)/\Theta(n^2) = \Theta(1/n)$. It follows that this algorithm is not work-optimal, even it is not work-effective.

Selection in logarithmic time on n processors

Now we show that the maximal element among n keys can be found, using even only n common CRCW PRAM processors and $\Theta(\lg\lg n)$ time. The used technique is the *divide-and-conquer*. For the simplicity let n be a square number.

The input and the output are the same as at the previous algorithm.

QUICK-SELECTION(X, y)

```
1  if p = 1
2     then y ← x_1
3     return y
4  if p > 1
5     then divide the input into groups G_1, G_2, ..., G_a and
        divide the processors into groups Q_1, Q_2, ..., Q_a
6  Q_i in parallel for i ← 1 to a
6     do recursively determines the maximal element M_i of the group G_i
7  QUADRATIC-SELECT(M)
8  return y
```

The algorithm divides the input into $\sqrt{p} = a$ groups $(G_1, G_2, \ldots, G_a)$ so, that each group contains a elements $(x_{(i-1)a+1}, x_{(i-1)a+2}, \ldots, x_{ia})$, and divides the processors into a groups $(Q_1, Q_2, \ldots, Q_a)$ so, that group Q_i contains a processors $P_{(i-1)a+1}, P_{(i-1)a+2}, \ldots, P_{ia}$. Then the group of processors Q_i computes recursively the maximum M_i of group G_i. Finally the previous algorithm QUADRATIC-SELECT gets as input the sequence $M = M_1, \ldots, M_a$ and finds the maximum y of the input sequence X.

Theorem 15.12 (selection in $\Theta(\lg\lg p)$ time). *Algorithm* QUICK-SELECT *determines the largest of p different elements in $O(\lg\lg p)$ time using n common* CRCW PRAM *processors.*

Proof Let the running time of the algorithm denoted by $W(n)$. Step 1 requires $W(\sqrt{n})$ time, step 2 requires $\Theta(1)$ time. Therefore $W(p)$ satisfies the recursive equation

$$W(p) = W(\sqrt{p}) + \Theta(1), \tag{15.28}$$

having the solution $\Theta(\lg\lg p)$. ■

The total work of algorithm QUICK-SELECT is $\Theta(p\lg\lg p)$, so its efficiency is $\Theta(p)/\Theta(p\lg\lg p) = \Theta(1/\lg\lg p)$, therefore QUICK-SELECT is not work-optimal, it is only work-effective.

Selection from integer numbers

If the problem is to find the maximum of n keys when the keys consist of one bit, then the problem can be solved using a logical OR operation, and so requires only constant time using n processors. Now we try to extend this observation.

Let c be a given positive integer constant, and we suppose the keys are integer numbers, belonging to the interval $[0, n^c]$. Then the keys can be represented using at

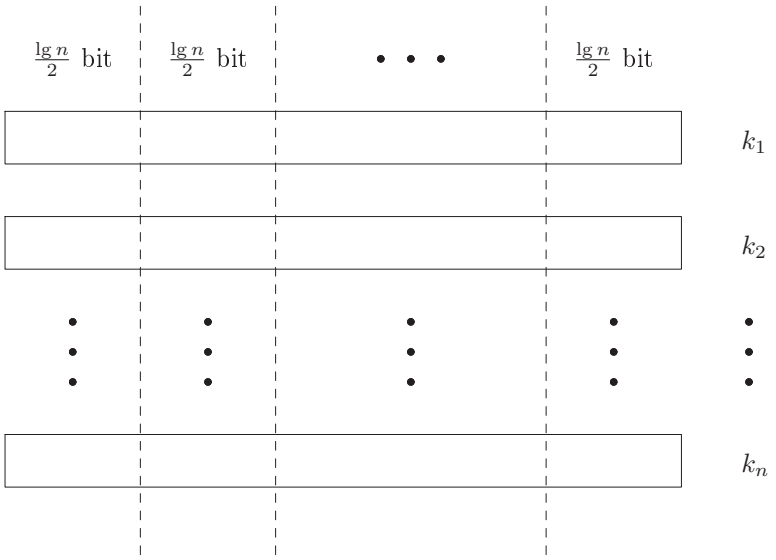


Figure 15.22. Selection of maximal integer number.

most $c \lg n$ bits. For the simplicity we suppose that all the keys are given as binary numbers of length $c \lg n$ bits.

The following algorithm INTEGER-SELECTION requires only constant time and n CRCW PRAM processors to find the maximum.

The basic idea is to partition the $b_1, b_2, \ldots, b_{2c}$ bits of the numbers into parts of length $(\lg n)/2$. The i-th part contains the bits $b_{(i-1)+1}$, $b_{(i-1)+2}, \ldots, b_{(i-1)+b_{(i-1)+(\lg n)/2}}$, the number of the parts is $2c$. Figure 15.22 shows the partition.

The input of INTEGER-SELECTION is the number of processors (n) and the sequence $X = x_1,\ x_2,\ \ldots,\ x_n$ containing different integer numbers, and output is the maximal number y.

INTEGER-SELECTION(p, X)

1 **for** $i \leftarrow 1$ **to** $2c$
2 **do** compute the maximum (M) of the remaining numbers on the base of their i-th part
3 delete the numbers whose i-th part is smaller than M
4 $y \leftarrow$ one of the remaining numbers
5 **return** y

The algorithm starts with searching the maximum on the base of the first part of the numbers. Then it delete the numbers, whose first part is smaller, than the maximum. Then this is repeated for the second, ..., last part of the numbers. Any of the non deleted numbers is maximal.

Theorem 15.13 (selection from integer numbers). *If the numbers are integers drawn from the interval* $[0, n^c]$, *then algorithm* INTEGER-SELECTION *determines the largest number among* n *numbers for any positive* c *in* $\Theta(1)$ *time using* n CRCW PRAM *processors.*

Proof Let suppose that we start with the selection of numbers, whose $(\lg n)/2$ most significant bits are maximal. Let this maximum in the first part denoted by M. It is sure that the numbers whose first part is smaller than M are not maximal, therefore can be deleted. If we execute this basis operation for all parts (that is $2c$ times), then exactly those numbers will be deleted, what are not maximal, and all maximal element remain.

If a key contains at most $(\lg n)/2$ bits, then its value is at most $\sqrt{n} - 1$. So algorithm INTEGER-SELECT in its first step determines the maximum of integer numbers taken from the interval $[0, \sqrt{n}-1]$. The algorithm assigns a processor to each number and uses $\sqrt{n}$ common memory locations $(M_1, M_2, \ldots, M_{\sqrt{n}-1})$, containing initially $-\infty$. In one step processor P_i writes k_i into M_{k_i}. Later the maximum of all numbers can be determined from $\sqrt{n}$ memory cells using n processors by Theorem 15.11 in constant time. ■

General selection

Let the sequence $X = x_1,\ x_2, \ldots, x_n$ contain different numbers and the problem is to select the kth smallest element of X. Let we have $p = n^2/\lg n$ CREW processors.

GENERAL-SELECTION(X)

1 divide the $n^2/\lg n$ processors into n groups $G_1, \ldots, G_n$ so, that group G_i contains the processors $P_{i,1}, P_{i,2}, \ldots, P_{i,n/\lg n}$ and divide the n elements into $n/\lg n$ groups $(X_1, X_2, \ldots, X_{n/\lg n})$ so, that group X_i contains the elements $x_{(i-1)\lg n)+1}, x_{(i-1)\lg n)+2}, \ldots, x_{(i-1)\lg n)+\lg n}$
2 P_{ij} **in parallel for** $i \leftarrow 1$ **to** n
3 **do** determine h_{ij} (how many elements of X_j are smaller, than x_i)
4 G_i **in parallel for** $i \leftarrow 1$ **to** n
5 **do** using OPTIMAL-PREFIX determine s_i (how many elements of X are smaller, than x_i)
6 $P_{i,1}$ **in parallel for** $i \leftarrow 1$ **to** n
7 **do if** $s_i = k - 1$
8 **return** x_i

Theorem 15.14 (general selection). *The algorithm* GENERAL-SELECTION *determines the* i*-th smallest of* n *different numbers in* $\Theta(\lg n)$ *time using* $n^2/\lg n$ *processors.*

Proof In lines 2–3 P_{ij} works as a sequential processor, therefore these lines require $\Theta\lg n$ time. Lines 4–5 require $\Theta \lg n$ time according to Theorem 15.3. Lines 6–8 can be executed in constant time, so the total running time is $\Theta(\lg n)$. ■

The work of GENERAL-SELECTION is $\Theta(n^2)$, therefore this algorithm is not work-effective.

15.6.5. Sorting

Given a sequence $X = x_1, x_2, \ldots, x_n$ the ***sorting problem*** is to rearrange the elements of X e.g. in increasing order.

It is well-known that any A sequential comparison-based sorting algorithm needs $N(n, A) = \Omega(n \lg n)$ comparisons, and there are comparison-based sorting algorithms with $O(n \lg n)$ running time.

There are also algorithms, using special operations or sorting numbers with special features, which solve the sorting problem in linear time. If we have to investigate all elements of X and permitted operations can handle at most 2 elements, then we get $N(n) = \Omega(n)$. So it is true, that among the comparison-based and also among the non-comparison-based sorting algorithms are asymptotically optimal sequential algorithms.

In this subsection we consider three different sorting algorithm.

Sorting in logarithmic time using n^2 processors
Using the ideas of algorithms QUADRATIC-SELECTION and OPTIMAL-PREFIX we can sort n elements using n^2 processors in $\lg n$ time.

QUADRATIC-SORT(K)

```
 1  if n = 1
 2     then y ← x_1
 3          return Y
 4  P_ij in parallel for i ← 1 to n, j ← 1 to n
        do if x_i < x_j
 5             then x_{i,j} ← 0
 6             else x_{i,j} ← 1
 7  divide the processors into n groups (G_1, G_2, ..., G_n) so, that group G_i contains
    processors P_{i,1}, P_{i,2}, ..., P_{i,n}
 8  G_i in parallel for i ← 1 to n
 9     do compute s_i = x_{i,1} + x_{i,2} + ··· + x_{i,n}
10  P_i1 in parallel for i ← 1 to n
11     do y_{s_i+1} ← x_i
12  return Y
```

In lines 4–7 the algorithm compares all pairs of the elements (as QUADRATIC-SELECTION), then in lines 7–9 (in a similar way as OPTIMAL-PREFIX WORKS) it counts, how many elements of X is smaller, than the investigated x_i, and finally in lines 10–12 one processor of each group writes the final result into the corresponding memory cell.

Theorem 15.15 (sorting in $\Theta(\lg n)$ time). *Algorithm* QUADRATIC-SORT *sorts n elements using n^2* CRCW PRAM *processors in $\Theta(\lg n)$ time.*

Proof Lines 8–9 require $\Theta \lg n$ time, and the remaining lines only constant time. ■

Since the work of QUADRATIC-SORT is $\Theta(n^2 \lg n)$, this algorithm is not work-effective.

Odd-even algorithm with $O(\lg n)$ running time
The next algorithm uses the ODD-EVEN-MERGE algorithm and the classical *divide-and-conquer* principle. The input is the sequence $X = x_1, \ldots, x_p$, containing the numbers to be sorted, and the output is the sequence $Y = y_1, \ldots, y_p$, containing the sorted numbers.

ODD-EVEN-SORT(X)

```
1  if n = 1
2     then Y ← X
3  if n > 1
4     then let X_1 = x_1, x_2, ..., x_{n/2} and X_2 = x_{n/2+1}, x_{n/2+2}, ..., x_n.
5  P_i in parallel for i ← 1 to n/2
6     do sort recursively X_1 to get Y_1
7  P_i in parallel for i ← n/2 + 1 to n
8     do sort recursively X_2 to get Y_2
9  P_i in parallel for i ← 1 to n
10    do merge Y_1 and Y_2 using ODD-EVEN-MERGE(Y_1, Y_2)
11 return Y
```

The running time of this EREW PRAM algorithm is $O(\lg^2 n)$.

Theorem 15.16 (sorting in $\Theta(\lg^2 n)$ time). *Algorithm* ODD-EVEN-SORT *sorts n elements in $\Theta(\lg^2 n)$ time using n* EREW PRAM *processors.*

Proof Let $W(n)$ be the running time of the algorithm. Lines 3–4 require $\Theta(1)$ time, Lines 5–8 require $W(n/2)$ time, and lines 9–10 require $\Theta(\lg n)$ time, line 11 require $\Theta(1)$ time. Therefore $W(n)$ satisfies the recurrence

$$W(n) = \Theta(1) + W(n/2) + \Theta(\lg n), \tag{15.29}$$

having the solution $W(n) = \Theta(\lg^2 n)$. ■

Example 15.7 *Sorting on 16 processors.* Sort using 16 processors the following numbers: 62, 19, 8, 5, 1, 13, 11, 16, 23, 31, 9, 3, 18, 12, 27, 34. At first we get the odd and even parts, then the first 8 processors gets the sequence $X_1 = 62, 19, 8, 5, 1, 13, 11, 16$, while the other 8 processors get $X_2 = 23, 31, 9, 4, 18, 12, 27, 34$. The output of the first 8 processors is $Y_1 = 1, 5, 8, 11, 13, 16, 19, 62$, while the output of the second 8 processors is $Y_2 = 3, 9, 12, 18, 23, 27, 31, 34$. The merged final result is $Y = 1, 3, 5, 8, 9, 11, 12, 13, 16, 18, 19, 23, 27, 31, 34, 62$.

The work of the algorithm is $\Theta(n \lg^2 n)$, its efficiency is $\Theta(1/\lg n)$, and its speedup is $\Theta(n/\lg n)$. The algorithm is not work-optimal, but it is work-effective.

Algorithm of Preparata with $\Theta(\lg n)$ running time
If we have more processors, then the running time can be decreased. The following recursive algorithm due to Preparata uses $n \lg n$ CREW PRAM processors and $\lg n$ time. Input is the sequence $X = x_1, x_2, \ldots, x_n]$, and the output is the sequence $Y = y_1, y_2, \ldots, y_n$ containing the sorted elements.

PREPARATA(X)

```
1  if n ≤ 20
2     then sort X using n processors and ODD-EVEN-SORT
3     return Y
4  divide the n elements into lg n parts (X_1, X_2, ..., X_{lg n}) so, that each part
   part contains n/lg n elements, and divide the processors into lg n groups
   (G_1, G_2, ..., G_n) so, that each group contains n processors
5  G_i in parallel for i ← 1 to lg n
6     do sort the part X_i recursively to get a sorted sequence S_i
7  divide the processors into (lg n)^2 groups (H_{1,1}, H_{1,2}, ..., H_{(lg n, lg n)})
   containing n/lg n processors
8  H_{i,j} in parallel for i ← 1 to lg n, j ← 1 to lg n
9        do merge S_i and S_j
10 divide the processors into n groups (J_1, J_2, ..., J_n) so, that each group
   contains lg n processors
11 J_i in parallel for i ← 1 to n
12    do determine the ranks of the x_i element in X using the local ranks
         received in line 9 and using the algorithm OPTIMAL-PREFIX
13       Y_i ← the elements of X having a rank i
14 return Y
```

This algorithm uses the *divide-and-conquer* principle. It divides the input into $\lg n$ parts, then merges each pair of parts. This merge results local ranks of the elements. The global rank of the elements can be computed summing up these local ranks.

Theorem 15.17 (sorting in $\Theta(\lg n)$ time). *Algorithm* PREPARATA *sorts n elements in $\Theta(\lg n)$ time using $n \lg n$* CREW PRAM *processors.*

Proof Let the running time be $W(n)$. Lines 4–6 require $W(n/\lg n)$ time, lines 7–12 together $\Theta(\lg\lg n)$. Therefore $W(n)$ satisfies the equation

$$W(n) = W(n/\lg n) + \Theta(\lg\lg n), \tag{15.30}$$

having the solution $W(n) = \Theta(\lg n)$. ■

The work of PREPARATA is the same, as the work of ODD-EVEN-SORT, but the speedup is better: $\Theta(n)$. The efficiency of both algorithms is $\Theta(1/\lg n)$.

Exercises

15.6-1 The memory cell M_1 of the global memory contains some data. Design an algorithm, which copies this data to the memory cells $M_2, M_3, \ldots, M_n$ in $O(\lg n)$ time, using n EREW PRAM processors.

15.6-2 Design an algorithm which solves the previous Exercise 15.6-1 using only $n/\lg n$ EREW PRAM processors saving the $O(\lg n)$ running time.

15.6-3 Design an algorithm having $O(\lg\lg n)$ running time and determining the maximum of n numbers using $n/\lg\lg n$ common CRCW PRAM processors.

15.6-4 Let X be a sequence containing n keys. Design an algorithm to determine the rank of any $k \in X$ key using $n/\lg n$ CREW PRAM processors and $O(\lg n)$ time.
15.6-5 Design an algorithm having $O(1)$ running time, which decides using n common CRCW PRAM processors, whether element 5 is contained by a given array $A[1\,..\,n]$, and if is contained, then gives the largest index i, for which $A[i] = 5$ holds.
15.6-6 Design algorithm to merge two sorted sequence of length m in $O(1)$ time, using n^2 CREW PRAM processors.
15.6-7 Determine the running time, speedup, work, and efficiency of all algorithms, discussed in this section.

15.7. Mesh algorithms

To illustrate another model of computation we present two algorithms solving the prefix problem on meshes.

15.7.1. Prefix on chain

Let suppose that processor P_i $(i = 1,\ 2,\ \ldots,\ p)$ of the chain $\mathcal{L} = \{P_1, P_2, \ldots, P_p\}$ stores element x_i in its local memory, and after the parallel computations the prefix y_i will be stored in the local memory of P_i.

At first we introduce a naive algorithm. Its input is the sequence of elements $X = x_1,\ x_2,\ \ldots,\ x_p$, and its output is the sequence $Y = y_1,\ y_2,\ \ldots, y_p$, containing the prefixes.

CHAIN-PREFIX(X)

1 P_1 sends $y_1 = x_1$ to P_2
2 P_i **in parallel for** $i \leftarrow 2$ **to** $p - 1$
3 **for** $i \leftarrow 2$**to**$p - 1$
4 **do** gets y_{i-1} from P_{i-1}, then computes and stores $y_i \leftarrow y_{i-1} \oplus x_i$
stores $z_i = z_{p-1} \oplus x_p$, and sends z_i to P_{i+1}
5 P_a gets z_{p-1} from P_{p-1}, then computes and stores $y_a = y_{a-1} \oplus x_a$

Saying the truth, this is not a real parallel algorithm.

Theorem 15.18 *Algorithm* CHAIN-PREFIX *determines the prefixes of p elements using a chain $\mathcal{C}_p$ in $\Theta(p)$ time.*

Proof The cycle in lines 2–5 requires $\Theta(p)$ time, line 1 and line 6 requires $\Theta(1)$ time. ■

Since the prefixes can be determined in $O(p)$ time using a sequential processor, and $w(p, p,$ CHAIN-PREFIX$) = pW(p, p,$ CHAIN-PREFIX$) = \Theta(p^2)$, so CHAIN-PREFIX is not work-effective.

15.7.2. Prefix on square

An algorithm, similar to CHAIN-PREFIX, can be developed for a square too.

Let us consider a square of size $a \times a$. We need an ***indexing*** of the processors. There are many different indexing schemes, but for the next algorithm SQUARE-PREFIX sufficient is the one of the simplest solutions, the ***row-major indexing scheme***, where processor $P_{i,j}$ gets the index $a(i-1)+j$.

The input and the output are the same, as in the case of CHAIN-PREFIX.

The processors $P_{i-1)a+1,\ P_{(i-1)a+2)},\ \ldots P_{(i-1)a)+a}$ form the processor row $R_i (1 \leq i \leq a)$ and the processors $P_{a+j},\ P_{2a+j},\ \ldots P_{a(a-1)+j}$ form the processor column C_j $(1 \leq j \leq a)$. The input stored by the processors of row R_i is denoted by X_i, and the similar output is denoted by Y_i.

The algorithm works in 3 rounds. In the first round (lines 1–8) processor rows R_i $(1 \leq i \leq a)$ compute the row-local prefixes (working as processors of CHAIN-PREFIX). In the second round (lines 9–17) the column C_a computes the prefixes using the results of the first round, and the processors of this column P_{ja} $(1 \leq j \leq a-1)$ send the computed prefix to the neighbour $P_{(j+1)a)}$. Finally in the third round the rows R_i $(2 \leq i \leq a)$ determine the final prefixes.

SQUARE-PREFIX(X)

1 $P_{j,1}$ **in parallel for** $j \leftarrow 1$ **to** a
2 **do** sends $y_{j,1} = x_{j,1}$ to $P_{j,2}$
3 $P_{j,i}$ **in parallel for** $i \leftarrow 1$ **to** $a-1$
4 **for** $i \leftarrow 2$**to**$a-1$
5 **do** gets $y_{j,i-1}$ from $P_{j,i-1}$, then computes and
6 stores $y_{j,}i = y_{j,p-1} \oplus x_{j,p}$, and sends $y_{j,i}$ to $P_{j,i+1}$
7 $P_{j,a}$ **in parallel for** $j \leftarrow 1$ **to** a
8 **do** gets $y_{j,a-1}$ from $P_{j,a-1}$, then computes and stores $y_{1,a} = y_{1,a-1} \oplus x_{1,a}$
9 $P_{1,a}$ sends $y_{1,a}$ to $P_{2,a}$
10 $P_{j,a}$ **in parallel for** $j \leftarrow 2$ **to** $a-1$
11 **for** $j \leftarrow 2$**to**$a-1$
12 **do** gets $y_{j-1,a}$ from $P_{j-1,a}$, then computes and stores
stores $y_{j,a} = y_{j-1,a} \oplus y_{j,a}$, and sends $y_{j,a}$ to $P_{j+1,a}$
13 $P_{a,a}$ gets $y_{a-1,a}$ from $P_{a-1,a}$, then computes and stores $y_{a,a} = y_{a-1,a} \oplus y_{a,a}$
14 $P_{j,a}$ **in parallel for** $j \leftarrow 1$ **to** $a-1$
15 **do** send $y_{j,a}$ to $P_{j+1,a}$
16 $P_{j,a}$ **in parallel for** $j \leftarrow 2$ **to** a
17 **do** sends $y_{j,a}$ to $P_{j,a-1}$
18 $P_{j,i}$ **in parallel for** $i \leftarrow a-1$ **downto** 2
19 **for** $j \leftarrow 2$**to**a
20 **do** gets $y_{j,a}$ from $P_{j,i+1}$, then computes and
21 stores $y_{j,i} = y_{j,i+1} \oplus y_{j,i}$, and sends $y_{j,a}$ to $P_{j,i-1}$
22 $P_{j,1}$ **in parallel for** $j \leftarrow 2$ **to** $a-1$
23 **do** gets $y_{j,a}$ from $P_{j,2}$, then computes and stores $y_{j,1} = y_{j,a} \oplus y_{j,1}$

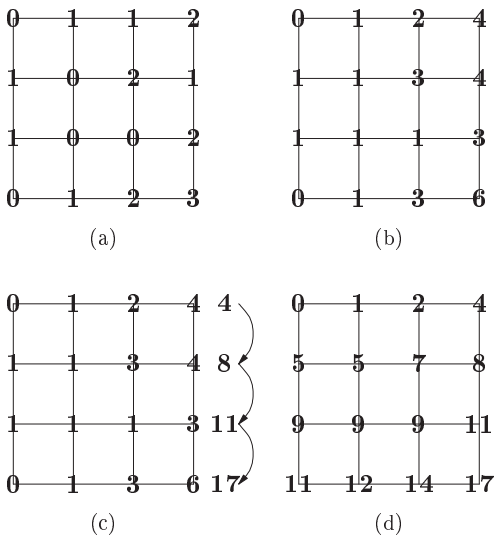


Figure 15.23. Prefix computation on square.

Theorem 15.19 *Algorithm* SQUARE-PREFIX *solves the prefix problem using a square of size $a \times a$, major row indexing in $3a + 2 = \Theta(a)$ time.*

Proof In the first round lines 1–2 contain 1 parallel operation, lines 3–6 require $a - 1$ operations, and line 8 again 1 opation, that is alltogether $a + 1$ operations. In a similar way in the third round lines 18–23 require $a + 1$ time units, and in round 2 lines 9–17 require a time units. The sum of the necessary time units is $3s + 2$. ■

Example 15.8 *Prefix computation on square of size* 4×4 Figure 15.23(a) shows 16 input elements. In the first round SQUARE-PREFIX computes the row-local prefixes, part (b) of the figure show the results. Then in the second round only the processors of the fourth column work, and determine the column-local prefixes – results are in part (c) of the figure. Finally in the third round algorithm determines the final results shown in part (d) of the figure.

Chapter notes

Basic sources of this chapter are for architectures and models the book of Leopold [220], and the book of Sima, Fountaine and Kacsuk [303], for parallel programming the book due to Kumar et al. [140] and [220], for parallel algorithms the books of Berman and Paul, [42] Cormen, Leiserson and Rivest [72], the book written by Horowitz, Sahni and Rajasekaran [167] and the book [176].

The website [324] contains the Top 500 list, a regularly updated survey of the most powerful computers worldwide [324]. It contains 42% clusters.

Described classifications of computers are proposed by Flynn [113], and Leopold

[220]. The Figures 15.1, 15.2, 15.3, 15.4, 15.5, 15.7 are taken from the book of Leopold [220], the program 15.6 from the book written by Gropp et al. [144].

The clusters are characterised using the book of Pfister [272], grids are presented on the base of the book and manuscript of Foster and Kellerman [117, 118].

With the problems of shared memory deal the book written by Hwang and Xu [171], the book due to Kleiman, Shah, and Smaalders [197], and the textbook of Tanenbaum and van Steen [315].

Details on concepts as tasks, processes and threads can be found in many textbook, e.g. in [302, 313]. Decomposition of the tasks into smaller parts is analysed by Tanenbaum and van Steen [315].

The laws concerning the speedup were described by Amdahl [15], Gustafson-Barsis [149] and Brent [48]. Kandemir, Ramanujam and Choudray review the different methods of the improvement of locality [186]. Wolfe [345] analyses in details the connection between the transformation of the data and the program code. In connection with code optimisation the book published by Kennedy and Allen [195] is a useful source.

The MPI programming model is presented according to Gropp, Snir, Nitzberg, and Lusk [144], while the base of the description of the OpenMP model is the paper due to Chandra, Dragum, Kohr, Dror, McDonald and Menon [60], further a review found on the internet [257].

Lewis and Berg [221] discuss pthreads, while Oaks and Wong [256] the Java threads in details. Description of *High Performance Fortran* can be found in the book Koelbel et al. [202]. Among others Wolfe [345] studied the parallelising compilers.

The concept of PRAM is due to Fortune and Wyllie and is known since 1978 [116]. BSP was proposed in 1990 by Valiant [334]. LogP has been suggested as an alternative of BSP by Culler et al. in 1993 [78]. QSM was introduced in 1999 by Gibbons, Matias and Ramachandran [131].

The majority of the pseudocode conventions used in Section 15.6 and the description of crossover points and comparison of different methods of matrix multiplication can be found in [73].

The Readers interested in further programming models, as skeletons, parallel functional programming, languages of coordination and parallel mobile agents, can find a detailed description in [220]. Further problems and parallel algorithms are analysed in the books of Leighton [215, 216] and in the chapter *Memory Management* of this book [29]. and in the book of Horowitz, Sahni and Rajasekaran [167]

Cost-optimal parallel merge is analysed by Wu and Olariu in [346]. New ideas (as the application of multiple comparisons to get a constant time sorting algoritm) of parallel sorting can be found in the paper of Garach, Golub, and Kruskal [126].

16. Systolic Systems

Systolic arrays probably constitute a perfect kind of special purpose computer. In their simplest appearance, they may provide only one operation, that is repeated over and over again. Yet, systolic arrays show an abundance of practice-oriented applications, mainly in fields dominated by iterative procedures: numerical mathematics, combinatorial optimisation, linear algebra, algorithmic graph theory, image and signal processing, speech and text processing, et cetera.

For a systolic array can be tailored to the structure of its one and only algorithm thus accurately! So that time and place of each executed operation are fixed once and for all. And communicating cells are permanently and directly connected, no switching required. The algorithm has in fact become hardwired. Systolic algorithms in this respect are considered to be ***hardware algorithms***.

Please note that the term *systolic algorithms* usually does not refer to a set of concrete algorithms for solving a single specific computational problem, as for instance *sorting*. And this is quite in contrast to terms like *sorting algorithms*. Rather, systolic algorithms constitute a special style of specification, programming, and computation. So algorithms from many different areas of application can be *systolic* in style. But probably not all well-known algorithms from such an area might be suited to systolic computation.

Hence, this chapter does not intend to present *all* systolic algorithms, nor will it introduce even the most important systolic algorithms from any field of application. Instead, with a few simple but typical examples, we try to lay the foundations for the Readers' general understanding of systolic algorithms.

The rest of this chapter is organised as follows: Section 16.1 shows some basic concepts of systolic systems by means of an introductory example. Section 16.2 explains how systolic arrays formally emerge from space-time transformations. Section 16.3 deals with input/output schemes. Section 16.4 is devoted to all aspects of control in systolic arrays. In Section 16.5 we study the class of linear systolic arrays, raising further questions.

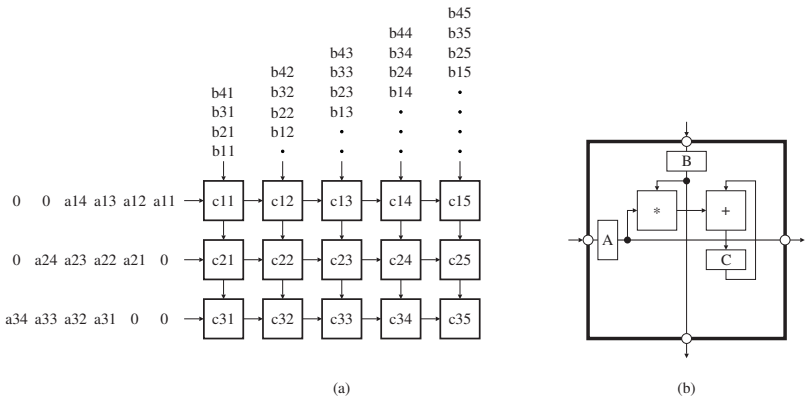


Figure 16.1. Rectangular systolic array for matrix product. **(a)** Array structure and input scheme. **(b)** Cell structure.

16.1. Basic concepts of systolic systems

The designation ***systolic*** follows from the operational principle of the systolic architecture. The systolic style is characterised by an intensive application of both *pipelining* and *parallelism*, controlled by a global and completely synchronous clock. ***Data streams*** pulsate rhythmically through the communication network, like streams of blood are driven from the heart through the veins of the body. Here, pipelining is not constrained to a single space axis but concerns *all data streams* possibly *moving in different directions* and *intersecting* in the cells of the systolic array.

A ***systolic system*** typically consists of a *host computer*, and the actual *systolic array*. Conceptionally, the host computer is of minor importance, just controlling the operation of the systolic array and supplying the data. The ***systolic array*** can be understood as a specialised network of cells rapidly performing data-intensive computations, supported by *massive parallelism*. A ***systolic algorithm*** is the program collaboratively executed by the cells of a systolic array.

Systolic arrays may appear very differently, but usually share a couple of key features: discrete time scheme, synchronous operation, regular (frequently two-dimensional) geometric layout, communication limited to directly neighbouring cells, and spartan control mechanisms.

In this section, we explain fundamental phenomena in context of systolic arrays, driven by a running example. A computational problem usually allows several solutions, each implemented by a specific systolic array. Among these, the most attractive designs (in whatever respect) may be very complex. Note, however, that in this educational text we are less interested in advanced solutions, but strive to present important concepts compactly and intuitively.

16.1.1. An introductory example: matrix product

Figure 16.1 shows a ***rectangular*** systolic array consisting of 15 ***cells*** for multiplying a $3 \times N$ matrix A by an $N \times 5$ matrix B. The *parameter* N is not reflected in the *structure* of this particular systolic array, but in the *input scheme* and the *running time* of the algorithm.

The input scheme depicted is based on the special choice of parameter $N = 4$. Therefore, Figure 16.1 gives a solution to the following problem instance:

$$A \cdot B = C \ ,$$

where

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix} \ ,$$

$$B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \end{pmatrix} \ ,$$

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} \end{pmatrix} \ ,$$

and

$$c_{ij} = \sum_{k=1}^{4} a_{ik} \cdot b_{kj} \qquad (1 \leq i \leq 3, 1 \leq j \leq 5) \ .$$

The cells of the systolic array can exchange data through ***links***, drawn as arrows between the cells in Figure 16.1(a). ***Boundary cells*** of the systolic array can also communicate with the *outside world*. All cells of the systolic array share a common ***connection pattern*** for communicating with their environment. The completely regular ***structure*** of the systolic array (placement and connection pattern of the cells) induces *regular data flows* along all connecting directions.

Figure 16.1(b) shows the ***internal structure*** of a cell. We find a *multiplier*, an *adder*, three *registers*, and four *ports*, plus some wiring between these units. Each ***port*** represents an interface to some external link that is attached to the cell. All our cells are of the same structure.

Each of the registers A, B, C can store a single data item. The designations of the registers are suggestive here, but arbitrary in principle. Registers A and B get their values from ***input ports***, shown in Figure 16.1(b) as small circles on the left resp. upper border of the cell.

The current values of registers A and B are used as operands of the multiplier and, at the same time, are passed through ***output ports*** of the cell, see the circles on the right resp. lower border. The result of the multiplication is supplied to the

adder, with the second operand originating from register C. The result of the addition eventually overwrites the past value of register C.

16.1.2. Problem parameters and array parameters

The 15 cells of the systolic array are organised as a rectangular pattern of three rows by five columns, exactly as with matrix C. Also, these dimensions directly correspond to the number of rows of matrix A and the number of columns of matrix B. The ***size of the systolic array***, therefore, *corresponds* to the *size of some data structures* for the problem to solve. If we had to multiply an $N_1 \times N_3$ matrix A by an $N_3 \times N_2$ matrix B in the general case, then we would need a systolic array with N_1 rows and N_2 columns.

The quantities N_1, N_2, N_3 are parameters of the problem to solve, because the number of operations to perform depends on each of them; they are thus ***problem parameters***. The size of the systolic array, in contrast, depends on the quantities N_1 and N_2, only. For this reason, N_1 and N_2 become also ***array parameters***, for this particular systolic array, whereas N_3 is *not* an array parameter.

Remark. For matrix product, we will see another systolic array in Section 16.2, with dimensions dependent on all three problem parameters N_1, N_2, N_3.

An $N_1 \times N_2$ systolic array as shown in Figure 16.1 would also permit to multiply an $M_1 \times M_3$ matrix A by an $M_3 \times M_2$ matrix B, where $M_1 \leq N_1$ and $M_2 \leq N_2$. This is important if we intend to use the same systolic array for the multiplication of matrices of varying dimensions. Then we would operate on a properly dimensioned rectangular subarray, only, consisting of M_1 rows and M_2 columns, and located, for instance, in the upper left corner of the complete array. The remaining cells would also work, but without any contribution to the solution of the whole problem; they should do no harm, of course.

16.1.3. Space coordinates

Now let's assume that we want to assign unique ***space coordinates*** to each cell of a systolic array, for characterising the geometric position of the cell relative to the whole array. In a *rectangular* systolic array, we simply can use the respective row and column numbers, for instance. The cell marked with c_{11} in Figure 16.1 thus would get the coordinates (1,1), the cell marked with c_{12} would get the coordinates (1,2), cell c_{21} would get (2,1), and so on. For the remainder of this section, we take space coordinates constructed in such a way for granted.

In principle it does not matter where the coordinate origin lies, where the axes are pointing to, which direction in space corresponds to the first coordinate, and which to the second. In the system presented above, the order of the coordinates has been chosen corresponding to the designation of the matrix components. Thus, the first coordinate stands for the rows numbered top to bottom from position 1, the second component stands for the columns numbered left to right, also from position 1.

Of course, we could have made a completely different choice for the coordinate system. But the presented system perfectly matches our particular systolic array: the

indices of a matrix element c_{ij} computed in a cell agree with the coordinates of this cell. The entered rows of the matrix A carry the same number as the first coordinate of the cells they pass; correspondingly for the second coordinate, concerning the columns of the matrix B. All links (and thus all passing data flows) are in parallel to some axis, and towards ascending coordinates.

It is not always so clear how expressive space coordinates can be determined; we refer to the systolic array from Figure 16.3(a) as an example. But whatsoever the coordinate system is chosen: it is important that the regular structure of the systolic array is obviously reflected in the coordinates of the cells. Therefore, almost always integral coordinates are used. Moreover, the coordinates of cells with minimum Euclidean distance should differ in one component, only, and then with distance 1.

16.1.4. Serialising generic operators

Each active cell (i, j) from Figure 16.1 computes exactly the element c_{ij} of the result matrix C. Therefore, the cell must evaluate the ***dot product***

$$\sum_{k=1}^{4} a_{ik} \cdot b_{kj} \ .$$

This is done iteratively: in each step, a product $a_{ik} \cdot b_{kj}$ is calculated and added to the current partial sum for c_{ij}. Obviously, the partial sum has to be *cleared*—or set to another initial value, if required—before starting the accumulation. Inspired by the classical notation of imperative programming languages, the general proceeding could be specified in pseudocode as follows:

MATRIX-PRODUCT(N_1, N_2, N_3)

```
1  for i ← 1 to N_1
2      do for j ← 1 to N_2
3             do c(i,j) ← 0
4                for k ← 1 to N_3
5                    do c(i,j) ← c(i,j) + a(i,k) · b(k,j)
6  return C
```

If $N_1 = N_2 = N_3 = N$, we have to perform N^3 multiplications, additions, and assignments, each. Hence the *running time* of this algorithm is of order $\Theta(N^3)$ for any sequential processor.

The sum operator $\sum$ is one of the so-called ***generic operators***, that combine an arbitrary number of operands. In the systolic array from Figure 16.1, all additions contributing to a particular sum are performed in the same cell. However, there are plenty of examples where the individual operations of a generic operator are spread over several cells—see, for instance, the systolic array from Figure 16.3.

Remark. Further examples of generic operators are: product, minimum, maximum, as well as the Boolean operators AND, OR, and EXCLUSIVE OR.

Thus, generic operators usually have to be ***serialised*** before the calculations to perform can be assigned to the cells of the systolic array. Since the distribution of

the individual operations to the cells is not unique, generic operators generally must be dealt with in another way than simple operators with fixed arity, as for instance the dyadic addition.

16.1.5. Assignment-free notation

Instead of using an imperative style as in algorithm MATRIX-PRODUCT, we better describe systolic programs by an ***assignment-free notation*** which is based on an *equational calculus*. Thus we avoid *side effects* and are able to *directly express parallelism*. For instance, we may be bothered about the reuse of the program variable $c(i,j)$ from algorithm MATRIX-PRODUCT. So, we replace $c(i,j)$ with a sequence of ***instances*** $c(i,j,k)$, that stand for the successive states of $c(i,j)$. This approach yields a so-called ***recurrence equation***. We are now able to state the general matrix product from algorithm MATRIX-PRODUCT by the following assignment-free expressions:

$$\begin{array}{ll}
\textit{input operations} & \\
c(i,j,0) = 0 & 1 \le i \le N_1, 1 \le j \le N_2 \ . \\
\textit{calculations} & \\
c(i,j,k) = c(i,j,k-1) + a(i,k) \cdot b(k,j) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \ . \\
\textit{output operations} & \\
c_{ij} = c(i,j,N_3) & 1 \le i \le N_1, 1 \le j \le N_2 \ .
\end{array} \tag{16.1}$$

System (16.1) explicitly describes the fine structure of the executed systolic algorithm. The first equation specifies all ***input data***, the third equation all ***output data***. The systolic array implements these equations by ***input/output operations***. Only the second equation corresponds to real calculations.

Each equation of the system is accompanied, on the right side, by a ***quantification***. The quantification states the set of values the ***iteration variables*** i and j (and, for the second equation, also k) should take. Such a set is called a ***domain***. The iteration variables i,j,k of the second equation can be combined in an ***iteration vector*** (i,j,k). For the input/output equations, the iteration vector would consist of the components i and j, only. To get a closed representation, we augment this vector by a third component k, that takes a fixed value. Inputs then are characterised by $k=0$, outputs by $k=N_3$. Overall we get the following system:

$$\begin{array}{ll}
\textit{input operations} & \\
c(i,j,k) = 0 & 1 \le i \le N_1, 1 \le j \le N_2, k = 0 \ . \\
\textit{calculations} & \\
c(i,j,k) = c(i,j,k-1) + a(i,k) \cdot b(k,j) & 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \ . \\
\textit{output operations} & \\
c_{ij} = c(i,j,k) & 1 \le i \le N_1, 1 \le j \le N_2, k = N_3 \ .
\end{array} \tag{16.2}$$

Note that although the domains for the input/output equations now are formally also of dimension 3, as a matter of fact they are only two-dimensional in the classical geometric sense.

16.1.6. Elementary operations

From equations as in system (16.2), we directly can infer the atomic entities to perform in the cells of the systolic array. We find these operations by instantiating each equation of the system with all points of the respective domain. If an equation contains several suboperations corresponding to one point of the domain, these are seen as a ***compound operation***, and are always processed together by the same cell in one working cycle.

In the second equation of system (16.2), for instance, we find the multiplication $a(i,k) \cdot b(k,j)$ and the successive addition $c(i,j,k) = c(i,j,k-1) + \cdots$. The corresponding ***elementary operations***—multiplication and addition—are indeed executed together as a *multiply-add* compound operation by the cell of the systolic array shown in Figure 16.1(b).

Now we can assign a designation to each elementary operation, also called *coordinates*. A straight-forward method to define suitable coordinates is provided by the iteration vectors (i,j,k) used in the quantifications.

Applying this concept to system (16.1), we can for instance assign the tuple of coordinates (i,j,k) to the calculation $c(i,j,k) = c(i,j,k-1) + a(i,k) \cdot b(k,j)$. The same tuple (i,j,k) is assigned to the input operation $c(i,j,k) = 0$, but with setting $k = 0$. By the way: all domains are disjoint in this example.

If we always use the iteration vectors as designations for the calculations and the input/output operations, there is no further need to distinguish between coordinates and iteration vectors. Note, however, that this decision also mandates that all operations belonging to a certain point of the domain together constitute a compound operation—even when they appear in different equations and possibly are not related. For simplicity, we always use the iteration vectors as coordinates in the sequel.

16.1.7. Discrete timesteps

The various elementary operations always happen in ***discrete timesteps*** in the systolic cells. All these timesteps driving a systolic array are of equal duration. Moreover, all cells of a systolic array work completely ***synchronous***, i.e., they all start and finish their respective communication and calculation steps at the same time. Successive timesteps controlling a cell seamlessly follow each other.

Remark. But haven't we learned from Albert Einstein that strict simultaneity is physically impossible? Indeed, all we need here are cells that operate almost simultaneously. Technically this is guaranteed by providing to all systolic cells a common ***clock signal*** that switches all registers of the array. Within the bounds of the usually achievable accuracy, the communication between the cells happens sufficiently synchronised, and thus no loss of data occurs concerning send and receive operations.

Therefore, it should be justified to assume a conceptional simultaneity for theoretical reasoning.

Now we can slice the physical time into units of a timestep, and number the timesteps consecutively. The origin on the time axis can be arbitrarily chosen, since time is running synchronously for all cells. A reasonable decision would be to take $t = 0$ as the time of the first input in any cell. Under this regime, the elementary compound operation of system (16.1) designated by (i, j, k) would be executed at time $i + j + k - 3$. On the other hand, it would be evenly justified to assign the time $i + j + k$ to the coordinates (i, j, k); because this change would only induce a global time shift by three time units.

So let us assume for the following that the execution of an instance (i, j, k) starts at time $i+j+k$. The first calculation in our example then happens at time $t = 3$, the last at time $t = N_1 + N_2 + N_3$. The *running time* thus amounts to $N_1 + N_2 + N_3 - 2$ timesteps.

16.1.8. External and internal communication

Normally, the data needed for calculation by the systolic array initially are not yet located inside the cells of the array. Rather, they must be infused into the array from the ***outside world***. The outside world in this case is a ***host computer***, usually a *scalar control processor* accessing a central *data storage*. The control processor, at the right time, fetches the necessary data from the storage, passes them to the systolic array in a suitable way, and eventually writes back the calculated results into the storage.

Each cell (i, j) must access the operands a_{ik} and b_{kj} during the timestep concerning index value k. But only the cells of the leftmost column of the systolic array from Figure 16.1 get the items of the matrix A directly as *input data* from the outside world. All other cells must be provided with the required values a_{ik} from a neighbouring cell. This is done via the horizontal *links* between neighbouring cells, see Figure 16.1(a). The item a_{ik} successively passes the cells $(i, 1)$, $(i, 2)$, ..., (i, N_2). Correspondingly, the value b_{kj} enters the array at cell $(1, j)$, and then flows through the vertical links, reaching the cells $(2, j)$, $(3, j)$, ... up to cell (N_1, j). An arrowhead in the figure shows in which *direction* the link is oriented.

Frequently, it is considered problematic to transmit a value over large distances within a single *timestep*, in a distributed or parallel architecture. Now suppose that, in our example, cell (i, j) got the value a_{ik} during timestep t from cell $(i, j - 1)$, or from the outside world. For the reasons described above, a_{ik} is not passed from cell (i, j) to cell $(i, j + 1)$ in the same timestep t, but one timestep later, i.e., at time $t+1$. This also holds for the values b_{kj}. The ***delay*** is visualised in the detail drawing of the cell from Figure 16.1(b): input data flowing through a cell always pass one register, and each passed register induces a delay of exactly one timestep.

Remark. For systolic architectures, it is mandatory that any path between two cells contains at least one register—even when forwarding data to a neighbouring cell, only. All registers in the cells are synchronously switched by the global clock signal of the systolic array. This results in the characteristic rhythmical traffic on all links of the systolic array. Because of the analogy with pulsating veins, the medical

term *systole* has been reused for the name of the concept.

To elucidate the delayed forwarding of values, we augment system (16.1) with further equations. Repeatedly *used* values like a_{ik} are represented by separate *instances*, one for each access. The result of this proceeding—that is very characteristic for the design of systolic algorithms—is shown as system (16.3).

$$
\begin{aligned}
&\textit{input operations} \\
&a(i,j,k) = a_{ik} && 1 \le i \le N_1, j = 0, 1 \le k \le N_3 \ , \\
&b(i,j,k) = b_{kj} && i = 0, 1 \le j \le N_2, 1 \le k \le N_3 \ , \\
&c(i,j,k) = 0 && 1 \le i \le N_1, 1 \le j \le N_2, k = 0 \ . \\
&\textit{calculations and forwarding} \\
&a(i,j,k) = a(i,j-1,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \ , \\
&b(i,j,k) = b(i-1,j,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \ , \\
&c(i,j,k) = c(i,j,k-1) \\
&\quad + a(i,j-1,k) \cdot b(i-1,j,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \ . \\
&\textit{output operations} \\
&c_{ij} = c(i,j,k) && 1 \le i \le N_1, 1 \le j \le N_2, k = N_3 \ .
\end{aligned}
\tag{16.3}
$$

Each of the partial sums $c(i,j,k)$ in the progressive evaluation of c_{ij} is calculated in a certain timestep, and then used only once, namely in the next timestep. Therefore, cell (i,j) must provide a register (named C in Figure 16.1(b)) where the value of $c(i,j,k)$ can be stored for one timestep. Once the old value is no longer needed, the register holding $c(i,j,k)$ can be overwritten with the new value $c(i,j,k+1)$. When eventually the dot product is completed, the register contains the value $c(i,j,N_3)$, that is the final result c_{ij}. Before performing any computation, the register has to be ***cleared***, i.e., preloaded with a zero value—or any other desired value.

In contrast, there is no need to store the values a_{ik} and b_{kj} permanently in cell (i,j). As we can learn from Figure 16.1(a), each row of the matrix A is delayed by one timestep with respect to the preceding row. And so are the columns of the matrix B. Thus the values $a(i,j-1,k)$ and $b(i-1,j,k)$ arrive at cell (i,j) exactly when the calculation of $c(i,j,k)$ is due. They are put to the registers A resp. B, then immediately fetched from there for the multiplication, and in the same cycle forwarded to the neighbouring cells. The values a_{ik} and b_{kj} are of no further use for cell (i,j) after they have been multiplied, and need not be stored there any longer. So A and B are overwritten with new values during the next timestep.

It should be obvious from this exposition that we urgently need to make economic use of the memory contained in a cell. Any calculation and any communication must be coordinated in space and time in such a way that storing of values is limited to the shortest-possible time interval. This goal can be achieved by immediately using and forwarding the received values. Besides the overall structure of the systolic array, choosing an appropriate *input/output scheme* and placing the corresponding number of *delays* in the cells essentially facilitates the desired coordination. Figure 16.1(b) in this respect shows the smallest possible delay by one timestep.

Geometrically, the input scheme of the example resulted from *skewing* the matrices A and B. Thereby some places in the ***input streams*** for matrix A became vacant and had to be filled with zero values; otherwise, the calculation of the c_{ij} would have been garbled. The input streams in length depend on the *problem parameter* N_3.

As can been seen in Figure 16.1, the items of matrix C are calculated ***stationary***, i.e., all additions contributing to an item c_{ij} happen in the same cell. ***Stationary variables*** don't move at all during the calculation in the systolic array. Stationary results eventually must be forwarded to a ***border*** of the array in a supplementary action for getting delivered to the outside world. Moreover, it is necessary to initialise the register for item c_{ij}. Performing these extra tasks requires a high expenditure of runtime and hardware. We will further study this problem in Section 16.4.

16.1.9. Pipelining

The characteristic operating style with globally synchronised discrete timesteps of equal duration and the strict separation in time of the cells by registers suggest systolic arrays to be special cases of *pipelined* systems. Here, the registers of the cells correspond to the well-known *pipeline registers*. However, classical pipelines come as linear structures, only, whereas systolic arrays frequently extend into more spatial dimensions—as visible in our example. A *multi-dimensional systolic array* can be regarded as a set of interconnected linear pipelines, with some justification. Hence it should be apparent that basic properties of one-dimensional pipelining also apply to multi-dimensional systolic arrays.

A typical effect of pipelining is the reduced ***utilisation*** at startup and during shut-down of the operation. Initially, the pipe is empty, no pipeline stage active. Then, the first stage receives data and starts working; all other stages are still idle. During the next timestep, the first stage passes data to the second stage and itself receives new data; only these two stages do some work. More and more stages become active until all stages process data in every timestep; the pipeline is now fully utilised for the first time. After a series of timesteps at maximum load, with duration dependent on the length of the data stream, the input sequence ceases; the first stage of the pipeline therefore runs out of work. In the next timestep, the second stage stops working, too. And so on, until eventually all stages have been fallen asleep again. Phases of reduced activity diminish the average performance of the whole pipeline, and the relative contribution of this drop in productivity is all the worse, the more stages the pipeline has in relation to the length of the data stream.

We now study this phenomenon to some depth by analysing the two-dimensional systolic array from Figure 16.1. As expected, we find a lot of idling cells when starting or finishing the calculation. In the first timestep, only cell $(1, 1)$ performs some useful work; all other cells in fact do calculations that work like null operations—and that's what they are supposed to do in this phase. In the second timestep, cells $(1, 2)$ and $(2, 1)$ come to real work, see Figure 16.2(a). Data is flooding the array until eventually all cells are doing work. After the last true data item has left cell $(1, 1)$, the latter is

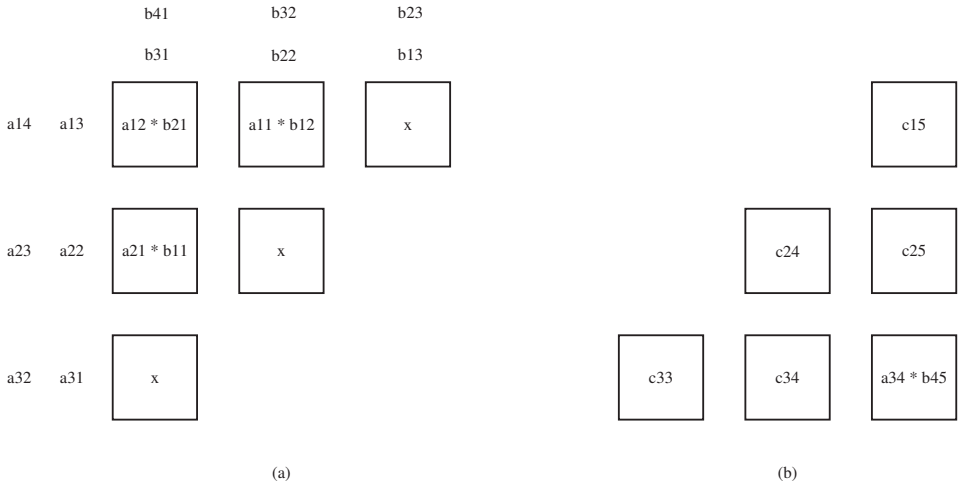


Figure 16.2. Two *snapshots* for the systolic array from Figure 16.1.

no longer contributing to the calculation but merely reproduces the finished value of c_{11}. Step by step, more and more cells drop off. Finally, only cell (N_1, N_2) makes a last necessary computation step; Figure 16.2(b) shows this concluding timestep.

Exercises

16.1-1 What must be changed in the input scheme from Figure 16.1(a) to multiply a 2×6 matrix by a 6×3 matrix on the same systolic array? Could the calculations be organised such that the result matrix would emerge in the lower right corner of the systolic array?

16.1-2 Why is it necessary to clear spare slots in the input streams for matrix A, as shown in Figure 16.1? Why haven't we done the same for matrix B also?

16.1-3 If the systolic array from Figure 16.1 should be interpreted as a pipeline: how many stages would you suggest to adequately describe the behaviour?

16.2. Space-time transformation and systolic arrays

Although the approach taken in the preceding section should be sufficient for a basic understanding of the topic, we have to work harder to describe and judge the properties of systolic arrays in a quantitative and precise way. In particular the solution of *parametric problems* requires a solid mathematical framework. So, in this section, we study central concepts of a formal theory on *uniform algorithms*, based on *linear transformations*.

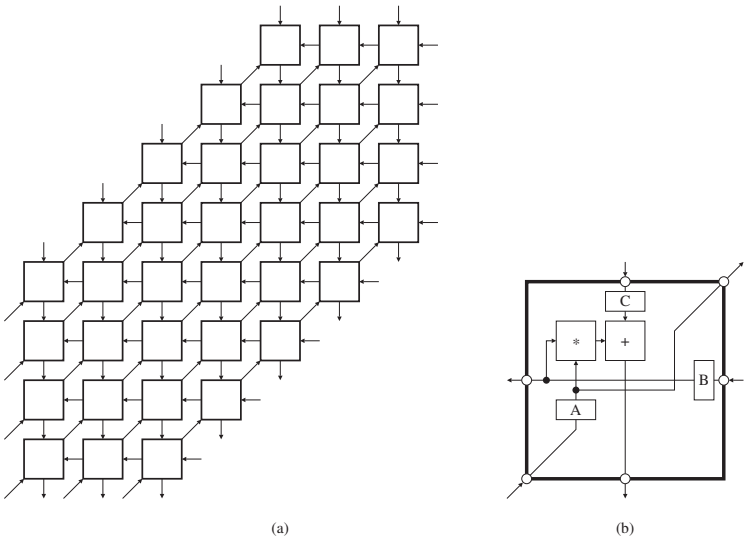


Figure 16.3. Hexagonal systolic array for matrix product. **(a)** Array structure and principle of the data input/output. **(b)** Cell structure.

16.2.1. Further example: matrix product without stationary variables

System (16.3) can be computed by a multitude of other systolic arrays, besides that from Figure 16.1. In Figure 16.3, for example, we see such an alternative systolic array. Whereas the same function is evaluated by both architectures, the appearance of the array from Figure 16.3 is very different:

- The number of cells now is considerably larger, altogether 36, instead of 15.
- The shape of the array is ***hexagonal***, instead of rectangular.
- Each cell now has three input ports and three output ports.
- The input scheme is clearly different from that of Figure 16.1(a).
- And finally: the matrix C here also flows through the whole array.

The cell structure from Figure 16.3(b) at first view does not appear essentially distinguished from that in Figure 16.1(b). But the differences matter: there are no *cyclic paths* in the new cell, thus *stationary variables* can no longer appear. Instead, the cell is provided with three input ports and three output ports, passing items of all three matrices through the cell. The direction of communication at the ports on the right and left borders of the cell has changed, as well as the assignment of the matrices to the ports.

16.2.2. The space-time transformation as a global view

How system (16.3) is related to Figure 16.3? No doubt that you were able to fully understand the operation of the systolic array from Section 16.1 without any special

aid. But for the present example this is considerably more difficult—so now you may be sufficiently motivated for the use of a mathematical formalism.

We can assign two fundamental measures to each elementary operation of an algorithm for describing the execution in the systolic array: the time *when* the operation is performed, and the position of the cell *where* the operation is performed. As will become clear in the sequel, after fixing the so-called *space-time transformation* there are hardly any degrees of freedom left for further design: practically all features of the intended systolic array strictly follow from the chosen space-time transformation.

As for the systolic array from Figure 16.1, the execution of an instance (i, j, k) in the systolic array from Figure 16.3 happens at time $t = i + j + k$. We can represent this expression as the dot product of a ***time vector***

$$\pi = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \tag{16.4}$$

by the iteration vector

$$v = \begin{pmatrix} i & j & k \end{pmatrix} , \tag{16.5}$$

hence

$$t = \pi \cdot v \; ; \tag{16.6}$$

so in this case

$$t = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} = i + j + k \; . \tag{16.7}$$

The space coordinates $z = (x, y)$ of the executed operations in the example from Figure 16.1 can be inferred as $z = (i, j)$ from the iteration vector $v = (i, j, k)$ according to our decision in Subsection 16.1.3. The chosen map is a ***projection*** of the space $\mathbb{R}^3$ along the k axis. This linear map can be described by a ***projection matrix***

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \; . \tag{16.8}$$

To find the space coordinates, we multiply the projection matrix P by the iteration vector v, written as

$$z = P \cdot v \; . \tag{16.9}$$

The ***projection direction*** can be represented by any vector u perpendicular to all rows of the projection matrix,

$$P \cdot u = \vec{0} \; . \tag{16.10}$$

For the projection matrix P from (16.8), one of the possible ***projection vectors*** would be $u = (0, 0, 1)$.

Projections are very popular for describing the space coordinates when designing a systolic array. Also in our example from Figure 16.3(a), the space coordinates are generated by projecting the iteration vector. Here, a feasible projection matrix is given by

$$P = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \; . \tag{16.11}$$

A corresponding projection vector would be $u = (1, 1, 1)$.

We can combine the projection matrix and the time vector in a matrix T, that fully describes the ***space-time transformation***,

$$\begin{pmatrix} z \\ t \end{pmatrix} = \begin{pmatrix} P \\ \pi \end{pmatrix} \cdot v = T \cdot v \ . \tag{16.12}$$

The first and second rows of T are constituted by the projection matrix P, the third row by the time vector π.

For the example from Figure 16.1, the matrix T giving the space-time transformation reads as

$$T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \ ; \tag{16.13}$$

for the example from Figure 16.3 we have

$$T = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \ . \tag{16.14}$$

Space-time transformations may be understood as a *global view* to the systolic system. Applying a space-time transformation—that is linear, here, and described by a matrix T—to a system of recurrence equations directly yields the external features of the systolic array, i.e., its ***architecture***—consisting of space coordinates, connection pattern, and cell structure.

Remark. Instead of purely linear maps, we alternatively may consider general affine maps, additionally providing a translative component, $T \cdot v + h$. Though as long as we treat all iteration vectors with a common space-time transformation, affine maps are not really required.

16.2.3. Parametric space coordinates

If the domains are numerically given and contain few points in particular, we can easily calculate the concrete set of space coordinates via equation (16.9). But when the domains are specified parametrically as in system (16.3), the positions of the cells must be determined by *symbolic evaluation*. The following explanation especially dwells on this problem.

Suppose that each cell of the systolic array is represented geometrically by a point with space coordinates $z = (x, y)$ in the two-dimensional space $\mathbb{R}^2$. From each iteration vector v of the domain S, by equation (16.9) we get the space coordinates z of a certain processor, $z = P \cdot v$: the operations denoted by v are projected onto cell z. The set $P(S) = \{P \cdot v : v \in S\}$ of space coordinates states the positions of all cells in the systolic array necessary for correct operation.

To our advantage, we normally use domains that can be described as the set of all integer points inside a convex region, here a subset of $\mathbb{R}^3$—called ***dense convex domains***. The convex hull of such a domain with a finite number of domain points is a *polytope*, with domain points as vertices. Polytopes map to polytopes again by arbitrary linear transformations. Now we can make use of the fact that each

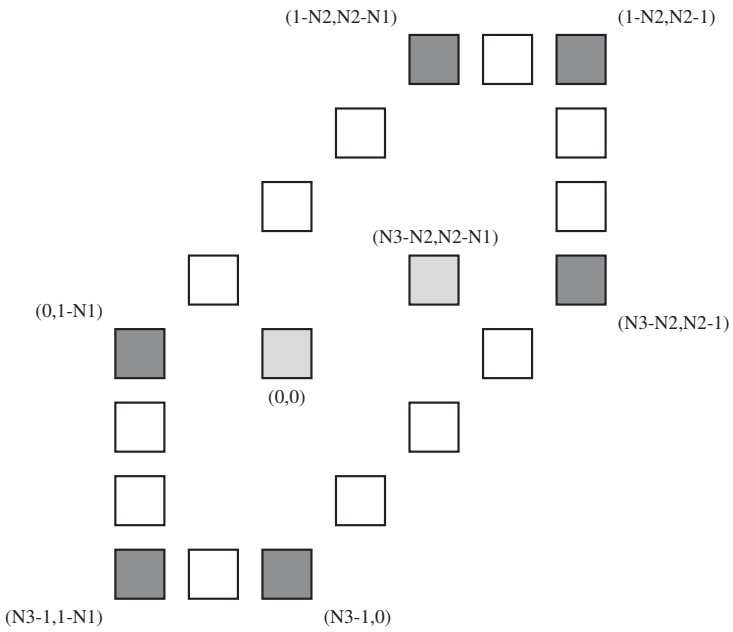


Figure 16.4. Image of a rectangular domain under projection. Most interior points have been suppressed for clarity. Images of previous vertex points are shaded.

projection is a linear transformation. Vertices of the destination polytope then are images of vertices of the source polytope.

Remark. But not all vertices of a source polytope need to be projected to vertices of the destination polytope, see for instance Figure 16.4.

When projected by an integer matrix P, the *lattice* $\mathbb{Z}^3$ maps to the lattice $\mathbb{Z}^2$ if P can be extended by an integer time vector π to a *unimodular*space-time matrix T. Practically any dense convex domain, apart from some exceptions irrelevant to usual applications, thereby maps to another dense convex set of space coordinates, that is completely characterised by the vertices of the hull polytope. To determine the *shape* and the *size* of the systolic array, it is therefore sufficient to apply the matrix P to the vertices of the convex hull of S.

Remark. Any square integer matrix with determinant ± 1 is called ***unimodular***. Unimodular matrices have unimodular inverses.

We apply this method to the integer domain

$$S = [1, N_1] \times [1, N_2] \times [1, N_3] \tag{16.15}$$

from system (16.3). The vertices of the convex hull here are

$$\begin{aligned} &(1,1,1), (N_1,1,1), (1,N_2,1), (1,1,N_3), \\ &(1,N_2,N_3), (N_1,1,N_3), (N_1,N_2,1), (N_1,N_2,N_3)\ . \end{aligned} \tag{16.16}$$

For the projection matrix P from (16.11), the vertices of the corresponding image

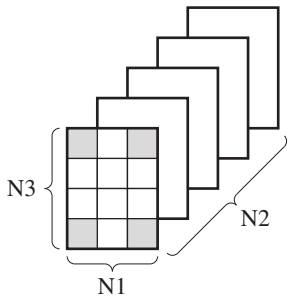


Figure 16.5. Partitioning of the space coordinates.

have the positions

$$\begin{aligned}&(N_3-1,0),(N_3-1,1-N_1),(0,1-N_1)\ ,\\&(1-N_2,N_2-N_1),(1-N_2,N_2-1),(N_3-N_2,N_2-N_1)\ .\end{aligned}\tag{16.17}$$

Since S has eight vertices, but the image $P(S)$ only six, it is obvious that two vertices of S have become *interior points* of the image, and thus are of no relevance for the size of the array; namely the vertices $(1,1,1)$ and (N_1,N_2,N_3). This phenomenon is sketched in Figure 16.4.

The settings $N_1=3$, $N_2=5$, and $N_3=4$ yield the vertices (3,0), (3,-2), (0,-2), (-4,2), (-4,4), and (-1,4). We see that space coordinates in principle can be negative. Moreover, the choice of an origin—that here lies in the interior of the polytope—might not always be obvious.

As the image of the projection, we get a systolic array with *hexagonal* shape and parallel opposite borders. On these, we find N_1, N_2, and N_3 integer points, respectively; cf. Figure 16.5. Thus, as opposed to our first example, *all* problem parameters here are also array parameters.

The area function of this region is of order $\Theta(N_1\cdot N_2+N_1\cdot N_3+N_2\cdot N_3)$, and thus depends on all three matrix dimensions. So this is quite different from the situation in Figure 16.1(a), where the area function—for the same problem—is of order $\Theta(N_1\cdot N_2)$.

Improving on this approximate calculation, we finally count the exact number of cells. For this process, it might be helpful to partition the entire region into subregions for which the number of cells comprised can be easily determined; see Figure 16.5. The points (0,0), $(N_3-1,0)$, $(N_3-1,1-N_1)$, and $(0,1-N_1)$ are the vertices of a rectangle with $N_1\cdot N_3$ cells. If we translate this point set up by N_2-1 cells and right by N_2-1 cells, we exactly cover the whole region. Each shift by one cell up and right contributes just another N_1+N_3-1 cells. Altogether this yields $N_1\cdot N_3+(N_2-1)\cdot(N_1+N_3-1)=N_1\cdot N_2+N_1\cdot N_3+N_2\cdot N_3-(N_1+N_2+N_3)+1$ cells.

For $N_1=3$, $N_2=5$, and $N_3=4$ we thereby get a number of 36 cells, as we have already learned from Figure 16.3(a).

16.2.4. Symbolically deriving the running time

The running time of a systolic algorithm can be symbolically calculated by an approach similar to that in Subsection 16.2.3. The time transformation according to formula (16.6) as well is a linear map. We find the timesteps of the first and the last calculations as the minimum resp. maximum in the set $\pi(S) = \{\pi \cdot v : v \in S\}$ of execution timesteps. Following the discussion above, it thereby suffices to vary v over the vertices of the convex hull of S.

The ***running time*** is then given by the formula

$$t_\Sigma = 1 + \max P(S) - \min P(S) \ . \tag{16.18}$$

Adding one is mandatory here, since the first as well as the last timestep belong to the calculation.

For the example from Figure 16.3, the vertices of the polytope as enumerated in (16.16) are mapped by (16.7) to the set of images

$$\{3, 2+N_1, 2+N_2, 2+N_3, 1+N_1+N_2, 1+N_1+N_3, 1+N_2+N_3, N_1+N_2+N_3\} \ .$$

With the basic assumption $N_1, N_2, N_3 \geq 1$, we get a minimum of 3 and a maximum of $N_1 + N_2 + N_3$, thus a running time of $N_1 + N_2 + N_3 - 2$ timesteps, as for the systolic array from Figure 16.1—no surprise, since the domains and the time vectors agree.

For the special problem parameters $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$, a running time of $12 - 3 + 1 = 10$ timesteps can be derived.

If $N_1 = N_2 = N_3 = N$, the systolic algorithm shows a running time of order $\Theta(N)$, using $\Theta(N^2)$ systolic cells.

16.2.5. How to unravel the communication topology

The ***communication topology*** of the systolic array is induced by applying the space-time transformation to the *data dependences* of the algorithm. Each data dependence results from a direct use of a variable instance to calculate another instance of the same variable, or an instance of another variable.

Remark. In contrast to the general situation where a data dependence analysis for imperative programming languages has to be performed by highly optimising compilers, data dependences here always are *flow dependences*. This is a direct consequence from the assignment-free notation employed by us.

The ***data dependences*** can be read off the quantified equations in our assignment-free notation by comparing their right and left sides. For example, we first analyse the equation $c(i,j,k) = c(i,j,k-1) + a(i,j-1,k) \cdot b(i-1,j,k)$ from system (16.3).

The value $c(i,j,k)$ is calculated from the values $c(i,j,k-1)$, $a(i,j-1,k)$, and $b(i-1,j,k)$. Thus we have a ***data flow*** from $c(i,j,k-1)$ to $c(i,j,k)$, a data flow from $a(i,j-1,k)$ to $c(i,j,k)$, and a data flow from $b(i-1,j,k)$ to $c(i,j,k)$.

All properties of such a data flow that matter here can be covered by a ***dependence vector***, which is the iteration vector of the calculated variable instance minus the iteration vector of the correspondingly used variable instance.

The iteration vector for $c(i,j,k)$ is (i,j,k); that for $c(i,j,k-1)$ is $(i,j,k-1)$. Thus, as the difference vector, we find

$$d_C = \begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} i \\ j \\ k-1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} . \tag{16.19}$$

Correspondingly, we get

$$d_A = \begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} i \\ j-1 \\ k \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{16.20}$$

and

$$d_B = \begin{pmatrix} i \\ j \\ k \end{pmatrix} - \begin{pmatrix} i-1 \\ j \\ k \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} . \tag{16.21}$$

In the equation $a(i,j,k) = a(i,j-1,k)$ from system (16.3), we cannot directly recognise which is the calculated variable instance, and which is the used variable instance. This example elucidates the difference between *equations* and *assignments*. When fixing that $a(i,j,k)$ should follow from $a(i,j-1,k)$ by a ***copy operation***, we get the same dependence vector d_A as in (16.20). Correspondingly for the equation $b(i,j,k) = b(i-1,j,k)$.

A variable instance with iteration vector v is calculated in cell $P \cdot v$. If for this calculation another variable instance with iteration vector v' is needed, implying a data dependence with dependence vector $d = v - v'$, the used variable instance is provided by cell $P \cdot v'$. Therefore, we need a communication from cell $z' = P \cdot v'$ to cell $z = P \cdot v$. In systolic arrays, all communication has to be via direct static links between the communicating cells. Due to the linearity of the transformation from (16.9), we have $z - z' = P \cdot v - P \cdot v' = P \cdot (v - v') = P \cdot d$.

If $P \cdot d = \vec{0}$, communication happens exclusively inside the calculating cell, i.e., in time, only—and not in space. Passing values in time is via registers of the calculating cell.

Whereas for $P \cdot d \neq \vec{0}$, a communication between different cells is needed. Then a link along the ***flow direction*** $P \cdot d$ must be provided from/to all cells of the systolic array. The vector $-P \cdot d$, oriented in counter flow direction, leads from space point z to space point z'.

If there is more than one dependence vector d, we need an appropriate *link* for each of them at every cell. Take for example the formulas (16.19), (16.20), and (16.21) together with (16.11), then we get $P \cdot d_A = (-1,1)$, $P \cdot d_B = (0,-1)$, and $P \cdot d_C = (1,0)$. In Figure 16.3(a), terminating at every cell, we see three links corresponding to the various vectors $P \cdot d$. This results in a ***hexagonal communication topology***—instead of the ***orthogonal communication topology*** from the first example.

16.2.6. Inferring the structure of the cells

Now we apply the space-related techniques from Subsection 16.2.5 to time-related questions. A variable instance with iteration vector v is calculated in timestep $\pi \cdot v$.

If this calculation uses another variable instance with iteration vector v', the former had been calculated in timestep $\pi \cdot v'$. Hence communication corresponding to the dependence vector $d = v - v'$ must take exactly $\pi \cdot v - \pi \cdot v'$ timesteps.

Since (16.6) describes a linear map, we have $\pi \cdot v - \pi \cdot v' = \pi \cdot (v - v') = \pi \cdot d$. According to the systolic principle, each communication must involve at least one register. The dependence vectors d are fixed, and so the choice of a time vector π is constrained by

$$\pi \cdot d \geq 1 \ . \tag{16.22}$$

In case $P \cdot d = \vec{0}$, we must provide ***registers*** for stationary variables in all cells. But each register is overwritten with a new value in every timestep. Hence, if $\pi \cdot d \geq 2$, the old value must be carried on to a further register. Since this is repeated for $\pi \cdot d$ timesteps, the cell needs exactly $\pi \cdot d$ registers per stationary variable. The values of the stationary variable successively pass all these registers before eventually being used. If $P \cdot d \neq \vec{0}$, the transport of values analogously goes by $\pi \cdot d$ registers, though these are not required to belong all to the same cell.

For each dependence vector d, we thus need an appropriate number of registers. In Figure 16.3(b), we see three input ports at the cell, corresponding to the dependence vectors d_A, d_B, and d_C. Since for these we have $P \cdot d \neq \vec{0}$. Moreover, $\pi \cdot d = 1$ due to (16.7) and (16.4). Thus, we need one register per dependence vector. Finally, the regularity of system (16.3) forces three output ports for every cell, opposite to the corresponding input ports.

Good news: we can infer in general that each cell needs only a few registers, because the number of dependence vectors d is statically bounded with a system like (16.3), and for each of the dependence vectors the amount of registers $\pi \cdot d$ has a fixed and usually small value.

The three input and output ports at every cell now permit the use of three moving matrices. Very differently from Figure 16.1, a dot product $\sum_{k=1}^{4} a_{ik} \cdot b_{kj}$ here is not calculated within a single cell, but dispersed over the systolic array. As a prerequisite, we had to dissolve the sum into a sequence of single additions. We call this principle a ***distributed generic operator***.

Apart from the three input ports with their registers, and the three output ports, Figure 16.3(b) shows a multiplier chained to an adder. Both units are induced in each cell by applying the transformation (16.9) to the domain S of the equation $c(i,j,k) = c(i,j,k-1) + a(i,j-1,k) \cdot b(i-1,j,k)$ from system (16.3). According to this equation, the addition has to follow the calculation of the product, so the order of the hardware operators as seen in Figure 16.3(b) is implied.

The source cell for each of the used operands follows from the projection of the corresponding dependence vector. Here, variable $a(i, j-1, k)$ is related to the dependence vector $d_A = (0,1,0)$. The projection $P \cdot d_A = (-1,1)$ constitutes the *flow direction* of matrix A. Thus the value to be used has to be expected, as observed by the calculating cell, in opposite direction $(1,-1)$, in this case from the port in the lower left corner of the cell, passing through register A. All the same, $b(i-1,j,k)$ comes from the right via register B, and $c(i,j,k-1)$ from above through register C. The calculated values $a(i,j,k)$, $b(i,j,k)$, and $c(i,j,k)$ are output into the opposite directions through the appropriate ports: to the upper right, to the left, and downwards.

If alternatively we use the projection matrix P from (16.8), then for d_C we get the direction $(0,0)$. The formula $\pi \cdot d_C = 1$ results in the requirement of exactly one register C for each item of the matrix C. This register provides the value $c(i,j,k-1)$ for the calculation of $c(i,j,k)$, and after this calculation receives the value $c(i,j,k)$. All this reasoning matches with the cell from Figure 16.1(b). Figure 16.1(a) correspondingly shows no links for matrix C between the cells: for the matrix is *stationary*.

Exercises

16.2-1 Each projection vector u induces several corresponding projection matrices P.

a. Show that

$$P = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \end{pmatrix}$$

also is a projection matrix fitting with projection vector $u = (1,1,1)$.

b. Use this projection matrix to transform the domain from system (16.3).

c. The resulting space coordinates differ from that in Subsection 16.2.3. Why, in spite of this, both point sets are topologically equivalent?

d. Analyse the cells in both arrangements for common and differing features.

16.2-2 Apply all techniques from Section 16.2 to system (16.3), employing a space-time matrix

$$T = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

16.3. Input/output schemes

In Figure 16.3(a), the *input/output scheme* is only sketched by the *flow directions* for the matrices A, B, C. The necessary details to understand the input/output operations are now provided by Figure 16.6.

The ***input/output scheme*** in Figure 16.6 shows some new phenomena when compared with Figure 16.1(a). The input and output cells belonging to any matrix are no longer threaded all on a single straight line; now, for each matrix, they lie along two adjacent *borders*, that additionally may differ in the number of links to the outside world. The data structures from Figure 16.6 also differ from that in Figure 16.1(a) in the angle of inclination. Moreover, the matrices A and B from Figure 16.6 arrive at the *boundary cells* with only one third of the *data rate*, compared to Figure 16.1(a).

Spending some effort, even here it might be possible in principle to construct—item by item—the appropriate input/output scheme fitting the present systolic array. But it is much more safe to apply a formal derivation. The following subsections are devoted to the presentation of the various methodical steps for achieving our goal.

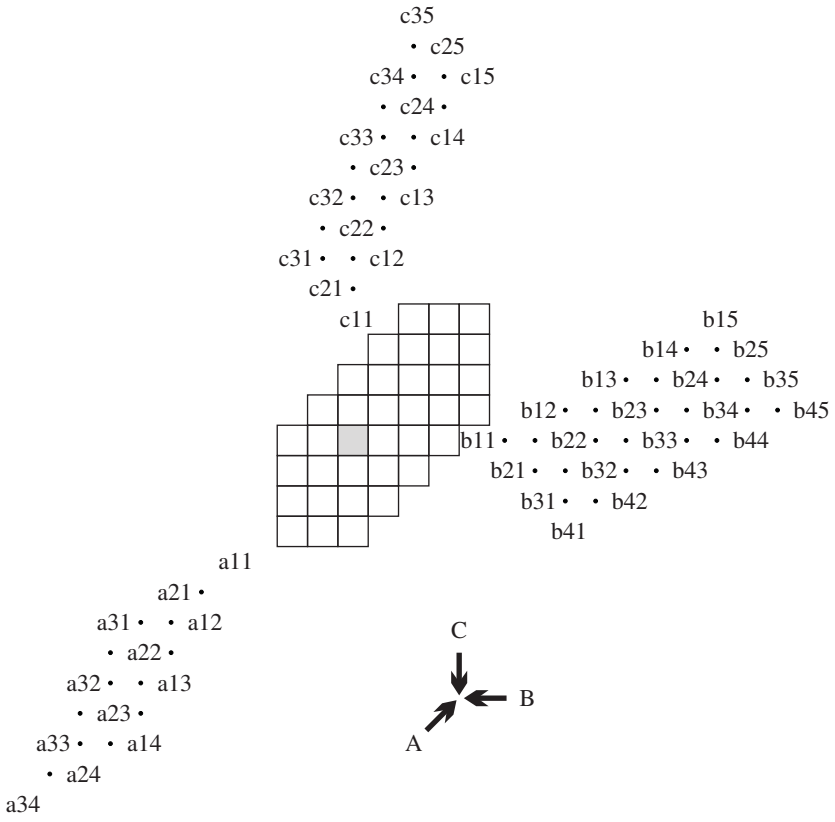


Figure 16.6. Detailed input/output scheme for the systolic array from Figure 16.3(a).

16.3.1. From data structure indices to iteration vectors

First, we need to construct a formal relation between the abstract data structures and the concrete variable instances in the assignment-free representation.

Each item of the matrix A can be characterised by a row index i and a column index k. These ***data structure indices*** can be comprised in a ***data structure vector*** $w = (i, k)$. Item a_{ik} in system (16.3) corresponds to the instances $a(i, j, k)$, with any j. The coordinates of these instances all lie on a line along direction $q = (0, 1, 0)$ in space $\mathbb{R}^3$. Thus, in this case, the formal change from data structure vector (i, k) to coordinates (i, j, k) can be described by the transformation

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ k \end{pmatrix} + j \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} . \tag{16.23}$$

In system (16.3), the coordinate vector (i, j, k) of every variable instance equals the iteration vector of the domain point representing the calculation of this variable instance. Thus we also may interpret formula (16.23) as a relation between *data structure vectors* and *iteration vectors*. Abstractly, the desired iteration vectors v

can be inferred from the data structure vector w by the formula

$$v = H \cdot w + \lambda \cdot q + p \; . \tag{16.24}$$

The affine vector p is necessary in more general cases, though always null in our example.

Because of $b(i,j,k) = b_{kj}$, the representation for matrix B correspondingly is

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} k \\ j \end{pmatrix} + i \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \; . \tag{16.25}$$

Concerning matrix C, each variable instance $c(i,j,k)$ may denote a different value. Nevertheless, all instances $c(i,j,k)$ to a fixed index pair (i,j) can be regarded as belonging to the same matrix item c_{ij}, since they all stem from the serialisation of the sum operator for the calculation of c_{ij}. Thus, for matrix C, following formula (16.24) we may set

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + k \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \; . \tag{16.26}$$

16.3.2. Snapshots of data structures

Each of the three matrices A, B, C is generated by two directions with regard to the *data structure indices*: along a row, and along a column. The difference vector (0,1) thereby describes a move from an item to the next item of the same row, i.e., in the next column: $(0,1) = (x, y+1) - (x,y)$. Correspondingly, the difference vector (1,0) stands for sliding from an item to the next item in the same column and next row: $(1,0) = (x+1, y) - (x,y)$.

Input/output schemes of the appearance shown in Figures 16.1(a) and 16.6 denote ***snapshots***: all positions of data items depicted, with respect to the entire systolic array, are related to a common timestep.

As we can notice from Figure 16.6, the rectangular shapes of the abstract data structures are mapped to parallelograms in the snapshot, due to the linearity of the applied space-time transformation. These parallelograms can be described by difference vectors along their borders, too.

Next we will translate difference vectors Δw from data structure vectors into spatial difference vectors Δz for the snapshot. Therefore, by choosing the parameter λ in formula (16.24), we pick a pair of iteration vectors v, v' that are mapped to the same timestep under our space-time transformation. For the moment it is not important which concrete timestep we thereby get. Thus, we set up

$$\pi \cdot v = \pi \cdot v' \quad \text{with} \quad v = H \cdot w + \lambda \cdot q + p \quad \text{and} \quad v' = H \cdot w' + \lambda' \cdot q + p \; , \tag{16.27}$$

implying

$$\pi \cdot H \cdot (w - w') + (\lambda - \lambda') \cdot \pi \cdot q = 0 \; , \tag{16.28}$$

and thus

$$\Delta\lambda = (\lambda - \lambda') = \frac{-\pi \cdot H \cdot (w - w')}{\pi \cdot q} \; . \tag{16.29}$$

Due to the linearity of all used transformations, the wanted spatial difference vector Δz hence follows from the difference vector of the data structure $\Delta w = w - w'$ as

$$\Delta z = P \cdot \Delta v = P \cdot H \cdot \Delta w + \Delta\lambda \cdot P \cdot q \; , \tag{16.30}$$

or

$$\Delta z = P \cdot H \cdot \Delta w - \frac{\pi \cdot H \cdot \Delta w}{\pi \cdot q} \cdot P \cdot q \; . \tag{16.31}$$

With the aid of formula (16.31), we now can determine the spatial difference vectors Δz for matrix A. As mentioned above, we have

$$H = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}, \quad q = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad P = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix}, \quad \pi = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \; .$$

Noting $\pi \cdot q = 1$, we get

$$\Delta z = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \cdot \Delta w + \Delta\lambda \cdot \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad \text{with} \quad \Delta\lambda = -\begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \Delta w \; .$$

For the rows, we have the difference vector $\Delta w = (0, 1)$, yielding the spatial difference vector $\Delta z = (2, -1)$. Correspondingly, from $\Delta w = (1, 0)$ for the columns we get $\Delta z = (1, -2)$. If we check with Figure 16.6, we see that the rows of A in fact run along the vector $(2, -1)$, the columns along the vector $(1, -2)$.

Similarly, we get $\Delta z = (-1, 2)$ for the rows of B, and $\Delta z = (1, 1)$ for the columns of B; as well as $\Delta z = (-2, 1)$ for the rows of C, and $\Delta z = (-1, -1)$ for the columns of C.

Applying these instruments, we are now able to reliably generate appropriate input/output schemes—although separately for each matrix at the moment.

16.3.3. Superposition of input/output schemes

Now, the shapes of the matrices A, B, C for the snapshot have been fixed. But we still have to adjust the matrices relative to the systolic array—and thus, also relative to each other. Fortunately, there is a simple graphical method for doing the task.

We first choose an arbitrary iteration vector, say $v = (1, 1, 1)$. The latter we map with the projection matrix P to the cell where the calculation takes place,

$$z = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \; .$$

The iteration vector (1,1,1) represents the calculations $a(1, 1, 1)$, $b(1, 1, 1)$, and $c(1, 1, 1)$; these in turn correspond to the data items a_{11}, b_{11}, and c_{11}. We now lay the input/output schemes for the matrices A, B, C on the systolic array in a way that the entries a_{11}, b_{11}, and c_{11} all are located in cell $z = (0, 0)$.

In principle, we would be done now. Unfortunately, our input/output schemes overlap with the cells of the systolic array, and are therefore not easily perceivable. Thus, we simultaneously retract the input/output schemes of all matrices in counter flow direction, place by place, until there is no more overlapping. With this method, we get exactly the input/output scheme from Figure 16.6.

As an alternative to this nice graphical method, we also could formally calculate an overlap-free placement of the various input/output schemes.

Only after specifying the input/output schemes, we can correctly calculate the number of timesteps effectively needed. The first relevant timestep starts with the first input operation. The last relevant timestep ends with the last output of a result. For the example, we determine from Figure 16.6 the beginning of the calculation with the input of the data item b_{11} in timestep 0, and the end of the calculation after output of the result c_{35} in timestep 14. Altogether, we identify 15 timesteps—five more than with pure treatment of the real calculations.

16.3.4. Data rates induced by space-time transformations

The input schemes of the matrices A and B from Figure 16.1(a) have a dense layout: if we drew the borders of the matrices shown in the figure, there would be no spare places comprised.

Not so in Figure 16.6. In any input data stream, each data item is followed by two spare places there. For the input matrices this means: the *boundary cells* of the systolic array receive a proper data item only every third timestep.

This property is a direct result of the employed space-time transformation. In both examples, the abstract data structures themselves are dense. But how close the various items really come in the input/output scheme depends on the absolute value of the determinant of the transformation matrix T: in every input/output data stream, the proper items follow each other with a spacing of exactly $|\det(T)|$ places. Indeed $|\det(T)| = 1$ for Figure 16.1; as for Figure 16.6, we now can rate the fluffy spacing as a practical consequence of $|\det(T)| = 3$.

What to do with spare places as those in Figure 16.6? Although each cell of the systolic array from Figure 16.3 in fact does useful work only every third timestep, it would be nonsense to pause during two out of three timesteps. Strictly speaking, we can argue that values on places marked with dots in Figure 16.6 have no influence on the calculation of the shown items c_{ij}, because they never reach an active cell at time of the calculation of a variable $c(i, j, k)$. Thus, we may simply fill spare places with any value, no danger of disturbing the result. It is even feasible to execute three different matrix products at the same time on the systolic array from Figure 16.3, without interference. This will be our topic in Subsection 16.3.7.

16.3.5. Input/output expansion and extended input/output scheme

When further studying Figure 16.6, we can identify another problem. Check, for example, the itinerary of c_{22} through the cells of the systolic array. According to the space-time transformation, the calculations contributing to the value of c_{22} happen

in the cells $(-1,0)$, $(0,0)$, $(1,0)$, and $(2,0)$. But the input/output scheme from Figure 16.6 tells us that c_{22} also passes through cell $(-2,0)$ before, and eventually visits cell $(3,0)$, too.

This may be interpreted as some ***spurious calculations*** being introduced into the system (16.3) by the used space-time transformation, here, for example, at the new domain points (2,2,0) and (2,2,5). The reason for this phenomenon is that the *domains of the input/output operations* are not in parallel to the chosen *projection direction*. Thus, some input/output operations are projected onto cells that do not belong to the ***boundary*** of the systolic array. But in the interior of the systolic array, no input/output operation can be performed directly. The problem can be solved by extending the trajectory, in flow or counter flow direction, from these inner cells up to the boundary of the systolic array. But thereby we introduce some new calculations, and possibly also some new domain points. This technique is called ***input/output expansion***.

We must avoid that the additional calculations taking place in the cells (-2,0) and (3,0) corrupt the correct value of c_{22}. For the matrix product, this is quite easy—though the general case is more difficult. The generic sum operator has a neutral element, namely zero. Thus, if we can guarantee that by new calculations only zero is added, there will be no harm. All we have to do is providing always at least one zero operand to any spurious multiplication; this can be achieved by filling appropriate input slots with zero items.

Figure 16.7 shows an example of a properly extended input/output scheme. Preceding and following the items of matrix A, the necessary zero items have been filled in. Since the entered zeroes count like data items, the input/output scheme from Figure 16.6 has been retracted again by one place. The calculation now begins already in timestep -1, but ends as before with timestep 14. Thus we need 16 timesteps altogether.

16.3.6. Coping with stationary variables

Let us come back to the example from Figure 16.1(a). For inputting the items of matrices A and B, no expansion is required, since these items are always used in *boundary cells* first. But not so with matrix C! The items of C are calculated in stationary variables, hence always in the same cell. Thus most results c_{ij} are produced in inner cells of the systolic array, from where they have to be moved—in a separate action—to *boundary cells* of the systolic array.

Although this new challenge, on the face of it, appears very similar to the problem from Subsection 16.3.5, and thus very easy to solve, in fact we here have a completely different situation. It is not sufficient to extend existing data flows forward or backward up to the boundary of the systolic array. Since for stationary variables the dependence vector is projected to the null vector, which constitutes no extensible direction, there can be no spatial flow induced by this dependency. Possibly, we can construct some auxiliary extraction paths, but usually there are many degrees of freedom. Moreover, we then need a *control mechanism* inside the cells. For all these reasons, the problem is further dwelled on in Section 16.4.

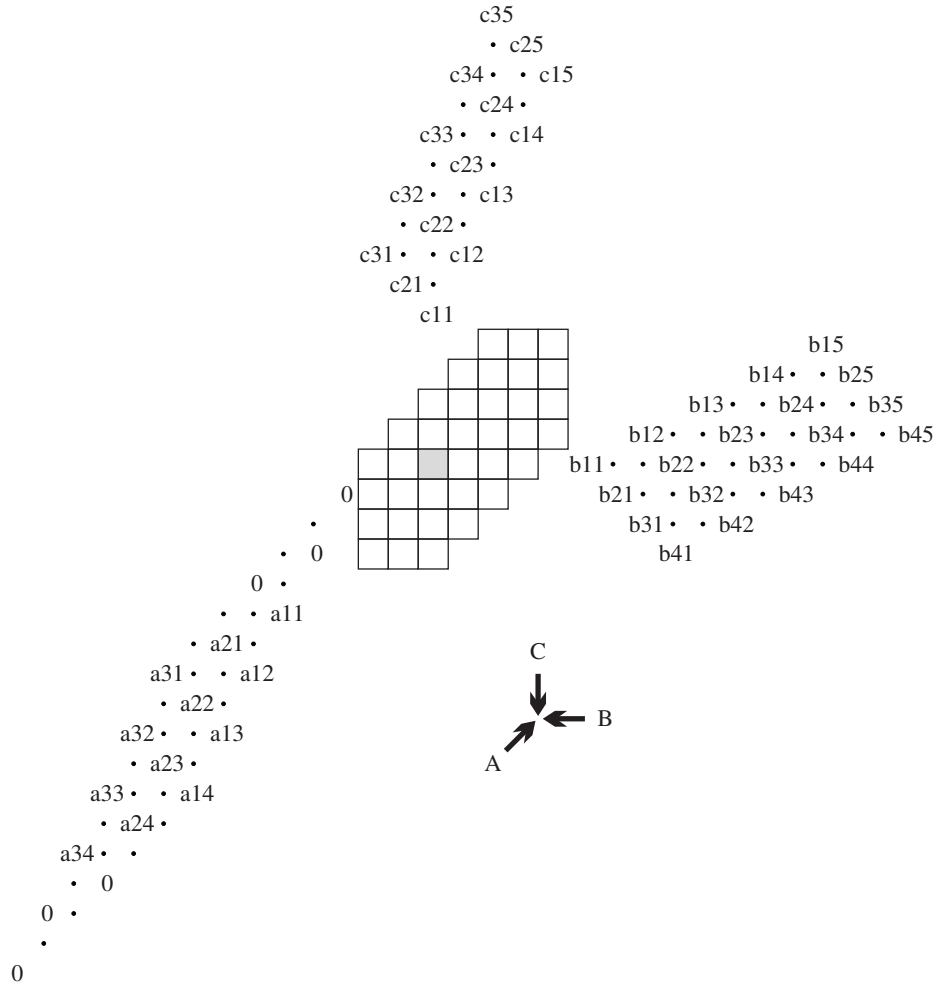


Figure 16.7. Extended input/output scheme, correcting Figure 16.6.

16.3.7. Interleaving of calculations

As can be easily noticed, the *utilisation* of the systolic array from Figure 16.3 with input/output scheme from Figure 16.7 is quite poor. Even without any deeper study of the starting phase and the closing phase, we cannot ignore that the average utilisation of the array is below one third—after all, each cell at most in every third timestep makes a proper contribution to the calculation.

A simple technique to improve this behaviour is to ***interleave*** calculations. If we have three independent matrix products, we can successively input their respective data, delayed by only one timestep, without any changes to the systolic array or its cells. Figure 16.8 shows a snapshot of the systolic array, with parts of the corresponding input/output scheme.

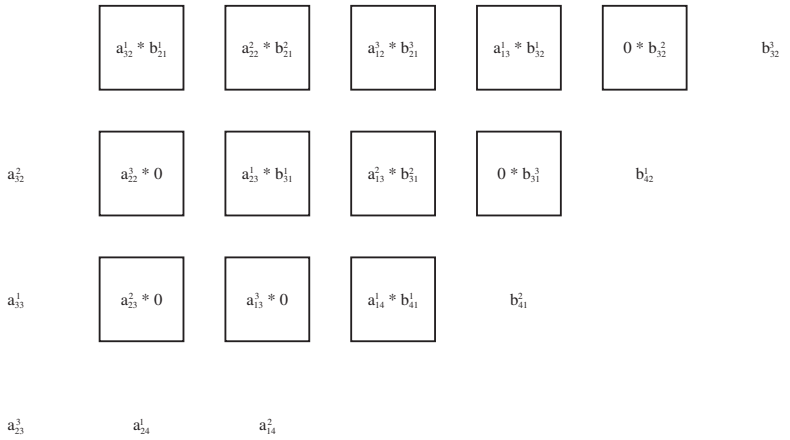


Figure 16.8. Interleaved calculation of three matrix products on the systolic array from Figure 16.3.

Now we must check by a formal derivation whether this idea is really working. Therefore, we slightly modify system (16.3). We augment the variables and the domains by a fourth dimension, needed to distinguish the three matrix products:

$$
\begin{aligned}
&\textit{input operations}\\
&a(i,j,k,l) = a^l_{ik} && 1 \le i \le N_1, j = 0, 1 \le k \le N_3, 1 \le l \le 3\ ,\\
&b(i,j,k,l) = b^l_{kj} && i = 0, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3\ ,\\
&c(i,j,k,l) = 0 && 1 \le i \le N_1, 1 \le j \le N_2, k = 0, 1 \le l \le 3\ .\\
&\textit{calculations and forwarding}\\
&a(i,j,k,l) = a(i,j-1,k,l) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3\ ,\\
&b(i,j,k,l) = b(i-1,j,k,l) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3\ ,\\
&c(i,j,k,l) = c(i,j,k-1,l)\\
&\quad + a(i,j-1,k,l) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3, 1 \le l \le 3\ .\\
&\quad \cdot b(i-1,j,k,l)\\
&\textit{output operations}\\
&c^l_{ij} = c(i,j,k,l) && 1 \le i \le N_1, 1 \le j \le N_2, k = N_3, 1 \le l \le 3\ .
\end{aligned}
\tag{16.32}
$$

Obviously, in system (16.32), problems with different values of l are not related. Now we must preserve this property in the systolic array. A suitable *space-time matrix* would be

$$
T = \begin{pmatrix} 0 & -1 & 1 & 0 \\ -1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}\ . \tag{16.33}
$$

Notice that T is not square here. But for calculating the space coordinates, the

fourth dimension of the iteration vector is completely irrelevant, and thus can simply be neutralised by corresponding zero entries in the fourth column of the first and second rows of T.

The last row of T again constitutes the *time vector* π. Appropriate choice of π embeds the three problems to solve into the space-time continuum, avoiding any intersection. Corresponding instances of the iteration vectors of the three problems are projected to the same cell with a respective spacing of one timestep, because the fourth entry of π equals 1.

Finally, we calculate the average *utilisation*—with or without interleaving—for the concrete problem parameters $N_1 = 3$, $N_2 = 5$, and $N_3 = 4$. For a single matrix product, we have to perform $N_1 \cdot N_2 \cdot N_3 = 60$ calculations, considering a multiplication and a corresponding addition as a compound operation, i.e., counting both together as only one calculation; input/output operations are not counted at all. The systolic array has 36 cells.

Without interleaving, our systolic array altogether takes 16 timesteps for calculating a single matrix product, resulting in an average utilisation of $60/(16 \cdot 36) \approx 0.104$ calculations per timestep and cell. When applying the described interleaving technique, the calculation of all three matrix products needs only two timesteps more, i.e., 18 timesteps altogether. But the number of calculations performed thereby has tripled, so we get an average utilisation of the cells amounting to $3 \cdot 60/(18 \cdot 36) \approx 0.278$ calculations per timestep and cell. Thus, by interleaving, we were able to improve the utilisation of the cells to 267 per cent!

Exercises

16.3-1 From equation (16.31), formally derive the spatial difference vectors of matrices B and C for the input/output scheme shown in Figure 16.6.

16.3-2 Augmenting Figure 16.6, draw an extended input/output scheme that forces both operands of all spurious multiplications to zero.

16.3-3 Apply the techniques presented in Section 16.3 to the systolic array from Figure 16.1.

16.3-4* Proof the properties claimed in Subsection 16.3.7 for the special space-time transformation (16.33) with respect to system (16.32).

16.4. Control

So far we have assumed that each cell of a systolic array behaves in completely the same way during every timestep. Admittedly there are some relevant examples of such systolic arrays. However, in general the cells successively have to work in several ***operation modes***, switched to by some control mechanism. In the sequel, we study some typical situations for exerting control.

16.4.1. Cells without control

The cell from Figure 16.3(b) contains the registers A, B, and C, that—when activated by the global clock signal—accept the data applied to their inputs and then reliably

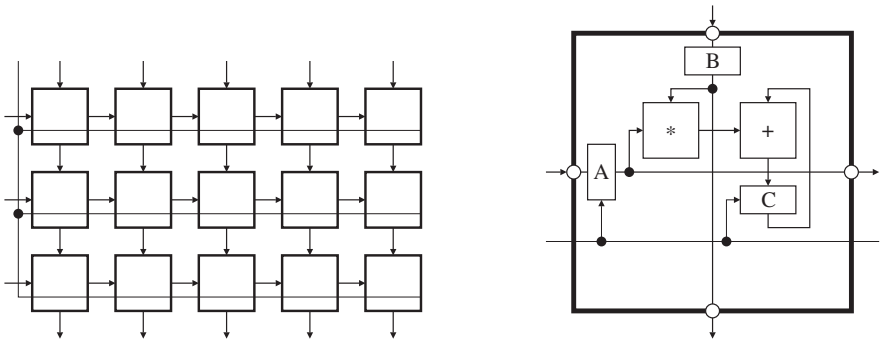


Figure 16.9. Resetting registers via global control. **(a)** Array structure. **(b)** Cell structure.

reproduce these values at their outputs for one clock cycle. Apart from this system-wide activity, the function calculated by the cell is invariant for all timesteps: a *fused multiply-add* operation is applied to the three input operands A, B, and C, with result passed to a neighbouring cell; during the same cycle, the operands A and B are also forwarded to two other neighbouring cells. So in this case, the cell needs *no control* at all.

The *initial values* $c(i, j, 0)$ for the execution of the generic sum operator—which could also be different from zero here—are provided to the systolic array via the *input streams*, see Figure 16.7; the *final results* $c(i, j, N_3)$ continue to flow into the same direction up to the boundary of the array. Therefore, the input/output activities for the cell from Figure 16.3(b) constitute an intrinsic part of the normal cell function. The price to pay for this extremely simple cell function without any control is a restriction in all three dimensions of the matrices: on a systolic array like that from Figure 16.3, with *fixed* array parameters N_1, N_2, N_3, an $M_1 \times M_3$ matrix A can only be multiplied by an $M_3 \times M_2$ matrix B if the relations $M_1 \leq N_1$, $M_2 \leq N_2$, and $M_3 \leq N_3$ hold.

16.4.2. Global control

In this respect, constraints for the array from Figure 16.1 are not so restrictive: though the problem parameters M_1 and M_2 also are bounded by $M_1 \leq N_1$ and $M_2 \leq N_2$, there is no constraint for M_3. Problem parameters unconstrained in spite of fixed array parameters can only emerge in time but not in space, thus mandating the use of *stationary variables*.

Before a new calculation can start, each register assigned to a stationary variable has to be *reset* to an initial state independent from the previously performed calculations. For instance, concerning the systolic cell from Figure 16.3(b), this should be the case for register C. By a *global* signal similar to the clock, register C can be cleared in all cells at the same time, i.e., reset to a zero value. To prevent a corruption of the reset by the current values of A or B, at least one of the registers A or B

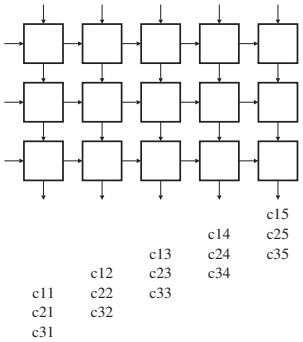


Figure 16.10. Output scheme with delayed output of results.

must be *cleared* at the same time, too. Figure 16.9 shows an array structure and a cell structure implementing this idea.

16.4.3. Local control

Unfortunately, for the matrix product the principle of the global control is not sufficient without further measures. Since the systolic array presented in Figure 16.1 even lacks another essential property: the results c_{ij} are not passed to the boundary but stay in the cells.

At first sight, it seems quite simple to forward the results to the boundary: when the calculation of an item c_{ij} is finished, the links from cell (i, j) to the neighbouring cells $(i, j+1)$ and $(i+1, j)$ are no longer needed to forward items of the matrices A and B. These links can be reused then for any other purpose. For example, we could pass all items of C through the downward-directed links to the lower border of the systolic array.

But it turns out that leading through results from the upper cells is hampered by ongoing calculations in the lower parts of the array. If the result c_{ij}, finished in timestep $i+j+N_3$, would be passed to cell $(i+1, j)$ in the next timestep, a conflict would be introduced between two values: since only one value per timestep can be sent from cell $(i+1, j)$ via the lower port, we would be forced to keep either c_{ij} or $c_{i+1\ j}$, the result currently finished in cell $(i+1, j)$. This effect would spread over all cells down.

To fix the problem, we could ***slow down*** the forwarding of items c_{ij}. If it would take two timesteps for c_{ij} to pass a cell, no collisions could occur. Then, the results stage a procession through the same link, each separated from the next by one timestep. From the lower boundary cell of a column, the host computer first receives the result of the bottom row, then that of the penultimate row; this procedure continues until eventually we see the result of the top row. Thus we get the output scheme shown in Figure 16.10.

How can a cell recognise when to change from forwarding items of matrix B to passing items of matrix C through the lower port? We can solve this task by an automaton combining global control with local control in the cell:

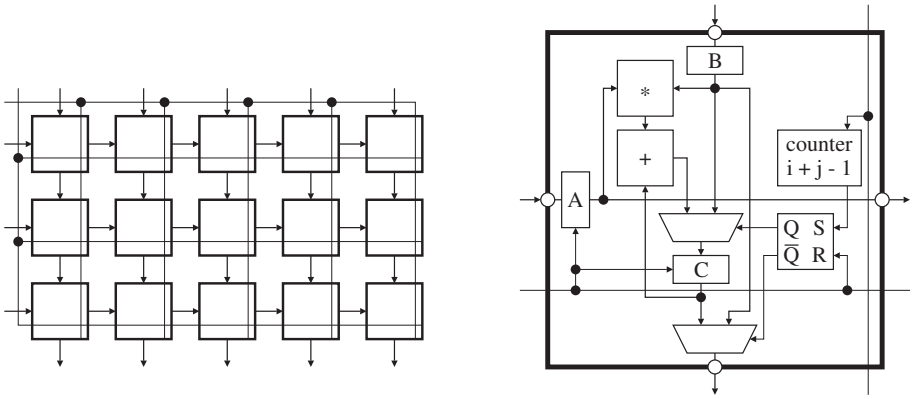


Figure 16.11. Combined local/global control. **(a)** Array structure. **(b)** Cell structure.

If we send a global signal to all cells at exactly the moment when the last items of A and B are input to cell $(1,1)$, each cell can start a countdown process: in each successive timestep, we decrement a counter initially set to the number of the remaining calculation steps. Thereby cell (i,j) still has to perform $i+j-1$ calculations before changing to ***propagation mode***. Later, the already mentioned global reset signal switches the cell back to ***calculation mode***.

Figure 16.11 presents a systolic array implementing this local/global principle. Basically, the array structure and the communication topology have been preserved. But each cell can run in one of two states now, switched by a *control logic*:

1. In *calculation mode*, as before, the result of the addition is written to register C. At the same time, the value in register B—i.e., the operand used for the multiplication—is forwarded through the lower port of the cell.
2. In *propagation mode*, registers B and C are connected in series. In this mode, the only function of the cell is to guide each value received at the upper port down to the lower port, thereby enforcing a *delay* of two timesteps.

The first value output from cell (i,j) in *propagation mode* is the currently calculated value c_{ij}, stored in register C. All further output values are results forwarded from cells above. A formal description of the algorithm implemented in Figure 16.11 is given by the assignment-free system (16.34).

input operations

$$\begin{aligned}
&a(i,j,k) = a_{ik} && 1 \le i \le N_1, j = 0, 1 \le k \le N_3 \ , \\
&b(i,j,k) = b_{kj} && i = 0, 1 \le j \le N_2, 1 \le k \le N_3 \ , \\
&c(i,j,k) = 0 && 1 \le i \le N_1, 1 \le j \le N_2, k = 0 \ .
\end{aligned}$$

calculations and forwarding

$$\begin{aligned}
&a(i,j,k) = a(i,j-1,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \ , \\
&b(i,j,k) = b(i-1,j,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \ , \\
&c(i,j,k) = c(i,j,k-1) \\
&\quad + a(i,j-1,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3 \ . \\
&\quad \cdot b(i-1,j,k)
\end{aligned} \tag{16.34}$$

propagation

$$\begin{aligned}
&b(i,j,k) = c(i,j,k-1) && 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i + N_3 \ , \\
&c(i,j,k) = b(i-1,j,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i - 1 + N_3 \ ,
\end{aligned}$$

output operations

$$c_{1+N_1+N_3-k,j} = b(i,j,k) \quad i = N_1, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3 \ .$$

It rests to explain how the control signals in a cell are generated in this model. As a prerequisite, the cell must contain a ***state flip-flop*** indicating the current *operation mode*. The output of this flip-flop is connected to the control inputs of both multiplexors, see Figure 16.11(b). The global reset signal clears the state flip-flop, as well as the registers A and C: the cell now works in *calculation mode*.

The global ready signal starts the countdown in all cells, so in every timestep the counter is diminished by 1. The counter is initially set to the precalculated value $i + j - 1$, dependent on the position of the cell. When the counter reaches zero, the flip-flop is set: the cell switches to *propagation mode*.

If desisting from a direct reset of the register C, the last value passed, before the reset, from register B to register C of a cell can be used as a freely decidable initial value for the next dot product to evaluate in the cell. We then even calculate, as already in the systolic array from Figure 16.3, the more general problem

$$C = A \cdot B + D \ , \tag{16.35}$$

detailed by the following equation system:

$$
\begin{aligned}
&\textit{input operations}\\
&a(i,j,k) = a_{ik} && 1 \le i \le N_1, j = 0, 1 \le k \le N_3\ ,\\
&b(i,j,k) = b_{kj} && i = 0, 1 \le j \le N_2, 1 \le k \le N_3\ ,\\
&c(i,j,k) = d_{ij} && 1 \le i \le N_1, 1 \le j \le N_2, k = 0\ .\\
&\textit{calculations and forwarding}\\
&a(i,j,k) = a(i,j-1,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3\ ,\\
&b(i,j,k) = b(i-1,j,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3\ ,\\
&c(i,j,k) = c(i,j,k-1)\\
&\quad + a(i,j-1,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3\ .\\
&\quad \cdot b(i-1,j,k)\\
&\textit{propagation}\\
&b(i,j,k) = c(i,j,k-1) && 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i + N_3\ ,\\
&c(i,j,k) = b(i-1,j,k) && 1 \le i \le N_1, 1 \le j \le N_2, 1 + N_3 \le k \le i - 1 + N_3\ .\\
&\textit{output operations}\\
&c_{1+N_1+N_3-k,j} = b(i,j,k) && i = N_1, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3\ .
\end{aligned}
\tag{16.36}
$$

16.4.4. Distributed control

The method sketched in Figure 16.11 still has the following drawbacks:

1. The systolic array uses global control signals, requiring a high technical accuracy.
2. Each cell needs a counter with counting register, introducing a considerable hardware expense.
3. The initial value of the counter varies between the cells. Thus, each cell must be individually designed and implemented.
4. The input data of any successive problem must wait outside the cells until all results from the current problem have left the systolic array.

These disadvantages can be avoided, if *control signals* are *propagated like data*—meaning a ***distributed control***. Therefore, we preserve the connections of the registers B and C with the multiplexors from Figure 16.11(b), but do not generate any control signals in the cells; also, there will be no global reset signal. Instead, a cell receives the necessary control signal from one of the neighbours, stores it in a new one-bit register S, and appropriately forwards it to further neighbouring cells. The primary control signals are generated by the *host computer*, and infused into the systolic array by *boundary cells*, only. Figure 16.12(a) shows the required array structure, Figure 16.12(b) the modified cell structure.

Switching to the *propagation mode* occurs successively down one cell in a column, always delayed by one timestep. The delay introduced by register S is therefore sufficient.

Reset to the *calculation mode* is performed via the same control wire, and thus also happens with a delay of one timestep per cell. But since the results c_{ij} sink down at half speed, only, we have to wait sufficiently long with the reset: if a cell is switched to *calculation mode* in timestep t, it goes to *propagation mode* in timestep $t + N_3$, and is reset back to *calculation mode* in timestep $t + N_1 + N_3$.

So we learned that in a systolic array, distributed control induces a different macroscopic timing behaviour than local/global control. Whereas the systolic array from Figure 16.12 can start the calculation of a new problem (16.35) every $N_1 + N_3$ timesteps, the systolic array from Figure 16.11 must wait for $2 \cdot N_1 + N_2 + N_3 - 2$ timesteps. The time difference $N_1 + N_3$ resp. $2 \cdot N_1 + N_2 + N_3 - 2$ is called the ***period***, its reciprocal being the ***throughput***.

System (16.37 and 16.38), divided into two parts during the typesetting, formally describes the relations between distributed control and calculations. We thereby assume an infinite, densely packed sequence of matrix product problems, the additional iteration variable l being unbounded. The equation headed *variables with alias* describes but pure identity relations.

$$
\begin{aligned}
&\textit{control}\\
&s(i,j,k,l) = 0 && i = 0, 1 \le j \le N_2, 1 \le k \le N_3\ ,\\
&s(i,j,k,l) = 1 && i = 0, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3\ ,\\
&s(i,j,k,l) = s(i-1,j,k,l) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ .\\
&\textit{input operations}\\
&a(i,j,k,l) = a^l_{ik} && 1 \le i \le N_1, j = 0, 1 \le k \le N_3\ ,\\
&b(i,j,k,l) = b^l_{kj} && i = 0, 1 \le j \le N_2, 1 \le k \le N_3\ ,\\
&b(i,j,k,l) = d^{l+1}_{N_1+N_3+1-k,j} && i = 0, 1 \le j \le N_2, 1 + N_3 \le k \le N_1 + N_3\ .\\
&\textit{variables with alias}\\
&c(i,j,k,l) = c(i,j,N_1+N_3,l-1) 1 \le i \le N_1, 1 \le j \le N_2, k = 0\ .
\end{aligned}
\tag{16.37}
$$

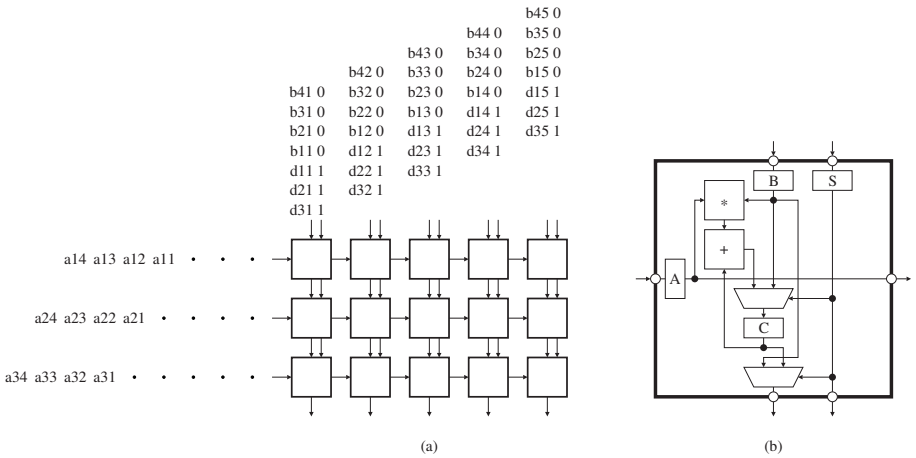


Figure 16.12. Matrix product on a rectangular systolic array, with output of results and distributed control. **(a)** Array structure. **(b)** Cell structure.

calculations and forwarding

$$a(i,j,k,l) = a(i,j-1,k,l) \qquad 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1+N_3 \ ,$$

$$b(i,j,k,l) = \begin{cases} b(i-1,j,k,l), \\ \text{if } s(i-1,j,k,l) = 0 \\ c(i,j,k-1,l), \\ \text{if } s(i-1,j,k,l) = 1 \end{cases} \qquad 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1+N_3 \ ,$$

$$c(i,j,k,l) = \begin{cases} c(i,j,k-1,l) \\ \quad + a(i,j-1,k,l) \\ \quad \cdot b(i-1,j,k,l), \\ \text{if } s(i-1,j,k,l) = 0 \\ b(i-1,j,k,l), \\ \text{if } s(i-1,j,k,l) = 1 \end{cases} \qquad 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1+N_3 \ . \tag{16.38}$$

output operations

$$c^l_{1+N_1+N_3-k,j} = b(i,j,k,l) \qquad i = N_1, 1 \le j \le N_2, 1+N_3 \le k \le N_1+N_3 \ .$$

Formula (16.39) shows the corresponding space-time matrix. Note that one entry of T is not constant but depends on the *problem parameters*:

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & N_1+N_3 \ . \end{pmatrix} \tag{16.39}$$

Interestingly, also the cells in a row switch one timestep later when moving one position to the right. Sacrificing some regularity, we could use this circumstance to

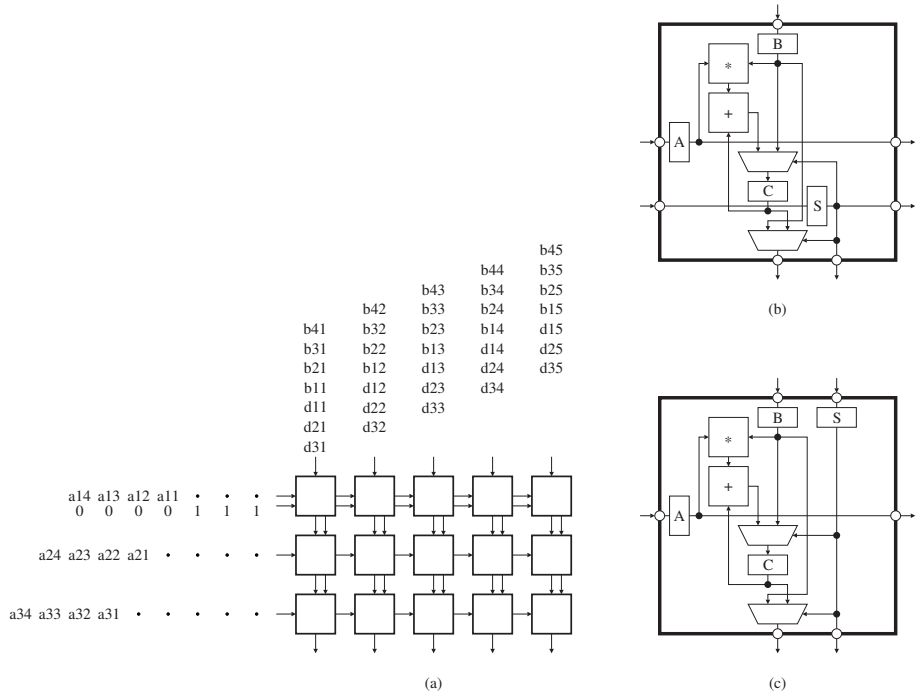


Figure 16.13. Matrix product on a rectangular systolic array, with output of results and distributed control. **(a)** Array structure. **(b)** Cell on the upper border. **(c)** Regular cell.

relieve the *host computer* by applying control to the systolic array at cell (1,1), only. We therefore would have to change the control scheme in the following way:

$$
\begin{aligned}
&\textit{control}\\
&s(i,j,k,l) = 0 && i = 1, j = 0, 1 \le k \le N_3\ ,\\
&s(i,j,k,l) = 1 && i = 1, j = 0, 1 + N_3 \le k \le N_1 + N_3\ ,\\
&s(i,j,k,l) = s(i-1,j,k,l) && 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ ,\\
&\ldots\\
&\textit{variables with alias}\\
&s(i,j,k,l) = s(i+1,j-1,k,l) && i = 0, 1 \le j \le N_2, 1 \le k \le N_1 + N_3\ ,\\
&\ldots
\end{aligned}
\tag{16.40}
$$

Figure 16.13 shows the result of this modification. We now need cells of two kinds: cells on the upper border of the systolic array must be like that in Figure 16.13(b); all other cells would be as before, see Figure 16.13(c). Moreover, the *communication topology* on the upper border of the systolic array would be slightly different from that in the regular area.

16.4.5. The cell program as a local view

The chosen *space-time transformation* widely determines the architecture of the systolic array. Mapping recurrence equations to space-time coordinates yields an explicit view to the *geometric properties* of the systolic array, but gives no real insight into the *function of the cells*. In contrast, the processes performed inside a cell can be directly expressed by a ***cell program***. This approach is particularly of interest if dealing with a *programmable systolic array*, consisting of cells indeed controlled by a repetitive program.

Like the ***global view***, i.e., the *structure* of the systolic array, the ***local view*** given by a *cell program* in fact is already fixed by the space-time transformation. But, this local view is only induced implicitly here, and thus, by a further mathematical transformation, an explicit representation must be extracted, suitable as a cell program.

In general, we denote instances of program variables with the aid of ***index expressions***, that refer to iteration variables. Take, for instance, the equation

$$c(i,j,k) = c(i,j,k-1) + a(i,j-1,k) \cdot b(i-1,j,k) \quad 1 \le i \le N_1, 1 \le j \le N_2, 1 \le k \le N_3$$

from system (16.3). The instance $c(i,j,k-1)$ of the program variable c is specified using the index expressions i, j, and $k-1$, which can be regarded as functions of the iteration variables i,j,k.

As we have noticed, the set of iteration vectors (i,j,k) from the quantification becomes a set of space-time coordinates (x,y,t) when applying a space-time transformation (16.12) with transformation matrix T from (16.14),

$$\begin{pmatrix} x \\ y \\ t \end{pmatrix} = T \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} . \tag{16.41}$$

Since each cell is denoted by space coordinates (x,y), and the cell program must refer to the current time t, the iteration variables i,j,k in the index expressions for the program variables are not suitable, and must be translated into the new coordinates x,y,t. Therefore, using the inverse of the space-time transformation from (16.41), we express the iteration variables i,j,k as functions of the space-time coordinates (x,y,t),

$$\begin{pmatrix} i \\ j \\ k \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} x \\ y \\ t \end{pmatrix} = \frac{1}{3} \cdot \begin{pmatrix} -1 & -2 & 1 \\ -1 & 1 & 1 \\ 2 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ t \end{pmatrix} . \tag{16.42}$$

The existence of such an inverse transformation is guaranteed if the space-time transformation is injective on the domain—and that it should always be: if not, some instances must be calculated by a cell in the same timestep. In the example, reversibility is guaranteed by the square, non singular matrix T, even without referral to the domain. With respect to the time vector π and any projection vector u, the property $\pi \cdot u \neq 0$ is sufficient.

Replacing iteration variables by space-time coordinates, which might be interpreted as a ***transformation of the domain***, frequently yields very unpleasant

index expressions. Here, for example, from $c(i, j, k-1)$ we get

$$c((-x-2\cdot y+t)/3, (-x+y+t)/3, (2\cdot x+y+t)/3) \ .$$

But, by a successive ***transformation of the index sets***, we can relabel the instances of the program variables such that the reference to cell and time appears more evident. In particular, it seems worthwhile to transform the equation system back into ***output normal form***, i.e., to denote the results calculated during timestep t in cell (x, y) by instances (x, y, t) of the program variables. We best gain a real understanding of this approach via an abstract mathematical formalism, that we can fit to our special situation.

Therefore, let

$$r(\psi_r(v)) = \mathcal{F}(\ldots, s(\psi_s(v)), \ldots) \qquad v \in S \tag{16.43}$$

be a quantified equation over a domain S, with program variables r and s. The ***index functions*** ψ_r and ψ_s generate the instances of the program variables as tuples of index expressions.

By transforming the domain with a function φ that is injective on S, equation (16.43) becomes

$$r(\psi_r(\varphi^{-1}(e))) = \mathcal{F}(\ldots, s(\psi_s(\varphi^{-1}(e))), \ldots) \qquad e \in \varphi(S) \ , \tag{16.44}$$

where φ^{-1} is a function that constitutes an inverse of φ on $\varphi(S)$. The new index functions are $\psi_r \circ \varphi^{-1}$ and $\psi_s \circ \varphi^{-1}$. Transformations of index sets don't touch the domain; they can be applied to each program variable separately, since only the instances of this program variable are renamed, and in a consistent way. With such renamings ϑ_r and ϑ_s, equation (16.44) becomes

$$r(\vartheta_r(\psi_r(\varphi^{-1}(e)))) = \mathcal{F}(\ldots, s(\vartheta_s(\psi_s(\varphi^{-1}(e)))), \ldots) \qquad e \in \varphi(S) \ . \tag{16.45}$$

If output normal form is desired, $\vartheta_r \circ \psi_r \circ \varphi^{-1}$ has to be the identity.

In the most simple case (as for our example), ψ_r is the identity, and ψ_s is an affine transformation of the form $\psi_s(v) = v - d$, with constant d—the already known *dependence vector*. ψ_r then can be represented in the same way, with $d = \vec{0}$. Transformation of the domains happens by the space-time transformation $\varphi(v) = T \cdot v$, with an invertible matrix T. For all index transformations, we choose the same $\vartheta = \varphi$. Thus equation (16.45) becomes

$$r(e) = \mathcal{F}(\ldots, s(e - T \cdot d), \ldots) \qquad e \in T(S) \ . \tag{16.46}$$

For the generation of a *cell program*, we have to know the following information for every timestep: the operation to perform, the source of the data, and the destination of the results—known from assembler programs as `opc`, `src`, `dst`.

The operation to perform (`opc`) follows directly from the function $\mathcal{F}$. For a cell with control, we must also find the timesteps when to perform this individual function $\mathcal{F}$. The set of these timesteps, as a function of the space coordinates, can

be determined by projecting the set $T(S)$ onto the time axis; for general polyhedric S with the aid of a *Fourier-Motzkin elimination*, for example.

In system (16.46), we get a new dependence vector $T \cdot d$, consisting of two components: a (vectorial) spatial part, and a (scalar) timely part. The *spatial* part Δz, as a difference vector, specifies *which* neighbouring cell has calculated the operand. We directly can translate this information, concerning the input of operands to cell z, into a port specifier with port position $-\Delta z$, serving as the `src` operand of the instruction. In the same way, the cell calculating the operand, with position $z - \Delta z$, must write this value to a port with port position Δz, used as the `dst` operand in the instruction.

The *timely* part of $T \cdot d$ specifies, as a time difference Δt, *when* the calculation of the operand has been performed. If $\Delta t = 1$, this information is irrelevant, because the reading cell z always gets the output of the immediately preceding timestep from neighbouring cells. However, for $\Delta t > 1$, the value must be buffered for $\Delta t - 1$ timesteps, either by the *producer* cell $z - \Delta z$, or by the *consumer* cell z—or by both, sharing the burden. This need can be realised in the cell program, for example, with $\Delta t - 1$ copy instructions executed by the producer cell $z - \Delta z$, preserving the value of the operand until its final output from the cell by passing it through $\Delta t - 1$ registers.

Applying this method to system (16.37 and 16.38), with transformation matrix T as in (16.39), yields

$$
\begin{aligned}
s(x,y,t) &= s(x-1,y,t-1) \\
a(x,y,t) &= a(x,y-1,t-1) \\
b(x,y,t) &= \begin{cases} b(x-1,y,t-1), \\ \text{if } s(x-1,y,t-1) = 0 \\ c(x,y,t-1), \\ \text{if } s(x-1,y,t-1) = 1 \end{cases} \\
c(x,y,t) &= \begin{cases} c(x,y,t-1) + a(x,y-1,t-1) \cdot b(x-1,y,t-1)\ , \\ \text{if } s(x-1,y,t-1) = 0 \\ b(x-1,y,t-1), \\ \text{if } s(x-1,y,t-1) = 1\ . \end{cases}
\end{aligned}
\tag{16.47}
$$

The iteration variable l, being relevant only for the input/output scheme, can be set to a fixed value prior to the transformation. The cell program for the systolic array from Figure 16.12, performed once in every timestep, reads as follows:

CELL-PROGRAM

```
1  S ← C(−1, 0)(0)
2  A ← C(0, −1)
3  B ← C(−1, 0)(1 : N)
4  C(1, 0)(0) ← S
5  C(0, 1) ← A
6  if S = 1
7     then C(1, 0)(1 : N) ← C
8          C ← B
9     else C(1, 0)(1 : N) ← B
10         C ← C + A · B
```

The port specifiers stand for local input/output to/from the cell. For each, a pair of qualifiers is derived from the geometric position of the ports relative to the centre of the cell. Port $\mathtt{C}(0,-1)$ is situated on the left border of the cell, $\mathtt{C}(0,1)$ on the right border; $\mathtt{C}(-1,0)$ is above the centre, $\mathtt{C}(1,0)$ below. Each port specifier can be augmented by a bit range: $\mathtt{C}(-1,0)(0)$ stands for bit 0 of the port, only; $\mathtt{C}(-1,0)(1:N)$ denotes the bits 1 to N. The designations A, B, ... without port qualifiers stand for registers of the cell.

By application of matrix T from (16.13) to system (16.36), we get

$$\begin{aligned}
&a(x,y,t) = a(x,y-1,t-1) && 1+x+y \le t \le x+y+N_3\ ,\\
&b(x,y,t) = b(x-1,y,t-1) && 1+x+y \le t \le x+y+N_3\ ,\\
&c(x,y,t) = c(x,y,t-1)\\
&\quad + a(x,y-1,t-1) && 1+x+y \le t \le x+y+N_3\ ,\\
&\quad \cdot b(x-1,y,t-1)\\
&b(x,y,t) = c(x,y,t-1) && x+y+1+N_3 \le t \le 2\cdot x+y+N_3\ ,\\
&c(x,y,t) = b(x-1,y,t-1) && x+y+1+N_3 \le t \le 2\cdot x+y-1+N_3\ .
\end{aligned} \tag{16.48}$$

Now the advantages of distributed control become obvious. The cell program for (16.47) can be written with referral to the respective timestep t, only. And thus, we need no reaction to global control signals, no counting register, no counting operations, and no coding of the local cell coordinates.

Exercises

16.4-1 Specify appropriate input/output schemes for performing, on the systolic arrays presented in Figures 16.11 and 16.12, two evaluations of system (16.36) that follow each other closest in time.

16.4-2 How could we change the systolic array from Figure 16.12, to efficiently support the calculation of matrix products with parameters $M_1 < N_1$ or $M_2 < N_2$?

16.4-3 Write a cell program for the systolic array from Figure 16.3.

16.4-4⋆ Which throughput allows the systolic array from Figure 16.3 for the assumed values of N_1, N_2, N_3? Which for general N_1, N_2, N_3?

16.4-5⋆ Modify the systolic array from Figure 16.1 such that the results stored in stationary variables are output through additional links directed half right down,

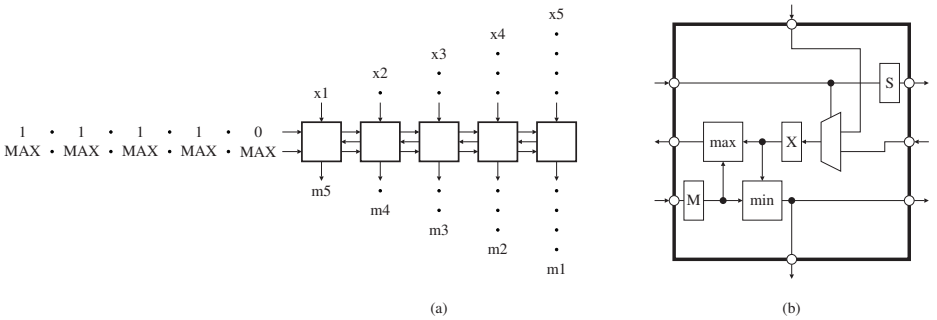


Figure 16.14. *Bubble sort* algorithm on a linear systolic array. **(a)** Array structure with input/output scheme. **(b)** Cell structure.

i.e., from cell (i, j) to cell $(i + 1, j + 1)$. Develop an assignment-free equation system functionally equivalent to system (16.36), that is compatible with the extended structure. How looks the resulting input/output scheme? Which period is obtained?

16.5. Linear systolic arrays

Explanations in the sections above heavily focused on *two-dimensional* systolic arrays, but in principle also apply to *one-dimensional* systolic arrays, called ***linear systolic arrays*** in the sequel. The most relevant difference between both kinds concerns the *boundary* of the systolic array. Linear systolic arrays can be regarded as consisting of *boundary cells*, only; under this assumption, input from and output to the *host computer* needs no special concern. However, the geometry of a linear systolic array provides one full dimension as well as one fictitious dimension, and thus communication along the full-dimensional axis may involve similar questions as in Subsection 16.3.5. Eventually, the boundary of the linear systolic array can also be defined in a radically different way, namely to consist of both ***end cells***, only.

16.5.1. Matrix-vector product

If we set one of the problem parameters N_1 or N_2 to value 1 for a systolic array as that from Figure 16.1, the matrix product means to multiply a matrix by a vector, from left or right. The two-dimensional systolic array then ***degenerates*** to a one-dimensional systolic array. The vector by which to multiply is provided as an input data stream through an end cell of the linear systolic array. The matrix items are input to the array simultaneously, using the complete broadside.

As for full matrix product, results emerge stationary. But now, they either can be drained along the array to one of the end cells, or they are sent directly from the producer cells to the *host computer*. Both methods result in different control mechanisms, time schemes, and running time.

Now, would it be possible to provide *all* inputs via end cells? The answer is negative if the running time should be of complexity $\Theta(N)$. Matrix A contains $\Theta(N^2)$ items, thus there are $\Theta(N)$ items per timestep to read. But the number of items receivable through an end cell during one timestep is bounded. Thus, the ***input/output data rate***—of order $\Theta(N)$, here—may already constrain the possible design space.

16.5.2. Sorting algorithms

For sorting, the task is to bring the elements from a set $\{x_1, \ldots, x_N\}$, subset of a totally ordered basic set G, into an ascending order $\{m_i\}_{i=1,\ldots,N}$ where $m_i \leq m_k$ for $i < k$. A solution to this problem is described by the following assignment-free equation system, where MAX denotes the maximum in G:

$$\begin{array}{ll}
\textit{input operations} & \\
x(i,j) = x_i & 1 \leq i \leq N, j = 0\ , \\
m(i,j) = MAX & 1 \leq j \leq N, i = j - 1\ . \\
 & \\
\textit{calculations} & \\
m(i,j) = \min\{x(i,j-1), m(i-1,j)\} & 1 \leq i \leq N, 1 \leq j \leq i\ , \\
x(i,j) = \max\{x(i,j-1), m(i-1,j)\} & 1 \leq i \leq N, 1 \leq j \leq i\ . \\
 & \\
\textit{output operations} & \\
m(i,j) = m_j & 1 \leq j \leq N, i = N\ .
\end{array} \tag{16.49}$$

By completing a projection along direction $u = (1, 1)$ to a space-time transformation

$$\begin{pmatrix} x \\ t \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix}\ , \tag{16.50}$$

we get the linear systolic array from Figure 16.14, as an implementation of the *bubble sort* algorithm.

Correspondingly, the space-time matrix

$$T = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \tag{16.51}$$

would induce another linear systolic array, that implements *insertion sort*. Eventually, the space-time matrix

$$T = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{16.52}$$

would lead to still another linear systolic array, this one for *selection sort*.

For the sorting problem, we have $\Theta(N)$ input items, $\Theta(N)$ output items, and $\Theta(N)$ timesteps. This results in an input/output data rate of order $\Theta(1)$. In contrast to the matrix-vector product from Subsection 16.5.1, the sorting problem with any prescribed input/output data rate in principle allows to perform the communication exclusively through the end cells of a linear systolic array.

Note that, in all three variants of sorting described so far, direct input is necessary to all cells: the values to order for bubble sort, the constant values MAX for insertion sort, and both for selection sort. However, instead of inputting the constants, the cells could generate them, or read them from a local memory.

All three variants require a cell control: insertion sort and selection sort use stationary variables; bubble sort has to switch between the processing of input data and the output of calculated values.

16.5.3. Lower triangular linear equation systems

System (16.53) below describes a *localised* algorithm for solving the linear equation system $A \cdot x = b$, where the $N \times N$ matrix A is a lower triangular matrix.

$$
\begin{array}{ll}
\textit{input operations} & \\
a(i,j) = a_{i,j+1} & 1 \le i \le N, 0 \le j \le i-1\ , \\
u(i,j) = b_i & 1 \le i \le N, j = 0\ . \\
 & \\
\textit{calculations and forwarding} & \\
u(i,j) = u(i,j-1) - a(i,j-1) \cdot x(i-1,j) & 2 \le i \le N, 1 \le j \le i-1\ , \\
x(i,j) = u(i,j-1)/a(i,j-1) & 1 \le i \le N, j = i\ , \\
x(i,j) = x(i-1,j) & 2 \le i \le N-1, 1 \le j \le i-1\ . \\
 & \\
\textit{output operations} & \\
x_i = x(i,j) & 1 \le i \le N, j = i\ .
\end{array}
\tag{16.53}
$$

All previous examples had in common that, apart from copy operations, the same kind of calculation had to be performed for each domain point: fused multiply/add for the matrix algorithms, minimum and maximum for the sorting algorithms. In contrast, system (16.53) contains some domain points where multiply and subtract is required, as well as some others needing division.

When projecting system (16.53) to a linear systolic array, depending on the chosen *projection direction* we get fixed or varying cell functions. Peculiar for projecting along $u = (1,1)$, we see a single cell with divider; all other cells need a multiply/subtract unit. Projection along $u = (1,0)$ or $u = (0,1)$ yields identical cells, all containing a divider as well as a multiply/subtract unit. Projection vector $u = (1,-1)$ results in a linear systolic array with three different cell types: both end cells need a divider, only; all other cells contain a multiply/subtract unit, with or without divider, alternatingly. Thus, a certain projection can introduce ***inhomogeneities*** into a systolic array—that may be desirable, or not.

Exercises

16.5-1 For both variants of matrix-vector product as in Subsection 16.5.1—output of the results by an end cell versus communication by all cells—specify a suitable array structure with input/output scheme and cell structure, including the necessary control mechanisms.

16.5-2 Study the effects of further projection directions on system (16.53).
16.5-3 Construct systolic arrays implementing insertion sort and selection sort, as mentioned in Subsection 16.5.2. Also draw the corresponding cell structures.
16.5-4⋆ The systolic array for bubble sort from Figure 16.14 could be operated without control by cleverly organising the input streams. Can you find the trick?
16.5-5⋆ What purpose serves the value MAX in system (16.49)? How system (16.49) could be formulated without this constant value? Which consequences this would incur for the systolic arrays described?

Problems

16-1 Band matrix algorithms

In Sections 16.1, 16.2, and Subsections 16.5.1, and 16.5.3, we always assumed ***full*** input matrices, i.e., each matrix item a_{ij} used could be nonzero in principle. (Though in a lower triangular matrix, items above the main diagonal are all zero. Note, however, that these items are not inputs to any of the algorithms described.)

In contrast, practical problems frequently involve ***band matrices***, cf. Kung/Leiserson [204]. In such a matrix, most diagonals are zero, left alone a small band around the main diagonal. Formally, we have $a_{ij} = 0$ for all i, j with $i - j \geq K$ or $j - i \geq L$, where K and L are positive integers. The ***band width***, i.e., the number of diagonals where nonzero items may appear, here amounts to $K + L - 1$.

Now the question arises whether we could profit from the band structure in one or more input matrices to optimise the systolic calculation. One opportunity would be to delete cells doing no useful work. Other benefits could be shorter input/output data streams, reduced running time, or higher throughput.

Study all systolic arrays presented in this chapter for improvements with respect to these criteria.

Chapter notes

The term *systolic array* has been coined by Kung and Leiserson in their seminal paper [204].

Karp, Miller, and Winograd did some pioneering work [188] for *uniform recurrence equations*.

Essential stimuli for a theory on the systematic design of systolic arrays have been Rao's PhD dissertation [282] and the work of Quinton [280].

The contribution of Teich and Thiele [319] shows that a formal derivation of the cell control can be achieved by methods very similar to those for a determination of the geometric array structure and the basic cell function.

The up-to-date book by Darte, Robert, and Vivien [79] joins advanced methods from compiler design and systolic array design, dealing also with the analysis of data dependences.

The monograph [357] still seems to be the most comprehensive work on systolic systems.

Each systolic array can also be modelled as a *cellular automaton*. The registers in a cell together hold the state of the cell. Thus, a *factorised* state space is adequate. Cells of different kind, for instance with varying cell functionality or position-dependent cell control, can be described with the aid of further components of the state space.

Each systolic algorithm also can be regarded as a *PRAM algorithm* with the same timing behaviour. Thereby, each register in a systolic cell corresponds to a PRAM memory cell, and vice versa. The EREW PRAM model is sufficient, because in every timestep exactly one systolic cell reads from this register, and then exactly one systolic cell writes to this register.

Each systolic system also is a special kind of *synchronous network* as defined by Lynch [227]. Time complexity measures agree. Communication complexity usually is no topic with systolic arrays. Restriction to input/output through boundary cells, frequently demanded for systolic arrays, also can be modelled in a synchronous network. The concept of *failures* is not required for systolic arrays.

The book written by Sima, Fountain and Kacsuk [303] considers the systolic systems in details.

V. DATA BASES

17. Memory Management

The main task of computers is to execute programs (even usually several programs running simultaneously). These programs and their data must be in the main memory of the computer during the execution.

Since the main memory is usually too small to store all these data and programs, modern computer systems have a secondary storage too for the provisional storage of the data and programs.

In this chapter the basic algorithms of memory management will be covered. In Section 17.1 static and dynamic partitioning, while in Section 17.2 the most popular paging methods will be discussed.

In Section 17.3 the most famous anomaly of the history of operating systems—the stunning features of FIFO page changing algorithm, interleaved memory and processing algorithms with lists—will be analysed.

Finally in Section 17.4 the discussion of the optimal and approximation algorithms for the optimisation problem in which there are files with given size to be stored on the least number of disks can be found.

17.1. Partitioning

A simple way of sharing the memory between programs is to divide the whole address space into slices, and assign such a slice to every process. These slices are called ***partitions***. The solution does not require any special hardware support, the only thing needed is that programs should be ready to be loaded to different memory addresses, i. e., they should be ***relocatable***. This must be required since it cannot be guaranteed that a program always gets into the same partition, because the total size of the executable programs is usually much more than the size of the whole memory. Furthermore, we cannot determine which programs can run simultaneously and which not, for processes are generally independent of each other, and in many cases their owners are different users. Therefore, it is also possible that the same program is executed by different users at the same time, and different instances work with different data, which can therefore not be stored in the same part of the memory. Relocation can be easily performed if the linker does not work with

absolute but with relative memory addresses, which means it does not use exact addresses in the memory but a base address and an offset. This method is called ***base addressing***, where the initial address is stored in the so called base register. Most processors know this addressing method, therefore, the program will not be slower than in the case using absolute addresses. By using base addressing it can also be avoided that – due to an error or the intentional behaviour of a user – the program reads or modifies the data of other programs stored at lower addresses of the memory. If the solution is extended by another register, the so called limit register which stores the biggest allowed offset, i. e. the size of the partition, then it can be assured that the program cannot access other programs stored at higher memory addresses either.

Partitioning was often used in mainframe computer operating systems before. Most of the modern operating systems, however, use virtual memory management which requires special hardware support.

Partitioning as a memory sharing method is not only applicable in operating systems. When writing a program in a language close to machine code, it can happen that different data structures with variable size—which are created and cancelled dynamically—have to be placed into a continuous memory space. These data structures are similar to processes, with the exception that security problems like addressing outside their own area do not have to be dealt with. Therefore, most of the algorithms listed below with some minor modifications can be useful for application development as well.

Basically, there are two ways of dividing the address space into partitions. One of them divides the initially empty memory area into slices, the number and size of which is predetermined at the beginning, and try to place the processes and other data structures continuously into them, or remove them from the partitions if they are not needed any more. These are called fixed partitions, since both their place and size have been fixed previously, when starting the operating system or the application. The other method is to allocate slices from the free parts of the memory to the newly created processes and data structures continuously, and to deallocate the slices again when those end. This solution is called dynamic partitioning, since partitions are created and destroyed dynamically. Both methods have got advantages as well as disadvantages, and their implementations require totally different algorithms. These will be discussed in the following.

17.1.1. Fixed partitions

Using ***fixed partitions*** the division of the address space is fixed at the beginning, and cannot be changed later while the system is up. In the case of operating systems the operator defines the partition table which is activated at next reboot. Before execution of the first application, the address space is already partitioned. In the case of applications partitioning has to be done before creation of the first data structure in the designated memory space. After that data structures of different sizes can be placed into these partitions.

In the following we examine only the case of operating systems, while we leave to the Reader the rewriting of the problem and the algorithms according to

given applications, since these can differ significantly depending on the kind of the applications.

The partitioning of the address space must be done after examination of the sizes and number of possible processes running on the system. Obviously, there is a maximum size, and programs exceeding it cannot be executed. The size of the largest partition corresponds to this maximum size. To reach the optimal partitioning, often statistic surveys have to be carried out, and the sizes of the partitions have to be modified according to these statistics before restarting the system next time. We do not discuss the implementation of this solution now.

Since there are a constant number (m) of partitions, their data can be stored in one or more arrays with constant lengths. We do not deal with the particular place of the partitions on this level of abstraction either; we suppose that they are stored in a constant array as well. When placing a process in a partition, we store the index of that partition in the process header instead of its starting address. However, concrete implementation can differ from this method, of course. The sizes of the partitions are stored in array $size[1\,..\,m]$. Our processes are numbered from 1 to n. Array $part[1\,..\,m]$ keeps track of the processes executed in the individual partitions, while its inverse, array $place[1\,..\,n]$ stores the places where individual processes are executed. A process is either running, or waiting for a partition. This information is stored in Boolean array $waiting[1\,..\,n]$: if process number i is waiting, then $waiting[i]$ = TRUE, else $waiting[i]$ = FALSE. The space requirements of the processes are different. Array $spacereq[1\,..\,n]$ stores the minimum sizes of partitions required to execute the individual processes.

Having partitions of different sizes and processes with different space requirements, we obviously would not like small processes to be placed into large partitions, while smaller partitions are empty, in which larger processes do not fit. Therefore, our goal is to assign each partition to a process fitting into it in a way that there is no larger process that would fit into it as well. This is ensured by the following algorithm:

LARGEST-FIT(*place*, *spacereq*, *size*, *part*, *waiting*)

```
1  for j ← 1 to m
2      do if part[j] = 0
3             then LOAD-LARGEST(place, spacereq, size, j, part, waiting)
```

Finding the largest process the whose space requirement is not larger than a particular size is a simple conditional maximum search. If we cannot find any processes meeting the requirements, we must leave the the partition empty.

```
LOAD-LARGEST(place, spacereq, size, p, part, waiting)
1  max ← 0
2  ind ← 0
3  for i ← 1 to n
4      do if waiting[i] and spacereq[i] ≤ size[p] and spacereq[i] > max
5            then ind ← i
6                 max ← spacereq[i]
7  if ind > 0
8    then part[p] ← ind
9         place[ind] ← p
10        waiting[ind] ← FALSE
```

The basic criteria of the correctness of all the algorithms loading the processes into the partitions is that they should not load a process into a partition which does not fit. This requirement is fulfilled by the above algorithm, since it can be derived from the conditional maximum search theorem exactly with the mentioned condition.

Another essential criterion is that it should not load more than one processes into the same partition, and also should not load one single process into more partitions simultaneously. The first case can be excluded, because we call the LOAD-LARGEST algorithm only for the partitions for which $part[j] = 0$ and if we load a process into partition number p, then we give $part[p]$ the index of the loaded process as a value, which is a positive integer. The second case can be proved similarly: the condition of the conditional maximum search excludes the processes for which $waiting[i] = \text{FALSE}$, and if the process number ind is loaded into one of the partitions, then the value of $waiting[ind]$ is set to FALSE.

However, the fact that the algorithm does not load a process into a partition where it does not fit, does not load more then one processes into the same partition, or one single process into more partitions simultaneously is insufficient. These requirements are fulfilled even by an empty algorithm. Therefore, we have to require something more: namely that it should not leave a partition empty, if there is a process that would fit into it. To ensure this, we need an invariant, which holds during the whole loop, and at the end of the loop it implies our new requirement. Let this invariant be the following: after examination of j partitions, there is no positive $k \leq j$, for which $part[k] = 0$, and for which there is a positive $i \leq n$, such as $waiting[i] = \text{TRUE}$, and $spacereq[i] \leq size[k]$.

Initialisation: At the beginning of the algorithm we have examined $j = 0$ partitions, so there is not any positive $k \leq j$.

Maintenance: If the invariant holds for j at the beginning of the loop, first we have to check whether it holds for the same j at the end of the loop as well. It is obvious, since the first j partitions are not modified when examining the $(j + 1)$-th one, and for the processes they contain $waiting[i] = \text{FALSE}$, which does not satisfy the condition of the conditional maximum search in the LOAD-LARGEST algorithm. The invariant holds for the $(j + 1)$-th partition at the end

of the loop as well, because if there is a process which fulfills the condition, the conditional maximum search certainly finds it, since the condition of our conditional maximum search corresponds to the requirement of our invariant set on each partition.

Termination: Since the loop traverses a fixed interval by one, it will certainly stop. Since the loop body is executed exactly as many times as the number of the partitions, after the end of the loop there is no positive $k \leq m$, for which $part[k] = 0$,, and for which there is a positive $i \leq n$, such that $waiting[i] = \text{TRUE}$ and $spacereq[i] \leq size[k]$, which means that we did not fail to fill any partitions that could be assigned to a process fitting into it.

The loop in rows 1–3 of the LARGEST-FIT algorithm is always executed in its entirety, so the loop body is executed $\Theta(m)$ times. The loop body performs a conditional maximum search on the empty partitions – or on partitions for which $part[j] = 0$. Since the condition in row 4 of the LOAD-LARGEST algorithm has to be evaluated for each j, the conditional maximum search runs in $\Theta(n)$. Although the loading algorithm will not be called for partitions for which $part[j] > 0$, as far as running time is concerned, in the worst case even all the partitions might be empty, therefore the time complexity of our algorithm is $\Theta(mn)$.

Unfortunately, the fact that the algorithm fills all the empty partitions with waiting processes fitting into them whenever possible is not always sufficient. A very usual requirement is that the execution of every process should be started within a determined time limit. The above algorithm does not ensure it, even if there is an upper limit for the execution time of the processes. The problem is that whenever the algorithm is executed, there might always be new processes that prevent the ones waiting for long from execution. This is shown in the following example.

Example 17.1 Suppose that we have two partitions with sizes of 5 kB and 10 kB. We also have two processes with space requirements of 8 kB and 9 kB. The execution time of both processes is 2 seconds. But at the end of the first second a new process appears with space requirement of 9 kB and execution time of 2 seconds again, and the same happens in every 2 seconds, i. e., in the third, fifth, etc. second. If we have a look at our algorithm, we can see that it always has to choose between two processes, and the one with space requirement of 9 kB will always be the winner. The other one with 8 kB will never get into the memory, although there is no other partition into which it would fit.

To be able to fulfill this new requirement mentioned above, we have to slightly modify our algorithm: the long waiting processes must be preferred over all the other processes, even if their space requirement is smaller than that of the others. Our new algorithm will process all the partitions, just like the previous one.

LARGEST-OR-LONG-WAITING-FIT(*place*, *spacereq*, *threshold*, *size*, *part*, *waiting*)

```
1  for j ← 1 to m
2      do if part[j] = 0
3            then LOAD-LARGEST-OR-LONG-WAITING( place, spacereq, threshold,
                                                size, j, part, waiting)
```

However, this time we keep track on the waiting time of each process. Since the algorithm is only executed when one or more partitions become free, we cannot examine the concrete time, but the number of cases where the process would have fit into a partition but we have chosen another process to fill it. To implement this, the conditional maximum search algorithm has to be modified: operations have to be performed also on items that meet the requirement (they are waiting for memory and they would fit), but they are not the largest ones among those. This operation is a simple increment of the value of a counter. We assume that the value of the counter is 0 when the process starts. The condition of the search has to be modified as well: if the value of the counter of a process is too high, (i. e., higher than a certain *threshold*), and it is higher than the value of the counter of the process with the largest space requirement found so far, then we replace it with this new process. The pseudo code of the algorithm is the following:

LOAD-LARGEST-OR-LONG-WAITING(*place*, *spacereq*, *threshold*, *size*, *p*, *part*, *waiting*)

```
1  max ← 0
2  ind ← 0
3  for i ← 1 to n
4      do if waiting[i] and spacereq[i] ≤ size[p]
5            then if (points[i] > threshold and points[i] > points[ind]) or
                     spacereq[i] > max
6                 then points[ind] ← points[ind] + 1
7                      ind ← i
8                      max ← spacereq[i]
9                 else points[i] ← points[i] + 1
10 if ind > 0
11   then part[p] ← ind
12        place[ind] ← p
13        waiting[ind] ← FALSE
```

The fact that the algorithm does not place multiple processes into the same partition can be proved the same way as for the previous algorithm, since the outer loop and the condition of the branch has not been changed. To prove the other two criteria (namely that a process will be placed neither into more then one partitions, nor into a partition into which it does not fit), we have to see that the condition of the conditional maximum search algorithm has been modified in a way that this property stays. It is easy to see that the condition has been split into two parts,

so the first part corresponds exactly to our requirement, and if it is not satisfied, the algorithm certainly does not place the process into the partition. The property that there are no partitions left empty also stays, since the condition for choosing a process has not been restricted, but extended. Therefore, if the previous algorithm found all the processes that met the requirements, the new one finds them as well. Only the order of the processes fulfilling the criteria has been altered. The time complexity of the loops has not changed either, just like the condition, according to which the inner loop has to be executed. So the time complexity of the algorithm is the same as in the original case.

We have to examine whether the algorithm satisfies the condition that a process can wait for memory only for a given time, if we suppose that there is some p upper limit for the execution time of the processes (otherwise the problem is insoluble, since all the partitions might be taken by an infinite loop). Furthermore, let us suppose that the system is not overloaded, i. e., we can find a q upper estimation for the number of the waiting processes in every instant of time. Knowing both limits it is easy to see that in the worst case to get assigned to a given partition a process has to wait for the processes with higher counters than its own one (at most q many), and at most *threshold* many processes larger than itself. Therefore, it is indeed possible to give an upper limit for the maximum waiting time for memory in the worst case: it is $(q + \mathit{threshold})p$.

Example 17.2 In our previous example the process with space requirement of 8 kB has to wait for $\mathit{threshold} + 1 = k$ other processes, all of which lasts for 2 seconds, i. e., the process with space requirement of 8 kB has to wait exactly for 2k seconds to get into the partition with size of 10 kB.

In our algorithms so far the absolute space requirement of the processes served as the basis of their priorities. However this method is not fair: if there is a partition, into which two processes would fit, and neither of them fits into a smaller partition, then the difference in their size does not matter, since sooner or later also the smaller one has to be placed into the same, or into another, but not smaller partition. Therefore, instead of the absolute space requirement, the size of the smallest partition into which the given process fits should be taken into consideration when determining the priorities. Furthermore, if the partitions are increasingly ordered according to their sizes, then the index of the smallest partition in this ordered list is the priority of the process. It is called the rank of the process. The following algorithm calculates the ranks of all the processes.

```
CALCULATE-RANK(spacereq, size, rank)
 1  order ← SORT(size)
 2  for i ← 1 to n
 3      do u ← 1
 4         v ← m
 5         rank[i] ← ⌊(u + v)/2⌋
 6         while order[rank[i]] < spacereq[i] or order[rank[i] + 1] > spacereq[i]
 7               do if order[rank[i]] < spacereq[i]
 8                     then u ← rank[i] + 1
 9                     else v ← rank[i] − 1
10               rank[i] ← ⌊(u + v)/2⌋
```

It is easy to see that this algorithm first orders the partitions increasingly according to their sizes, and then calculates the rank for each process. However, this has to be done only at the beginning, or when a new process comes. In the latter case the inner loop has to be executed only for the new processes. Ordering of the partitions does not have to be performed again, since the partitions do not change. The only thing that must be calculated is the smallest partition the process fits into. This can be solved by a logarithmic search, an algorithm whose correctness is proved. The time complexity of the rank calculation is easy to determine: the ordering of the partition takes $\Theta(m \log_2 m)$ steps, while the logarithmic search $\Theta(\log_2 m)$, which has to be executed for n processes. Therefore the total number of steps is $\Theta((n + m) \log_2 m)$.

After calculating the ranks we have to do the same as before, but for ranks instead of space requirements.

```
LONG-WAITING-OR-NOT-FIT-SMALLER(place, spacereq, threshold, size, part,
waiting)
1  for j ← 1 to m
2      do if part[j] = 0
3            then LOAD-LONG-WAITING-OR-NOT-SMALLER( place, spacereq,
                                                    threshold, size, j,
                                                    part, waiting)
```

In the loading algorithm, the only difference is that the conditional maximum search has to be executed not on array *size*, but on array *rank*:

LOAD-LONG-WAITING-OR-NOT-SMALLER(*place*, *spacereq*, *threshold*, *size*, *p*, *part*, *waiting*)

```
 1  max ← 0
 2  ind ← 0
 3  for i ← 1 to n
 4      do if waiting[i] and spacereq[i] ≤ size[p]
 5            then if (points[i] > threshold and points[i] > points[ind]) or
                       rank[i] > max
 6                 then points[ind] ← points[ind] + 1
 7                      ind ← i
 8                      max ← rank[i]
 9                 else points[i] ← points[i] + 1
10  if ind > 0
11     then part[p] ← ind
12          place[ind] ← p
13          waiting[ind] ← FALSE
```

The correctness of the algorithm follows from the previous version of the algorithm and the algorithm calculating the rank. The time complexity is the same as that of the previous versions.

Example 17.3 Having a look at the previous example it can be seen that both the processes with space requirement of 8 kB and 9 kB can fit only into the partition with size of 10 kB, and cannot fit into the 5 kB one. Therefore their ranks will be the same (it will be two), so they will be loaded into the memory in the order of their arrival, which means that the 8 kB one will be among the first two.

17.1.2. Dynamic partitions

Dynamic partitioning works in a totally different way from the fixed one. Using this method we do not search for the suitable processes for every empty partition, but search for suitable memory space for every waiting process, and there we create partitions dynamically. This section is restricted to the terminology of operating systems as well, but of course, the algorithms can be rewritten to solve problems connected at the application level as well.

If all the processes would finish at the same time, there would not be any problems, since the empty memory space could be filled up from the bottom to the top continuously. Unfortunately, however, the situation is more complicated in the practice, as processes can differ significantly from each other, so their execution time is not the same either. Therefore, the allocated memory area will not always be contiguous, but there might be free partitions between the busy ones. Since copying within the memory is an extremely expensive operation, in practice it is not effective to collect the reserved partitions into the bottom of the memory. Collecting the partitions often cannot even be carried out due to the complicated relative addressing

methods often used. Therefore, the free area on which the new processes have to be placed is not contiguous. It is obvious, that every new process must be assigned to the beginning of a free partition, but the question is, which of the many free partitions is the most suitable.

Partitions are the simplest to store in a linked list. Naturally, many other, maybe more efficient data structures could be found, but this is sufficient for the presentation of the algorithms listed below. The address of the first element of linked list P is stored in $head[P]$. The beginning of the partition at address p is stored in $beginning[p]$, its size in $size[p]$, and the process assigned to it is stored in variable $part[p]$. If the identifier of a process is 0, then it is an empty one, otherwise it is a allocated. In the linked list the address of the next partition is $next[p]$.

To create a partition of appropriate size dynamically, first we have to divide a free partition, which is at least as big as needed into two parts. This is done by the next algorithm.

SPLIT-PARTITION($border$, $beginning$, $next$, $size$, p, q)

```
1  beginning[q] ← beginning[p] + border
2  size[q] ← size[p] − border
3  size[p] ← border
4  next[q] ← next[p]
5  next[q] ← q
```

In contrast to the algorithms connected to the method of fixed partitions, where processes were chosen to partitions, here we use a reverse approach. Here we inspect the list of the processes, and try to find to each waiting process a free partition into which it fits. If we found one, we cut the required part off from the beginning of the partition, and allocate it to the process by storing its beginning address in the process header. If there is no such free partition, then the process remains in the waiting list.

PLACE(P, $head$, $next$, $last$, $beginning$, $size$, $part$, $spacereq$, $place$)

```
1  for i ← 1 to n
2      do if waiting[i] = TRUE
3           then ⋆-FIT( P, head, next, last, beginning, size, part, spacereq, place,
                        waiting, i)
```

The $\star$ in the pseudo code is to be replaced by one of the words FIRST, NEXT, BEST, LIMITED-BEST, WORST or LIMITED-WORST.

There are several possibilities for choosing the suitable free partition. The more simple idea is to go through the list of the partitions from the beginning until we find the first free partition into which it fits. This can easily be solved using linear searching.

```
FIRST-FIT(P, head, next, last, beginning, size, part, spacereq, place, waiting, f)
1  p ← head[P]
2  while waiting[f] = TRUE and p ≠ NIL
3        do if part[p] = 0 and size[p] ≥ spacereq[f]
4              then SPLIT-PARTITION(p,q,spacereq[f])
5                   part[p] ← f
6                   place[f] ← p
7                   waiting[f] ← FALSE
8           p ← next[p]
```

To prove the correctness of the algorithm several facts have to be examined. First, we should not load a process into a partition into which it does not fit. This is guaranteed by the linear search theorem, since this criteria is part of the property predicate.

Similarly to the fixed partitioning, the most essential criteria of correctness is that one single process should not be placed into multiple partitions simultaneously, and at most one processes may be placed into one partition. The proof of this criteria is word by word the same as the one stated at fixed partitions. The only difference is that instead of the conditional maximum search the linear search must be used.

Of course, these conditions are not sufficient in this case either, since they are fulfilled by even the empty algorithm. We also need prove that the algorithm finds a place for every process that fits into any of the partitions. For this we need an invariant again: after examining j processes, there is no positive $k \leq j$, for which *waiting*[k], and for which there is a p partition, such that *part*[p] $= 0$, and *size*[p] $\geq$ *spacereq*[k].

Initialisation: At the beginning of the algorithm we have examined $j = 0$ many partitions, so there is no positive $k \leq j$.

Maintenance: If the invariant holds for j at the beginning of the loop, first we have to check whether it holds for the same j at the end of the loop as well. It is obvious, since the first j processes are not modified when examining the $(j+1)$-th one, and for the partitions containing them *part*[p] > 0, which does not satisfy the predicate of the linear search in the FIRST-FIT algorithm. The invariant statement holds for the $(j+1)$-th process at the end of the loop as well, since if there is a free memory slice which fulfills the condition, the linear search certainly finds it, because the condition of our linear search corresponds to the requirement of our invariant set on each partition.

Termination: Since the loop traverses a fixed interval by one, it certainly stops. Since the loop body is executed exactly as many times as the number of the processes, after the loop has finished, it holds that there is no positive $k \leq j$, for which *waiting*[k], and for which there is a p partition, such that *part*[p] $= 0$, and *size*[p] $\geq$ *spacereq*[i], which means that we did not keep any processes fitting into any of the partitions waiting.

Again, the time complexity of the algorithm can be calculated easily. We examine all the n processes in any case. If, for instance, all the processes are waiting, and the

partitions are all reserved, the algorithm runs in $\Theta(nm)$.

However, when calculating the time complexity, we failed to take some important points of view into consideration. One of them is that m is not constant, but executing the algorithm again and again it probably increases, since the processes are independent of each other, start and end in different instances of time, and their sizes can differ considerably. Therefore, we split a partition into two more often than we merge two neighbouring ones. This phenomenon is called ***fragmentation the memory***. Hence, the number of steps in the worst case is growing continuously when running the algorithm several times. Furthermore, linear search divides always the first partition with appropriate size into two, so after a while there will be a lot of small partitions at the beginning of the memory area, unusable for most processes. Therefore the average execution time will grow as well. A solution for the latter problem is to not always start searching at the beginning of the list of the partitions, but from the second half of the partition split last time. When reaching the end of the list, we can continue at the beginning until finding the first suitable partition, or reaching the starting partition again. This means we traverse the list of the partitions cyclically.

NEXT-FIT(P, *head*, *next*, *last*, *beginning*, *size*, *part*, *spacereq*, *place*, *waiting*, f)

```
 1  if last[P] ≠ NIL
 2     then p ← next[last[P]]
 3     else p ← head[P]
 4  while waiting[f] and p ≠ last[P]
 5        do if p = NIL
 6              then p ← head[P]
 7           if part[p] = 0 and size[p] ≥ spacereq[f]
 8              then SPLIT-PARTITION(p,q,spacereq[f])
 9                   part[p] ← f
10                   place[f] ← p
11                   waiting[f] ← FALSE
12                   last[P] ← p
13           p ← next[p]
```

The proof of the correctness of the algorithm is basically the same as that of the FIRST-FIT, as well as its time complexity. Practically, there is a linear search in the inner loop again, only the interval is always rotated in the end. However, this algorithm traverses the list of the free areas evenly, so does not fragment the beginning of the list. As a consequence, the average execution time is expected to be smaller than that of the FIRST-FIT.

If the only thing to be examined about each partition is whether a process fits into it, then it can easily happen that we cut off large partitions for small processes, so that there would not be partitions with appropriate sizes for the later arriving larger processes. Splitting unnecessarily large partitions can be avoided by assigning each process to the smallest possible partition into which it fits.

BEST-FIT(P, *head*, *next*, *last*, *beginning*, *size*, *part*, *spacereq*, *place*, *waiting*, f)

```
1  min ← ∞
2  ind ← NIL
3  p ← head[P]
4  while p ≠ NIL
5        do if part[p] = 0 and size[p] ≥ spacereq[f] and size[p] < min
6              then ind ← p
7                   min ← size[p]
8           p ← next[p]
9  if ind ≠ NIL
10   then SPLIT-PARTITION(ind,q,spacereq[f])
11        part[ind] ← f
12        place[f] ← ind
13        waiting[f] ← FALSE
```

All the criteria of the correctness of the algorithm can be proved in the same way as previously. The only difference from the FIRST-FIT is that conditional minimum search is applied instead of linear search. It is also obvious that this algorithm will not split a partition larger than minimally required.

However, it is not always efficient to place each process into the smallest space into which it fits. It is because the remaining part of the partition is often too small, unsuitable for most of the processes. It is disadvantageous for two reasons. On the one hand, these partitions are still on the list of free partitions, so they are examined again and again whenever searching for a place for a process. On the other hand, many small partitions together compose a large area that is useless, since it is not contiguous. Therefore, we have to somehow avoid the creation of too small free partitions. The meaning of too small can be determined by either a constant or a function of the space requirement of the process to be placed. (For example, the free area should be twice as large as the space required for the process.) Since this limit is based on the whole partition and not only its remaining part, we will always consider it as a function depending on the process. Of course, if there is no partition to fulfill this extra condition, then we should place the process into the largest partition. So we get the following algorithm.

LIMITED-BEST-FIT(P, *head*, *next*, *last*, *beginning*, *size*, *part*, *spacereq*, *place*, *waiting*, f)

```
1  min ← ∞
2  ind ← NIL
3  p ← head[P]
4  while p ≠ NIL
```

```
5     do if part[p] = 0 and size[p] ≥ spacereq[f] and
                        ((size[p] < min and size[p] ≥ Limit(f))
                        or ind = nil or (min < Limit(f) and size[p] > min))
6           then ind ← p
7                min ← size[p]
8        p ← next[p]
9  if ind ≠ nil
10   then Split-Partition(ind,q,spacereq[f])
11        part[ind] ← f
12        place[f] ← ind
13        waiting[f] ← false
```

This algorithm is more complicated than the previous ones. To prove its correctness we have to see that the inner loop is a conditional minimum searching. The first part of the condition, i. e. that $part[p] = 0$, and $size[p] \geq spacereq[f]$ means that we try to find a free partition suitable for the process. The second part is a disjunction: we replace the item found so far with the newly examined one in three cases. The first case is when $size[p] < min$, and $size[p] \geq \textsc{Limit}(spacereq[f])$, which means that the size of the examined partition is at least as large as the described minimum, but it is smaller than the the smallest one found so far. If there were no more conditions, this would be a conditional minimum search for the conditions of which we added that the size of the partition should be above a certain limit. But there are two other cases, when we replace the previously found item to the new one. One of the cases is that $ind = \textsc{nil}$, i. e., the newly examined partition is the first one which is free, and into which the process fits. This is needed because we stick to the requirement that if there is a free partition suitable for the process, then the algorithm should place the process into such a partition. Finally, according to the third condition, we replace the previously found most suitable item to the current one, if $min < \textsc{Limit}(spacereq[f])$ and $size[p] > min$, which means that the minimum found so far did not reach the described limit, and the current item is bigger than this minimum. This condition is important for two reasons. First, if the items examined so far do not fulfill the most recent condition, but the current one does, then we replace it, since in this case $min < \textsc{Limit}(spacereq[f]) \leq size[p]$, i. e., the size of the current partition is obviously larger. Second, if neither the size of partition found so far, nor that of the current one reaches the described limit, but the currently examined one approaches it better from below, then $min < size[p] < \textsc{Limit}(spacereq[f])$ holds, therefore, also in this case we replace the item found so far by the current one. Hence, if there are partitions at least as large as the described limit, then the algorithm places each process into the smallest one among them, and if there is no such partition, then in the largest suitable one.

There are certain problems, where the only requirement is that the remaining free spaces should be the largest possible. It can be guaranteed if each process is placed into the largest free partition:

WORST-FIT(*P*, *head*, *next*, *last*, *beginning*, *size*, *part*, *spacereq*, *place*, *waiting*, *f*)

```
1  max ← 0
2  ind ← NIL
3  p ← head[P]
4  while p ≠ NIL
5        do if part[p] = 0 and size[p] ≥ spacereq[f] and size[p] > max
6              then ind ← p
7                   min ← size[p]
8           p ← next[p]
9  if ind ≠ NIL
10   then SPLIT-PARTITION(ind,q,spacereq[f])
11        part[ind] ← f
12        place[f] ← ind
13        waiting[f] ← FALSE
```

We can prove the correctness of the algorithm similarly to the BEST-FIT algorithm; the only difference is that maximum search has to be used instead of conditional maximum search. As a consequence, it is also obvious that the sizes of the remaining free areas are maximal.

The WORST-FIT algorithm maximises the smallest free partition, i. e. there will be only few partitions which are too small for most of the processes. It follows from the fact that it always splits the largest partitions. However, it also often prevents large processes from getting into the memory, so they have to wait on an auxiliary storage. To avoid this we may extend our conditions with an extra an one, similarly to the BEST-FIT algorithm. In this case, however, we give an upper limit instead of a lower one. The algorithm only tries to split partitions smaller than a certain limit. This limit also depends on the space requirement of the process. (For example the double of the space requirement.) If the algorithm can find such partitions, then it chooses the largest one to avoid creating too small partitions. If it finds only partitions exceeding this limit, then it splits the smallest one to save bigger ones for large processes.

LIMITED-WORST-FIT(*f*, *beginning*, *head*, *place*, *spacereq*, *next*, *size*, *part*, *waiting*, *waiting*)

```
1  max ← 0
2  ind ← NIL
3  p ← head[P]
```

```
4  while p ≠ NIL
5        do if part[p] = 0 and size[p] ≥ spacereq[f] and
                              ((size[p] > max and size[p] ≤ LIMIT(f)) or ind = NIL or
                              (max > LIMIT(f) and size[p] < max))
6              then ind ← p
7                   min ← size[p]
8           p ← next[p]
9  if ind ≠ NIL
10   then SPLIT-PARTITION(ind,q,spacereq[f])
11        part[ind] ← f
12        place[f] ← ind
13        waiting[f] ← FALSE
```

It is easy to see that this algorithm is very similar to the LIMITED-BEST-FIT, only the relation signs are reversed. The difference is not significant indeed. In both algorithms the same two conditions are to be fulfilled: there should not be too small partitions, and large free partitions should not be wasted for small processes. The only difference is which condition is taken account in the first place and which in the second. The actual problem decides which one to use.

Exercises

17.1-1 We have a system containing two fixed partitions with sizes of 100 kB, one of 200 kB and one of 400 kB. All of them are empty at the beginning. One second later five processes arrive almost simultaneously, directly after each other without significant delay. Their sizes are 80 kB, 70 kB, 50 kB, 120 kB and 180 kB respectively. The process with size of 180 kB ends in the fifth second after its arrival, but by that time another process arrives with space requirement of 280 kB. Which processes are in which partitions in the sixth second after the first arrivals, if we suppose that other processes do not end until that time, and the LARGEST-FIT algorithm is used? What is the case if the LARGEST-OR-LONG-WAITING-FIT or the LONG-WAITING-OR-NOT-FIT-SMALLER algorithm is used with threshold value of 4?

17.1-2 In a system using dynamic partitions the list of free partition consists of the following items: one with size of 20 kB, followed by one of 100 kB, one of 210 kB, one of 180 kB, one of 50 kB, one of 10 kB, one of 70 kB, one of 130 kB and one of 90 kB respectively. The last process was placed into the partition preceding the one of 180 kB. A new process with space requirement of 40 kB arrives into the system. Into which partition is it to be placed using the FIRST-FIT, NEXT-FIT, BEST-FIT, LIMITED-BEST-FIT, WORST-FIT or the LIMITED-WORST-FIT algorithms?

17.1-3 An effective implementation of the WORST-FIT algorithm is when the partitions are stored in a binary heap instead of a linear linked list. What is the time complexity of the PLACE algorithm perform in this case?

17.2. Page replacement algorithms

As already mentioned, the memory of the modern computer systems consists of several levels. These levels usually are organised into a seemingly single-level memory, called ***virtual memory***. Users do not have to know this structure with several levels in detail: operating systems manage these levels.

The most popular methods to control this virtual memory are paging and segmentation. Paging divides both memory levels into fixed-sized units, called ***frames***. In this case programs are also divided into parts of the same size as frams have: these parts of the programs (and data) are called ***pages***. Segmentation uses parts of a program with changing size—these parts are called ***segments***.

For the simplicity let us suppose that the memory consists of only two levels: the smaller one with shorter access time is called ***main memory*** (or ***memory*** for short), and the larger one with larger access time is called ***backing memory*** .

At the beginning, the main memory is empty, and there is only one program consisting of n parts in the backing memory. Suppose that during the run of the program there are instructions to be executed, and the execution of each instruction there requires an access to a certain page of the program. After processing the reference string, the following problems have to be solved.

1. Where should we place the segment of the program responsible for executing the next instruction in the main memory (if it is not there)?
2. When should we place the segments of the program in the main memory?
3. How should we deallocate space for the segments of the program to be placed into the main memory?

It is the placing algorithms that give the answer to the first question: as far as paging is concerned, the answer is simply anywhere—since the page frames of the main memory are of the same size and access time. During segmentation there are program segments and free memory areas, called holes alternating in the main memory–and it is the segment placing algorithms that gives the answer to the first question.

To the second question the answer is given by the transferring algorithms: in working systems the answer is on demand in most of the cases, which means that a new segment of the program starts to be loaded from the backing memory when it turns out that this certain segment is needed. Another solution would be preloading, but according to the experiences it involves a lot of unnecessary work, so it has not become wide-spread.

It is the replacement algorithms that give the answer to the third question: as far as paging is concerned, these are the page replacement algorithms, which we present in this section. Segment replacement algorithms used by segmentation apply basically the ideas of page replacement algorithms—completed them according to the different sizes of the segments.

Let us suppose that the size of the physical memory is m page frames, while that of the backing memory is n page frames. Naturally the inequality $1 \leq m \leq n$ holds for the parameters. In practice, n is usually many times bigger than m. At the beginning the main memory is empty, and there is only one program in the backing

memory. Suppose that during the run of the program there are p instructions to be executed, and to execute the t-th instruction ($1 \leq t \leq p$) the page r_t is necessary, and the result of the execution of the instruction also can be stored in the same page, i. e., we are modelling the execution of the program by ***reference string*** $R = \langle r_1, r_2, \ldots, r_p \rangle$. In the following we examine only the case of demand paging, to be more precise, only the page replacement algorithms within it.

If it is important to differentiate reading from writing, we will use ***writing array*** $W = \langle w_1, w_2, \ldots, w_p \rangle$ besides array R. Entry w_t of array W is TRUE if we are writing onto page r_t, otherwise $w_1 =$ FALSE.

Demand paging algorithms fall into two groups; there are ***static*** and ***dynamic*** algorithms. At the beginning of the running of the program both types fill the page frames of the physical memory with pages, but after that static algorithms keep exactly m page frames reserved until the end of the running, while dynamic algorithms allocate at most m page frames.

17.2.1. Static page replacement

The input data of static page replacement algorithms are: the size of the main memory measured in number of the page frames (m), the size of the program measured in number of of pages (n), the running time of the program measured in number of instructions (p) and the reference string (R); while their output is the ***number of the page faults*** (*pagefault*).

Static algorithms are based on managing the ***page table***. The page table is a matrix with size of $n \times 2$, the i-th ($i \in [0\,..\,n-1]$) row of which refers to the i-th page. The first entry of the row is a logical variable (***present/absent bit***), the value of which keeps track of whether the page is in the main memory in that certain instant of time: if the i-th page is in the main memory, then $pagetable[i,1] =$ TRUE and $pagetable[i,2] = j$, where $j \in [0\,..\,m-1]$ shows us that the page is in the j-th page frame of the main memory. If the i-th page is not in the main memory, then $pagetable[i,1] =$ FALSE and $pagetable[i,2]$ is non-defined. Work variable *busy* contains the number of the busy page frames
indexframe!free of the main memory.

If the size of the pages is z, then the physical address f can be calculated from virtual address v so that $j = \lfloor v/z \rfloor$ gives us the ***index of the virtual page frame***, and $v - z\lfloor v/z \rfloor$ gives us offset s referring to virtual address v. If the j-th page is in the main memory in the given instant of time—which is indicated by $pagetable[i,1] =$ TRUE—, then $f = s + z \cdot pagetable[i,2]$. If, however, the i-th page is not in the main memory, then a page fault occurs. In this case we choose one of the page frames of the main memory using the page replacement algorithm, load the j-th page into it, refresh the j-th row of the page table and then calculate f.

The operation of the demand paging algorithms can be described by a Mealy automaton having an initial status. This automaton can be given as $(Q, q_0, X, Y, \delta, \lambda)$, where Q is the set of the control states, $q_o \in Q$ is the initial control state, X is the input alphabet, Y is the output alphabet, $\delta : Q \times X \to Q$ is the state transition function and $\lambda : Q \times X \to Y$ is the output function.

We do not discuss the formalisation of how the automaton stop.

Sequence $R_p = \langle r_1, r_2, \ldots, r_p \rangle$ (or $R_p = \langle r_1, r_2, \ldots, r_\infty \rangle$) is called ***reference string***.

The description of the algorithms can be simplified introducing memory states S_t ($t = 1, 2, \ldots$): this state is the set of the pages stored in the main memory of the automat after processing the t-th input sign. In the case of static demand paging algorithms $S_0 = \emptyset$. If the new memory status differs from the old one (which means that a new page had to be swapped in), then a page fault has occurred. Consequently, both a swapping of a page into an empty frame and page replacement are called page fault.

In case of page replacement algorithms—according to Denning's proposition—instead of λ and δ we use the state transition function $g_P : Q \times M \times X \to Q \times Y$. Since for the page replacement algorithms $X = \{0, 1, \ldots, n-1\}$ and $Y = X \cup \emptyset$, holds, these two items can be omitted from the definition, so page replacement algorithm P can be described by the triple (Q, q_0, g_P).

Our first example is one of the simplest page replacement algorithms, the FIFO (**F**irst **I**n **F**irst **O**ut), which replaces the pages in the order of their loading in. Its definition is the following: $q_0 = \langle\rangle$ and

$$g_{\text{FIFO}}(S, q, x) = \begin{cases} (S, q, \epsilon), & \text{if } x \in S\ , \\ (S \cup \{x\}, q', \epsilon), & \text{if } x \notin S,\ |S| = k < m\ , \\ (S \setminus \{y_1\} \cup \{x\}, q", y_1), & \text{if } x \notin S \text{ and } |S| = k = m\ , \end{cases} \tag{17.1}$$

where $q = \langle y_1, y_2, \ldots, y_k \rangle$, $q' = \langle y_1, y_2, \ldots, y_k, x \rangle$ and $q'' = \langle y_2, y_3, \ldots, y_m, x \rangle$.

Running of the programs is carried out by the following *-RUN algorithm. In this section the * in the name of the algorithms has to be replaced by the name of the page replacement algorithm to be applied (FIFO, LRU OPT, LFU or NRU). In the pseudocodes it is supposed that the called procedures know the values of the variable used in the calling procedure, and the calling procedure accesses to the new values.

*-RUN($m, n, p, R, faultnumber, pagetable$)

```
1  faultnumber ← 0
2  busy ← 0
3  for i ← 0 to n − 1                          ▷ Preparing the pagetable.
4       do pagetable[i, 1] ← FALSE
5  *-PREPARE(pagetable)
6  for i ← 1 to p                              ▷ Run of the program.
7       do *-EXECUTES(pagetable, i)
8  return faultnumber
```

The following implementation of the algorithm keeps track of the order of loading in the pages by queue Q. The preparing algorithm has to create the empty queue, i. e., to execute the instruction $Q \leftarrow \emptyset$.

In the following pseudocode *swap-out* is the index of the page to be replaced, and *swap-in* is the index of the page of the main memory into which the new page is going to be swapped in.

FIFO-EXECUTES($pagetable, t$),

```
1  if pagetable[r_t, 1] = TRUE                          ▷ The next page is in.
2     then NIL
3  if pagetable[r_t, 1] = FALSE                         ▷ The next page is out.
4     then pagefault ← pagefault + 1
5          if busy < m                                  ▷ Main memory is not full.
6             then INQUEUE(Q, r_t)
7                  swap-in ← busy
8                  busy ← busy + 1
9          if busy = m                                  ▷ Main memory is full.
10            then replaces ← ENQUEUE(Q)
11                 pagetable[swap-out, 1] ← FALSE
12                 swap-in ← pagetable[swap-out, 2]
13         WRITE(swap-in, swap-out)
14         READ(r_t, load)                              ▷ Reading.
15         pagetable[r_t, 1] ← TRUE                     ▷ Updating of the data.
16         pagetable[r_t, 2] ← loads
```

Procedure writing writes the page chosen to be swapped out into the backing memory: its first parameter answers the question where from (from which page frame of the memory) and its second parameter answers where to (to which page frame of the backing memory). Procedure READING reads the page needed to execute the next instruction from the backing memory into the appropriate page frame of the physical memory: its first parameter is where from (from which page frame of the backing memory) and its second parameter is where to (to which page frame of the memory). When giving the parameters of both the procedures we use the fact that the page frames are of the same size, therefore, the initial address of the j-th page frame is j-times the page size z in both memories. Most of the page replacement algorithms do not need to know the other entries of reference string R to process reference r_t, so when calculating space requirement we do not have to take the space requirement of the series into consideration. An exception for this is algorithm OPT for example. The space requirement of the FIFO-RUN algorithm is determined by the size of the page frame - this space requirement is $O(m)$. The running time of the FIFO-RUN algorithm is de-termined by the loop. Since the procedure called in rows 6 and 7 performs only a constant number of steps (provided that queue-handling operations can be performed in $O(1)$, the run-ning time of the FIFO-RUN algorithm is $O(p)$. Note that some of the pages do not change while being in the memory, so if we assign a modified bit to the pages in the memory, then we can spare the writing in row 12 in some of the cases.

Our next example is one of the most popular page replacement algorithms, the LRU (**L**east **R**ecently **U**sed), which replaces the page used least recently. Its definition is the following: $q_0 = ()$ and

$$g_{\mathrm{LRU}}(S,q,x) = \begin{cases} (S, q''', \epsilon), & \text{if } x \in S\ , \\ (S \cup \{x\}, q', \epsilon), & \text{if } x \notin S,\ |S| = k < m\ , \\ (S \setminus \{y_1\} \cup \{x\}, q", y_1), & \text{if } x \notin S \text{ and } |S| = k = m\ , \end{cases} \tag{17.2}$$

where $q = \langle y_1, y_2, \ldots, y_k \rangle$, $q' = \langle y_1, y_2, \ldots, y_k, x \rangle$, $q'' = \langle y_2, y_3, \ldots, y_m, x \rangle$ and if $x = y_k$, then $q''' = \langle y_1, y_2, \ldots, y_{k-1}, \ldots, y_{k+1} \ldots y_m, y_k \rangle$.

The next implementation of LRU does not need any preparations. We keep a record of the time of the last usage of the certain pages in array *last-call*$[0..n-1]$, and when there is a replacement needed, the least recently used page can be found with linear search.

LRU-EXECUTES(*pagetable*, t)

```
 1  if pagetable[r_t, 1] = TRUE                          ▷ The next page is in.
 2     then last-ref[r_t] ← t
 3  if pagetable[r_t, 1] = FALSE                         ▷ The next page is not in.
 4     then pagefault ← pagefault + 1
 5          if busy < m                                  ▷ The physical memory is not full.
 6             then swap-in ← busy
 7                  busy ← busy + 1
 8          if busy = m                                  ▷ The physical memory is full.
 9             then swap-out ← r_{t-1}
10                  for i ← 0 to n − 1
11                      do if pagetable[i, 1] = TRUE and
                              last-ref[i] < last-ref[swap-out]
12                           then swap-out ← last-ref[i]
13                  pagetable[swap-out, 1] ← FALSE
14                  swap-in ← pagetable[swap-out, 2]
15                  WRITE(swap-in, swap-out)
16          READ(r_t, swap-in)                           ▷ Reading.
17          pagetable[r_t, 1] ← TRUE                     ▷ Updating.
18          pagetable[r_t, 2] ← swap-in
19          last-ref[r_t] ← t
```

If we consider the values of both n and p as variables, then due to the linear search in rows 10–11, the running time of the LRU-RUN algorithm is $O(np)$.

The following algorithm is optimal in the sense that with the given conditions (fixed m and n) it causes a minimal number of page faults. This algorithm chooses the page from the ones in the memory, which is going to be used at the latest (if there are several page that are not needed any more, then we choose the one at the lowest memory address from them) to be replaced. This algorithm does not need any preparations either.

OPT-EXECUTES$(t, pagetable, R)$

```
 1  if pagetable[r_t, 1] = TRUE                          ▷ The next page is in.
 2     then NIL
 3  if pagetable[r_t, 1] = FALSE                         ▷ The next page is not in.
 4     then pagefault ← pagefault + 1
 5          if busy < m                                  ▷ The main memory is not full.
 6             then swap-in ← busy
 7                  busy ← busy + 1
 8          if busy = m                                  ▷ The main memory is full.
 9             then OPT-SWAP-OUT(t, R)
10                  pagetable[swap-out, 1] ← FALSE
11                  swap-in ← pagetable[swap-out, 2]
12                  WRITE(swap-in, swap-out)
13          READ(r_t, swap-in)                           ▷ Reading.
14          pagetable[r_t, 1] ← TRUE                     ▷ Updating.
15          pagetable[r_t, 2] ← swap-in
```

Procedure OPT-SWAP-OUT determines the index of the page to be replaced.

OPT-SWAP-OUT(t, R)

```
 1  guarded ← 0                                          ▷ Preparation.
 2  for j ← 0 to m − 1
 3       do frame[j] ← FALSE
 4  s ← t + 1                          ▷ Determining the protection of the page frames.
 5  while s ≤ p and pagetable[r_s, 1] = TRUE and frame[pagetable[r_s, 2]] = FALSE and
                guarded < m − 1
 6          do guarded ← guarded + 1
 7             frame[r_s] ← TRUE
 8             s ← s + 1
 9  swap-out ← m − 1        ▷ Finding the frame containing the page to be replaced.
10  j ← 0
11  while frame[j] = TRUE
12          do j ← j + 1
13  swap-out ← j
14  return swap-out
```

Information about pages in the main memory is stored in $frame[0\,..\,m-1]$: $frame[j]$ = TRUE means that the page stored in the j-th frame is protected from being replaced due to its going to be used soon. Variable protected keeps track of how many protected pages we know about. If we either find $m-1$ protected pages or reach the end of R, then we will choose the unprotected page at the lowest memory address for the page to be replaced.

Since the OPT algorithm needs to know the entire array R, its space requirement is $O(p)$. Since in rows 5–8 of the OPT-SWAP-OUT algorithm at most the remaining part of R has to be looked through, the running time of the OPT-SWAP-OUT algorithm is $O(p^2)$. The following LFU (Least Frequently Used) algorithm chooses the

least frequently used page to be replaced. So that the page replacement would be obvious we suppose that in the case of equal frequencies we replace the page at the lowest address of the physical memory. We keep a record of how many times each page has been referenced since it was loaded into the physical memory with the help of array frequency[1..n - 1]. This algorithm does not need any preparations either.

LFU-EXECUTES($pagetable, t$)

```
 1  if pagetable[r_t, 1] = TRUE                          ▷ The next page is in.
 2     then frequency[r_t] ← frequency[r_t] + 1
 3  if pagetable[r_t, 1] = FALSE                         ▷ The next page is not in.
 4     then pagefault ← pagefault + 1
 5          if busy < m                                  ▷ The main memory is not full.
 6             then swap-in ← busy
 7                  busy ← busy + 1
 8          if busy = m                                  ▷ The physical memory is full.
 9             then swap-out ← r_{t-1}
10                  for i ← n − 1 downto 0
11                      do if pagetable[i, 1] = TRUE and
                              frequency[i] ≤ frequency[swap-out]
12                           then swap-out ← last-ref[i]
13                  pagetable[swap-out, 1] ← FALSE
14                  swap-in ← pagetable[swap-out, 2]
15                  KIÍR(swap-in, swap-out)
16          READ(r_t, pagetable[swap-out, 2])            ▷ Reading.
17          pagetable[r_t, 1] ← TRUE                     ▷ Updating.
18          pagetable[r_t, 2] ← swap-in
19          frequency[r_t] ← 1
```

Since the loop body in rows 11–13 of the LFU-EXECUTES algorithm has to be executed at most n-times, the running time of the algorithm is $O(np)$. There are certain operating systems in which there are two status bits belonging to the pages in the physical memory. The referenced bit is set to TRUE whenever a page is referenced (either for reading or writing), while the dirty bit is set to TRUE whenever modifying (i.e. writing) a page. When starting the program both of the status bits of each page is set to FALSE. At stated intervals (e. g. after every k-th instruction) the operating system sets the referenced bit of the pages which has not been referenced since the last setting to FALSE. Pages fall into four classes according to the values of their two status bits: class 0 contains the pages not referenced and not modified, class 1 the not referenced but modified, class 2 the referenced, but not modified, and finally, class 3 the referenced and modified ones.

The NRU (**N**ot **R**ecently **U**sed) algorithm chooses a page to be replaced from the nonempty class with the smallest index. So that the algorithm would be deterministic, we suppose that the NRU algorithm stores the elements of each class in a row.

The preparation of this algorithm means to fill arrays referenced and dirty containing the indicator bits with FALSE values, to zero the value of variable performed

showing the number of the operations performed since the last zeroing and to create four empty queues.

NRU-PREPARES(n)

```
1  for i ← 0 to n − 1
2      do referenced[j] ← FALSE
3         dirty[j] ← FALSE
4  Q_0 ← ∅
5  Q_1 ← ∅
6  Q_2 ← ∅
7  Q_3 ← ∅
```

NRU-EXECUTES($referenced, dirty, k, R, W$)

```
 1  if pagetable[r_t, 1] = TRUE                          ▷ The next page is in.
 2     then if W[r_t] = TRUE
 3             then dirty[r_t] ← TRUE
 4  if pagetable[r_t, 1] = FALSE                         ▷ The next page is not in.
 5     then pagefault ← pagefault + 1
 6          if busy < m                                  ▷ The main memory is not full.
 7             then swap-in ← busy
 8                  busy ← busy + 1
 9                  referenced[r_t] ← TRUE
10                  if W[r_t] = TRUE
11                     then dirty[r_t] ← TRUE
12          if busy = m                                  ▷ The main memory is full.
13             then NRU-SWAP-OUT(t, swap-out)
14                  pagetable[swap-out, 1] ← FALSE
15                  swap-in ← pagetable[swap-out, 2]
16                  if dirty[sap-out] = TRUE
17                     then WRITE(swap-in, swap-out)
18          READ(r_t, pagetable[swap-in, 2])             ▷ Reading.
19          pagetable[r_t, 1] ← TRUE                     ▷ Updating.
20          pagetable[r_t, 2] ← swap-in
21  if t/k = ⌊t/k⌋
22     then for i ← 0 to n − 1
23             do if referenced[i] = FALSE
24                   then dirty[i] ← FALSE
```

Choosing the page to be replaced is based on dividing the pages in the physical memory into four queues (Q_1, Q_2, Q_3, Q_4).

NRU-SWAP-OUT(*time*)

```
1  for i ← 0 to n − 1                                  ▷ Classifying the pages.
2      do if referenced[i] = FALSE
3            then if dirty[i] = FALSE
4                    then ENQUEUE(Q1,i)
5                    else ENQUEUE(Q2,i)
6            else  if dirty[i] = FALSE
7                    then ENQUEUE(Q3,i)
8                    else ENQUEUE(Q4,i)
9  if Q1 ≠ ∅                                           ▷ Choosing the page to be replaced.
10   then swap-out ← DEQUEUE(Q1)
11   else if Q2 ≠ ∅
12           then swap-out ← DEQUEUE(Q2)
13           else if Q3 ≠ ∅
14                   then swap-out ← DEQUEUE(Q3)
15                   else swap-out ← DEQUEUE(Q4)
16 return swap-out
```

The space requirement of the RUN-NRU algorithm is $O(m)$ and its running time is $O(np)$. The SECOND-CHANCE algorithm is a modification of FIFO. Its main point is that if the referenced bit of the page to be replaced is FALSE according to FIFO, then we swap it out. If, however, its referenced bit is TRUE, then we set it to FALSE and put the page from the beginning of the queue to the end of the queue. This is repeated until a page is found at the be-ginning of the queue, the referenced bit of which is FALSE. A more efficient implementation of this idea is the CLOCK algorithm which stores the in-dices of the m pages in a circular list, and uses a hand to point to the next page to be replaced.

The essence of the LIFO (**L**ast **I**n **F**irst **O**ut) algorithm is that after filling in the physical memory according to the requirements we always replace the last arrived page, i. e., after the initial period there are $m-1$ pages constantly in the memory—and all the replacements are performed in the page frame with the highest address.

17.2.2. Dynamic paging

It is typical of most of the computers that there are multiple programs running simultane-ously on them. If there is paged virtual memory on these computers, it can be managed both locally and globally. In the former case each program's demand is dealt with one by one, while in the latter case a program's demand can be satisfied even at other programs' expenses. Static page replacement algorithms using local management have been discussed in the last section. Now we present two dynamic algorithms. The WS (WORKING-SET) algorithm is based on the experience that when a program is run-ning, in relatively short time there are only few of its pages needed. These pages form the working set belonging to the given time interval. This working set can be defined for example as the set of the pages needed for the last h instructions. The operation of the algorithm can be illustrated as pushing a "window" with length of h along reference array R, and keeping the pages seen

through this window in the memory.

WS($pagetable, t, h$)

```
1  if pagetable[r_t, 1] = FALSE                    ▷ The next page is not in.
2     then WS-SWAP-OUT(t)
3          WRITE(pagetable[swap-out, 2], swap-out)
4          pagetable[r_t, 1] ← TRUE
5          pagetable[r_t, 2] ← swap-out
6  if t > h                                        ▷ Does r_{t-h} in the memory?
7     then j ← h − 1
8          while r_j ≠ r_{t-h} and j < t
9                do j ← j + 1
10         if j > t
11            then pagetable[r_{t-h}, 1] ← FALSE
```

When discussing the WS algorithm, to make it as simple as possible, we suppose that $h \leq n$,, therefore, storing the pages seen through the window in the memory is possible even if all the h references are different (in practice, h is usually significantly bigger than n due to the many repetitions in the reference string).

The WS-SWAP-OOUT algorithm can be a static page replacement algorithm, for instance, which chooses the page to be replaced from all the pages in the memory—i. e., globally. If, for example, the FIFO algorithm with running time $\Theta(p)$ is used for this purpose, then the running time of the WS algorithm will be $\Theta(hp)$, since in the worst case it has to examine the pages in the window belonging to every single instruction.

The PFF (**P**age **F**requency **F**ault) algorithm uses a parameter as well. This algorithm keeps record of the number of the instructions executed since the last page fault. If this number is smaller when the next page fault occurs than a previously determined value of parameter d, then the program will get a new page frame to be able to load the page causing page fault. If, however, the number of instructions executed without any page faults reaches value d, then first all the page frames containing pages that have not been used since the last page fault will be taken away from the program, and after that it will be given a page frame for storing the page causing page fault.

PFF($pagetable, t, d$)

```
1  counter ← 0                                     ▷ Preparation.
2  for i ← 1 to n
3       do pagetable[i, 1] ← FALSE
4          referenced[i] ← FALSE
```

```
5  for j ← 1 to p                                          ▷ Running.
6      do if pagetable[r_t, 1] = TRUE
7            then counter ← counter + 1
8            else PFF-Swap-In(t, d, swap-out)
9                 Write(pagetable[swap-out, 2], swap-out)
10                pagetable[r_t, 1] ← TRUE
11         for i ← to n
12             do if referenced[i] = FALSE
13                   then pagetable[i, 1] ← FALSE
14                        referenced[i] ← FALSE
```

Exercises

17.2-1 Consider the following reference string: $R = \langle 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6 \rangle$. How many page faults will occur when using FIFO, LRU or OPT algorithm on a computer with main memory containing k $(1 \leq k \leq 8)$ page frames?

17.2-2 Implement the FIFO algorithm using a pointer—instead of queue Q—pointing to the page frame of the main memory, which is the next one to load a page.

17.2-3 What would be the advantages and disadvantages of the page replacement algorithms' using an $m \times 2$ page map—besides the page table—the j-th row of which indicating whether the j-th row of the physical memory is reserved, and also reflecting its content?

17.2-4 Write and analyse the pseudo code pseudocode of Second-Chance, Clock and LIFO algorithms.

17.2-5 Is it possible to decrease the running time of the NFU algorithm (as far as its order of magnitude is concerned) if the pages are not classed only after each page faults, but the queues are maintained continuously?

17.2-6 Another version, NFU', of the NRU algorithm is also known, which uses four sets for classing the pages, and it chooses the page to be replaced from the nonempty set with the smallest index by chance. Write the pseudo code of operations In-Set and From-Set needed for this algorithm, and calculate the space requirement and running time of the NFU' algorithm.

17.2-7* Extend the definition of the page replacement automat so that it would stop after processing the last entry of the finite reference sequence. *Hint.* Complete the set of incoming signs with an 'end of the sequence' sign.

17.3. Anomalies

When the first page replacement algorithms were tested in the IBM Watson Research Institute at the beginning of the 1960's, it caused a great surprise that in certain cases increasing the size of the memory leads to an increase in running time of the programs. In computer systems the phenomenon, when using more recourses leads

to worse results is called anomaly. Let us give three concrete examples. The first one is in connection with the FIFO page replacement algorithm, the second one with the LIST-SCHEDULING algorithm used for processor scheduling, and the third one with parallel program execution in computers with interleaved memories.

Note that in two examples out of the three ones a very rare phenomenon can be observed, namely that the degree of the anomaly can be any large.

17.3.1. Page replacement

Let m, M, n and p be positive integers ($1 \leq m \leq n < \infty$), k a non-negative integer, $A = \{a_1,\ a_2, \ldots, a_n\}$ a finite alphabet. A^k is the set of the words over A with length k, and A^* the words over A with finite length. Let m be the number of page frames in the main memory of a small, and M a big computer. The FIFO algorithm has already been defined in the previous section. Since in this subsection only the FIFO page replacement algorithm is discussed, the sign of the page replacement algorithm can be omitted from the notations.

Let us denote the number of the page faults by $f_P(R, m)$. The event, when $M > m$ and $f_P(R, M) > f_P(R, m)$ is called ***anomaly***. In this case the quotient $f_P(R, M)/f_P(R, m)$ is the degree of the anomaly. The efficiency of algorithm P is measured by paging speed $E_P(R, m)$ which is defined as

$$E_P(R, m) = \frac{f_P(R, m)}{p}\ , \tag{17.3}$$

for a finite reference string $R = \langle r_1, r_2, \ldots\rangle$, while for an infinite reference string $R = \langle r_1, r_2, \ldots\rangle$ by

$$E_P(R, m) = \liminf_{k\to\infty} \frac{f_P(R_k, m)}{k}\ . \tag{17.4}$$

Let $1 \leq m < n$ and let $C = (1, 2, \ldots, n)^*$ be an infinite, circular reference sequence. In this case $E_{\mathrm{FIFO}}(C, m) = 1$.

If we process the reference string $R = \langle 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5\rangle$, then we will get 9 page faults in the case of $m = 3$, and 10 ones in the case of $m = 4$, therefore, $f_{\mathrm{FIFO}}(R, m) = 10/9$. Bélády, Nelson and Shedler has given the following necessary and sufficient condition for the existing of the anomaly.

Theorem 17.1 *There exists a reference sequence R for which the* FIFO *page replacement algorithm causes an anomaly if, and only if $m < M < 2m - 1$.*

The following has been proved as far as the degree of the anomaly is concerned.

Theorem 17.2 *If $m < M < 2m - 1$, then for every $\epsilon > 0$ there exists a reference sequence $R = \langle r_1, r_2, \ldots, r_p\rangle$ for which*

$$\frac{f(R, M)}{f(R, m)} > 2 - \epsilon\ . \tag{17.5}$$

Bélády, Nelson and Shedler had the following conjecture.

Conjecture 17.3 *For every reference sequence R and memory sizes $M > m \geq 1$*

$$\frac{f_{\mathrm{FIFO}}(R,M)}{f_{\mathrm{FIFO}}(R,m)} \leq 2\ . \tag{17.6}$$

This conjecture can be refuted e. g. by the following example. Let $m = 5$, $M = 6$, $n = 7$, $k \geq 1$, and $R = UV^k$, where $V = (1,2,3,4,5,6,7)^3$ and $U = \langle 1,2,3,4,5,6,7,1,2,4,5,6,7,3,1,2,4,5,7,3,6,2,1,4,7,3,6,2,5,7,3,6,2,5\rangle$. If execution sequence U is executed using a physical memory with $m = 5$ page frames, then there will be 29 page faults, and the processing results in controlling status (7,3,6,2,5). After that every execution of reference sequence V causes 7 new page faults and results in the same controlling status.

If the reference string U is executed using a main memory with $M = 6$ page frames, then we get control state $\langle 2,3,4,5,6,7\rangle$ and 14 page faults. After that every execution of reference sequence V causes 21 new page faults and results in the same control state.

Choosing $k = 7$ the degree of the anomaly will be $(14 + 7 \times 21)/(29 + 7 \times 7) = 161/78 > 2$. As we increment the value of k, the degree of the anomaly will go to three. Even more than that is true: according to the following theorem by Péter Fornai and Antal Iványi the degree of the anomaly can be any arbitrarily large.

Theorem 17.4 *For any large number L it is possible to give parameters m, M and R so that the following holds:*

$$\frac{f(R,M)}{f(R,m)} > L\ . \tag{17.7}$$

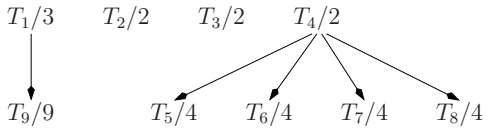
17.3.2. Scheduling with lists

Suppose that we would like to execute n ***tasks*** on p processors. By the execution the priority order of the programs has to be taken into consideration. The processors operate according to First Fit, and the execution is carried out according to a given list L. E. G. Coffman jr. wrote in 1976 that decreasing the number of processors, decreasing ***execution time*** t_i of the tasks, reducing the precedence restrictions, and altering the list can each cause an anomaly. Let the ***vector of the execution times*** of the tasks denoted by $\mathbf{t}$, the precedence relation by $<$, the list by L, and execution time of all the tasks with a common list on p equivalent processors by $C(p, L, <, \mathbf{t})$.

The degree of the anomaly is measured by the ratio of the execution time C' at the new parameters and execution time C at the original parameters. First let us show four examples for the different types of the anomaly.

Example 17.4 Consider the following task system τ_1 and its scheduling S_1 received using list $L = (T_1,\ T_2,\ \ldots,\ T_9)$ on $m = 3$ equivalent processors. In this case $C_{\max}(S_1) = 12$ (see Figure 17.1), which can be easily proved to be the optimal value.

Example 17.5 Schedule the previous task system τ_1 for $m = 3$ equivalent processors with list $L' = \langle T_1, T_2, T_4, T_5, T_6, T_3, T_9, T_7, T_8\rangle$. In this case for the resulting scheduling S_2 we



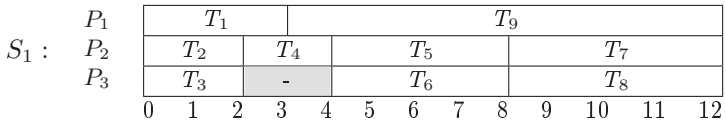


Figure 17.1. Task system τ_1,and its optimal schedule.

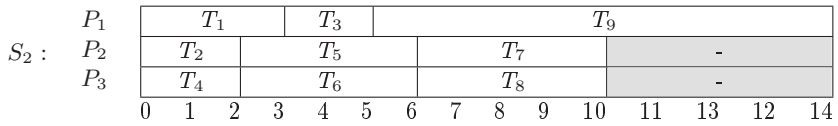


Figure 17.2. Scheduling of the task system τ_1 at list L'.

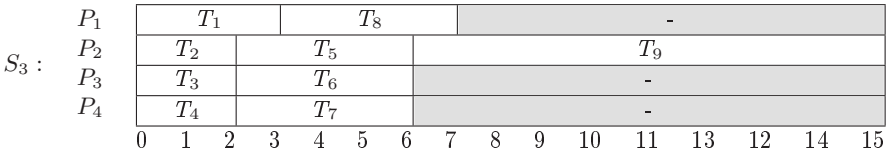


Figure 17.3. Scheduling of the task system τ_1 using list L on $m' = 4$ processors.

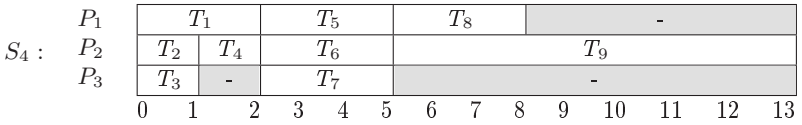


Figure 17.4. Scheduling of τ_2 with list L on $m = 3$ processors.

get $C_{max}(S_2) = 14$ (see Figure 17.2).

Example 17.6 Schedule the task system τ_1 with list L for $m' = 4$ processors. It results in $C_{max}(S_3) = 15$ (see Figure 17.3).

Example 17.7 Decrement the executing times by one in τ_1. Schedule the resulting task system τ_2 with list L for $m = 3$ processors. The result is: $C_{max}(S_4) = 13$ (see Figure 17.4).

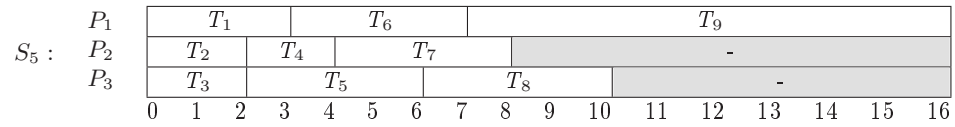


Figure 17.5. Scheduling task system τ_3 on $m = 3$ processors.

Example 17.8 Reduce the precedence restrictions: omit edges $(T_4,\ T_5)$ and $(T_4,\ T_6)$ from the graph. The result of scheduling S_5 of the resulting task system τ_3 can be seen in Figure 17.5: $C_{\max}(S_5) = 16$.

The following example shows that the increase of maximal finishing time in the worst case can be caused not only by a wrong choice of the list.

Example 17.9 Let task system τ and its optimal scheduling S_{OPT} be as showed by Figure 17.6. In this case $C_{max}(S_{\mathrm{OPT}}) = 19$.

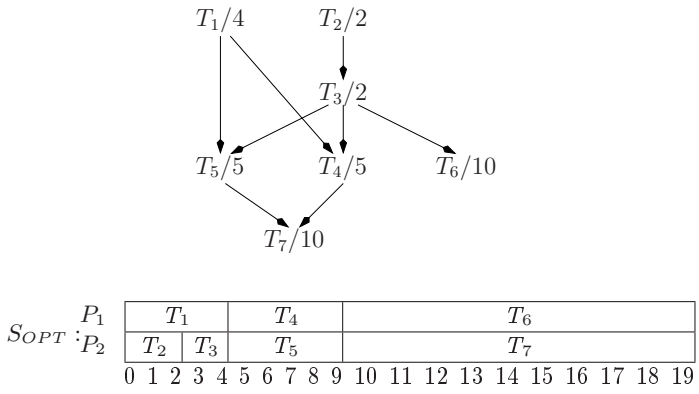


Figure 17.6. Task system τ and its optimal scheduling S_{OPT} on two processors.

We can easily prove that if the executing times are decremented by one, then in the resulting task system τ' we cannot reach a better result than $C_{max}(S_6) = 20$ with any lists (see Figure 17.7).

After these examples we give a relative limit reflecting the effects of the scheduling parameters. Suppose that for given task systems τ and τ' we have $T' = T$, $<' \subseteq <$, $\mathbf{t}' \leq \mathbf{t}$. Task system τ is scheduled with the help of list L, and τ' with L'—the former on m, while the latter on m' equivalent processors. For the resulting schedulings S and S' let $C(S) = C$ and $C(S') = C'$.

Theorem 17.5 (scheduling limit). *. With the above conditions*

$$\frac{C'}{C} \leq 1 + \frac{m-1}{m'} \ . \tag{17.8}$$

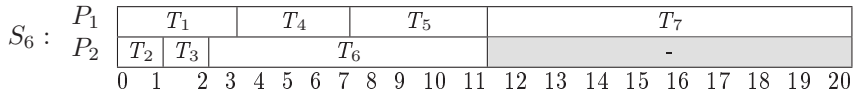


Figure 17.7. Optimal list scheduling of task system τ'.

Proof Consider scheduling diagram D' for the parameters with apostrophes (for S'). Let the definition of two subsets—A and B—of the interval $[0, C')$ be the following: $A = \{t \in [0, C') |$ all the processors are busy in time $t\}$, $B = [0, C') \setminus A$. Note that both sets are unions of disjoint, half-open (closed from the left and open from the right) intervals.

Let T_{j_1} be a task the execution of which ends in C' instant of time according to D1 (i. e., $f_{j_1} = C'$). In this case there are two possibilities: Starting point s_{j_1} of task T_{j_1} is either an inner point of B, or not.

1. If s_{j_1} is an inner point of B, then according to the definition of B there is a processor for which with $\varepsilon > 0$ it holds that it does not work in the interval $[s_{j_1} - \varepsilon, s_{j_1})$. This can only occur if there is a task T_{j_2} for which $T_{j_2} <' T_{j_1}$ and $f_{j_2} = s_{j_1}$ (case a).

2. If s_{j_1} is not an inner point of B, then either $s_{j_1} = 0$ (case b), or $s_{j_1} > 0$. If B has got a smaller element than s_{j_1} (case c), then let $x_1 = \sup\{x \mid x < s_{j_1} \text{ and } x \in B\}$, else let $x_1 = 0$ (case d). If $x_1 > 0$, then it follows from the construction of A and B that there is a processor for which a task T_{j_2} can be found the execution of which is still in progress in this time interval, and for which $T_{j_2} <' T_{j_1}$.

Summarising the two cases we can say that either there is a task $T_{j_2} <' T_{j_1}$ for which in the case of $y \in [f_{j_2}, s_{j_1})$ holds $y \in A$ (case a or c), or for every number $x < s_{j_1}$ $x \in A$ or $x < 0$ holds (case a or d).

Repeating this procedure we get a task chain $T_{j_r}, T_{j_{r-1}}, \ldots, T_{j_1}$ for which it holds that in the case of $x < s_{j_r}$ either $x \in A$ or $x < 0$. This proves that there are tasks for which

$$T_{j_r} <' T_{j_{r-1}} <' \cdots <' T_{j_1} \ , \tag{17.9}$$

and in every instant of time t there is a processor which is working, and is executing one of the elements of the chain. It yields

$$\sum_{\phi \in S'} t'(\phi) \leq (m' - 1) \sum_{k=1}^{r} t'_{j_k} \ , \tag{17.10}$$

where f denotes the empty periods, so the sum on the left hand side refers to all the empty periods in S'.

Based on (11.9) and $<' \subseteq <$, therefore,

$$C \geq \sum_{k=1}^{r} t_{j_k} \geq \sum_{k=1}^{r} t'_{j_k} \ . \tag{17.11}$$

S_7 :

T_1	T_{m+1}
T_2	T_{m+2}
$\vdots$	$\vdots$
T_{m-1}	T_{2m-1}
T_m	

Figure 17.8. Scheduling $S_7(\tau_4)$ belonging to list $L = (T_1, \ldots, T_n)$.

Since

$$mC \geq \sum_{i=1}^{n} t_i \geq \sum_{i=1}^{n} t'_i \ , \tag{17.12}$$

and

$$C' = \frac{1}{m'} \left(\sum_{i=1}^{n} t'_i + \sum_{\phi \in S'} t'(\phi) \right) ,$$

így (17.10), (17.11) and (17.12)

$$C' \leq \frac{1}{m'} \Big(mC + (m' - 1)C \Big) ,$$

based on (17.10), (17.11) and (17.12) we get

$$C' \leq \frac{1}{m'} \Big(mC + (m' - 1)C \Big) ,$$

implying $C'/C \leq 1 + (m-1)/m'$. ■

The following examples show us not only that the limit in the theory is the best possible, but also that we can get the given limit (at least asymptotically) by altering any of the parameters.

Example 17.10 In this example the list has changed, $<$ is empty, m is arbitrary. Execution times are the following:

$$t_i = \begin{cases} 1, & \text{if } i = 1, \ldots, m-1 \ , \\ m, & \text{if } i = m \ , \\ m-1, & \text{if } i = m+1, \ldots, 2m-1 \ . \end{cases}$$

If this task system τ_4 is scheduled for m processors with list $L = (T_1, \ldots, T_{2m-1}n)$, then we get the optimal scheduling $S_7(\tau_4)$ which can be seen in Figure 17.8.

If we use the list $L' = (T_{m+1}, \ldots, T_{2m-1}, T_1, \ldots, T_{m-1}, T_m)$ instead of list L, then we get scheduling $S_8(\tau_4)$ which can be seen in Figure 17.9.

In this case $C = (S_7) = m$, $C'(S_8) = 2m - 1$, therefore $C'/C = 2 - 1/m$; which means that altering the list results in the theorem holding with equality, i.e., the expression on the right hand side of the $\leq$ sign cannot be decreased.

Example 17.11 In this example we decrease the execution times. We use list $L = L' =$

S_8 :

T_{m+1}				T_m
T_{m+2}				-
$\vdots$				$\vdots$
T_{2m-1}				-
T_1	T_2	...	T_{m-1}	-

Figure 17.9. Scheduling $S_8(\tau_4)$ belonging to list L'.

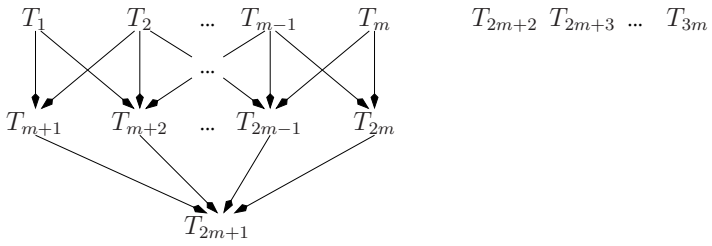


Figure 17.10. Identical graph of task systems τ_5 and τ_5'.

$\langle T_1, \ldots, T_{3m} \rangle$. in both cases. Here as well as in the remaining part of the chapter ε denotes an arbitrarily small positive number. Original execution times are stored in vector $\mathbf{t} = (t_1, \ldots, t_n)$, where

$$t_i = \begin{cases} 2\varepsilon, & \text{if } i = 1, \ldots, m \ , \\ 1, & \text{if } i = m+1, \ldots, 2m \ , \\ m-1, & \text{if } i = 2m+1, \ldots, 3m \ . \end{cases}$$

The new execution times are

$$t_i' = \begin{array}{ll} t_i - \varepsilon, & \textit{if } i = 1, \ldots, m-1 \ , \\ t_i, & \textit{if } i = m, \ldots, 3m \ . \end{array}$$

The precedence graph of task system τ_5, and its modification τ_5' are shown in Figure 17.10, while optimal scheduling $S_9(\tau_5)$ and scheduling $S_{10}(\tau_5')$ can be seen in Figure 17.11. Here $C = C_{\max}(S_9(\tau_5)) = m+2\varepsilon$ and $C' = C_{\max}(S_{10}(\tau_5') = 2m-1+\varepsilon$C = Cmax, therefore, increasing ε C'/C goes to value $2-1/m$ ($\lim_{\varepsilon \to 0} C'/C = 2-1/m$). This means that altering the execution times we can approach the limit in the theorem arbitrarily closely.

Example 17.12 In this example we reduce the precedence restrictions. The precedence graph of task system τ_6 is shown in Figure 17.12.

The execution times of the tasks are: $t_1 = \varepsilon$, $t_i = 1$, if $i = 1, \ldots, m^2 - m + 1$, and $t_{m^2-m+2} = m$. The optimal scheduling $S_{11}(\tau_6)$ of τ_6 belonging to list $L = (T_1, \ldots, T_{m^2-m+2})$ can be seen in Figure 17.13.

Omitting all the precedence restrictions from τ_6 we get the task system τ_6'. Scheduling $S_{12}(\tau_6')$ is shown in Figure 17.14.

Example 17.13 This time the number of the processors will be increased from m to m'.

S_9 :

T_1	T_{m+1}	T_{2m+1}
T_2	T_{m+2}	T_{2m+2}
$\vdots$	$\vdots$	$\vdots$
T_{m-1}	T_{2m-1}	T_{3m-1}
T_m	T_{2m}	T_{3m}

S_{10} :

T_1	T_{2m+2}		T_{2m-1}	T_{2m+1}
T_2	T_{2m+3}		-	
$\vdots$	$\vdots$		$\vdots$	
T_{m-1}	T_{3m}		-	
T_m	T_{m+1}	$\cdots$	T_{2m-1}	-

Figure 17.11. Schedulings $S_9(\tau_5)$ and $S_{10}(\tau_5')$.

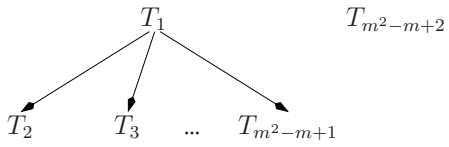


Figure 17.12. Graph of the task system τ_6.

S_{11} :

T_1	T_2	T_{m+1}	...	T_{m^2-2m+2}
T_{m^2-m+2}				-
-	T_3	T_{m+2}	...	T_{m^2-2m+3}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
-	T_m	T_{2m-1}	...	T_{m^2-m+1}

Figure 17.13. Optimal scheduling $S_{11}(\tau_6)$.

S_{12} :

T_1	T_{m+1}	$\cdots$	T_{m^2-m+1} -
T_2	T_{m+2}	$\cdots$	T_{m^2-m+2}
T_3	T_{m+3}	$\cdots$	-
$\vdots$	$\vdots$	$\ddots$	$\vdots$
T_{m-1}	T_{2m+1}	$\cdots$	-
T_m	T_{2m}	$\cdots$	-

Figure 17.14. Scheduling $S_{12}(\tau_6')$.

The graph of task system τ_7 is shown by Figure 17.15, and the running times are

$$t_i = \begin{cases} \varepsilon, & \text{if } i = 1, \ldots, m+1 \ , \\ 1, & \text{if } i = m+2, \ldots, mm' - m' + m + 1 \ , \\ m', & \text{if } i = mm' - m' + m + 2 \ . \end{cases}$$

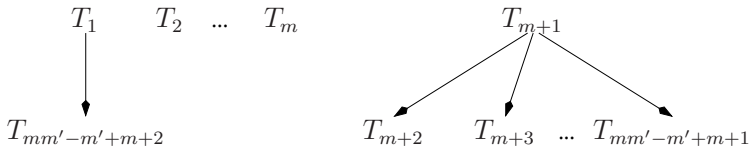


Figure 17.15. Precedence graph of task system τ_7.

S_{13} :

T_1	T_{m+1}	T_{m+2}	$\cdots$	T_a
T_2	$T_{mm'-m'+2}$			-
T_3	-	T_{m+3}	$\cdots$	T_b
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
T_m	-	T_{2m}	$\cdots$	T_c

Figure 17.16. The optimal scheduling $S_{13}(\tau_7)$ ($a = mm'-m'+3$, $b = a+1$, $c = mm'-m'+m+1$).

S_{14} :

T_1	T_{m+2}	$\cdots$	T_a	$T_{mm'-m'+m+2}$
T_2	T_{m+3}	$\cdots$	T_{a+1}	-
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
T_b	T_c	$\cdots$	T_d	-
-	$\vdots$	$\ddots$	$\vdots$	$\vdots$
-	T_e	$\cdots$	T_f	-

Figure 17.17. The optimal scheduling $S_{14}(\tau_7)$ ($a = mm' - 2m' + m + 2$, $b = m + 1$, $c = 2m + 2$, $d = mm' - 2m' + 2m + 2$, $e = m + m' + 1$, $f = mm' - m' + m + 1$).

The optimal scheduling of the task system on m, and m' processors is shown by Figure 17.16 and Figure 17.17.

Comparing the $C = C_{\max}(S_{13}(\tau_7)) = m'+2\varepsilon$, and $C' = C_{\max}(S_{14}(\tau_7)) = m'+m-1+\varepsilon$ maximal finishing times we get the ratio $C'/C = 1+(m-1-\varepsilon)(m'+2\varepsilon)$ and so again the required asymptotic value: $\lim_{\varepsilon\to 0} C'/C = 1+(m-1)/m'$

With the help of these examples we proved the following statement.

Theorem 17.6 (sharpness of the scheduling limit). *The limit given for the relative speed (11.8) is asymptotically sharp for the changing of (any of the) parameters m, t, $<$ and L.*

17.3.3. Parallel processing with interleaved memory

We describe the parallel algorithm modelling the operating of computers with interleaved memory in a popular way. The sequence of dumplings is modelling the

reference string, the giants the processors and the bites the commands executed simultaneously. Dwarfs $D_0,\ D_1,\ \ldots,\ D_r$ $(r \geq 0)$ cook dumplings of n different types. Every dwarf creates an infinite sequence of dumplings.

These sequences are usually given as random variables with possible values 1, 2, $\ldots, n$. For the following analysis of the extreme cases deterministic sequences are used.

The dumplings eating giants G_b $(b = 1, 2, \ldots)$ eat the dumplings. The units of the eating are the bits.

The appetite of the different giants is characterised by the parameter b. Giant $]_b$ is able to eat up most b dumplings of the same sort at one bite.

Giant G_b eats the following way. He chooses for his first bite from the beginning from the beginning of the dumpling sequence of dwarf T_0 so many dumplings, as possible (at most b of the same sort), and he adds to these dumplings so many ones from the beginning of the sequences of the dwarfs $D1,\ D_2,\ \ldots$, as possible.

After assembling the first bite the giant eats it, then he assembles and eats the second, third, ... bites.

Example 17.14 To illustrate the model let us consider an example. We have two dwarfs (D_0 and D_1) and the giant G_2. The dumpling sequences are

$$\begin{array}{l} 12121233321321321 \\ 24444444, \end{array} \tag{17.13}$$

or in a shorter form

$$\begin{array}{l} (12)^(3)^2(321)^* \\ 2(4)^*, \end{array} \tag{17.14}$$

where the star (*) denotes a subsequence repeated infinitely many times.

For his first bite G_2 chooses from the first sequence the first four dumlings 1212 (because the fifth dumpling is the third one of the sort 1) and no dumpling from the second sequence (because the beginning element is 2, and two dumplings of this sort is chosen already). The second bite contains the subsequence 1233 from the first sequence, and the dumplings 244 from the second one. The other bites are identical: 321321 from the first sequence and 44 from the second one. In a short form the bites are as follows:

$$\begin{array}{llll} \|1212 & \|1233 & \|321321 & \|^* \\ \|-- & \|244 & \|44 & \| \end{array} \tag{17.15}$$

(bites are separated by double lines).

For given dumpling sequences and a given giant G_b let B_t $(t = 1, 2, \ldots)$ denote the number of dumplings in the t-th bite. According to the eating-rules $b \leq B_t \leq bn$ holds for every t.

Considering the elements of the dumpling sequences as random variables with possible values $1, 2, \ldots, n$ and given distribution we define the dumpling-eating speed S_b (concerning the given sequences) of G_b as the average number of dumplings in one bite for a long time, more precisely

$$S_b = \liminf_{t \to \infty} \mathbf{E}\left(\frac{\sum_{i=1}^{t} B_i}{t}\right), \tag{17.16}$$

where $\mathbf{E}(\xi)$ denotes the expected value of the random variable ξ.

One can see that the defined limit always exists.

Maximal and minimal speed-ratio

Let us consider the case, when we have at least one dumpling sequence, at least one type of dumplings, and two different giants, that is let $r \geq 0,\ n \geq 1,\ b > c \geq 1$. Let the sequences be deterministic.

Since for every bite-size B_t $(t = 1, 2, \ldots)$ of G_b holds $b \leq B_t \leq bn$, the same bounds are right for every average value $(\sum B_i)_{i=1}^t)/t$ and for every expected value $E((\sum_{i=1}^t B_i)/t)$, too. From this it follows, that the limits S_b and S_c defined in (17.16) also must lie between these bounds, that is

$$b \leq S_b \leq bn, \quad g \leq S_c \leq cn\ . \tag{17.17}$$

Choosing the maximal value of S_b and the minimal value of S_c and vice versa we get the following trivial upper and lower bounds for the speed ratio S_b/S_c:

$$\frac{b}{cn} \leq \frac{S_b}{S_c} \leq \frac{bn}{c}\ . \tag{17.18}$$

Now we show that in many cases these trivial bounds cannot be improved (and so the dumpling eating speed of a small giant can be any times bigger than that of a big giant).

Theorem 17.7 *If $r \geq 1,\ n \geq 3,\ b > c \geq 1$, then there exist dumpling sequences, for which*

$$\frac{S_b}{S_c} = \frac{b}{cn}\ , \tag{17.19}$$

further

$$\frac{S_b}{S_c} = \frac{bn}{c}\ . \tag{17.20}$$

Proof To see the sharpness of the lower limit in the inequality (17.18) giving the natural limits let consider the following sequences:

$$\begin{array}{l} 1^b 2^{2b+1} 1^* \\ 1^{b+1}(23\ldots n)^*\ . \end{array} \tag{17.21}$$

Giant G_b eats these sequences in the following manner:

$$\begin{array}{llllll} \|1^b 2^b & \|2^b & \|21^b & \|1^b & \|^* \\ \|-- & \|1^b & \|-- & \|-- & \|. \end{array} \tag{17.22}$$

Here $B_1 = 2b$, $B_2 = 2b$, $B_3 = b + 1$, $B_t = b$ (for $t = 4, 5, \ldots$.

For the given sequences we have

$$S_b = \lim_{t\to\infty} \frac{2b + 1 + tb}{t} = b. \tag{17.23}$$

G_c eats these sequences as follows:

$$\begin{array}{llllllll} \|1^c & \|^\alpha\ 1^{c_1} 2^c & \|2^c & \|^\beta\ 2^c & \|2^c & \|^\gamma\ 2^{c_4} 1^c & \|1^c & \|^* \\ \|-- & \|1^{c_2} & \|1^c & \|1^{c_3} & \|-- & \|(23\ldots)^{c_3} & \|(23\ldots)^c & \|. \end{array} \tag{17.24}$$

Here

$$\alpha = \left\lceil \frac{b-c}{c} \right\rceil ;\ c_1 = b - \alpha c;\ c_2 = c - c_1 ;$$
$$\beta = \left\lceil \frac{b+1-c_2-c}{c} \right\rceil ;\ c_3 = b+1-c_2-\beta c ;$$
$$\gamma = \left\lceil \frac{2b+1-c(\beta+2)}{c} \right\rceil ;\ c_4 = 2b+1-c(\beta+\gamma+2) ;$$
$$c_5 = c - c_4.$$

In this case we get (in a similar way, as we have got S_b)

$$S_c = cn \tag{17.25}$$

and therefore

$$\frac{S_b}{S_c} = \frac{b}{cn} . \tag{17.26}$$

In order to derive the exact upper bound, we consider the following sequence:

$$\begin{array}{l} 12^{2b+1}1^* \\ 1^{b-1}3^{2b}1^b(23\ldots n)^* . \end{array} \tag{17.27}$$

G_b eats these sequences as follows:

$$\begin{array}{lllll} \|12^b & \|2^b & \|21^b & \|1^b & \|^* \\ \|1^{b-1}3^b & \|3^b1^b & \|(23\ldots)^{b-1} & \|(23\ldots n)^b & \| . \end{array} \tag{17.28}$$

From here we get

$$S_b = \lim_{t\to\infty} \frac{3b+3b+n(b-1)+2+(t-3)bn}{t} = bn . \tag{17.29}$$

G_c'c eating is characterised by

$$\begin{array}{llllllllll} \|12^c & \|2^c & \|^\alpha 2^c & \|2^c & \|^\beta\ 2^{c_3}1^c & \|1^c & \|1^c & \|1^c & \|^* \\ \|1^{c_1} & \|1^c & \|1^{c_2}3^c & \|3^c & \|3^c & \|3^c & \|3^{c_4} & \|-- & \| , \end{array} \tag{17.30}$$

where

$$c_1 = c-1;\ \alpha = \left\lceil \frac{b-c-c_1}{c} \right\rceil ;\ c_2 = b-1-c_1-\alpha c ;$$
$$\beta = \left\lceil \frac{2b+1-c(\alpha+\beta)}{c} \right\rceil ; \gamma = \left\lceil \frac{2b-c(\beta+2)}{c} \right\rceil \ ;$$
$$c_4 = 2b - c(\beta+\gamma+2) c_3 = 2b+1-c(\alpha+\beta+2) .$$

Since $B_t = c$ for $t = \alpha+\beta+\gamma+5,\ t = \alpha+\beta+\gamma+6, \ldots$, therefore $S_c = c$, and so $S_b/S_c = bn/c$. ■

$$\alpha = \left\lceil \frac{b-c}{c} \right\rceil ;\ c_1 = b - \alpha c;\ c_2 = c - c_1 ;$$
$$\beta = \left\lceil \frac{b+1-c_2-c}{c} \right\rceil ;\ c_3 = b+1-c_2-\beta c ;$$
$$\gamma = \left\lceil \frac{2b+1-c(\beta+2)}{c} \right\rceil ;\ c_4 = 2b+1-c(\beta+\gamma+2) ;$$
$$c_5 = c - c_4.$$

17.3.4. Avoiding the anomaly

We usually try to avoid anomalies.

For example at page replacing the sufficient condition of avoiding it is that the replacing algorithm should have the stack property: if the same reference string is run on computers with memory sizes of m and $m+1$, then after every reference it holds that the bigger memory contains all the pages that the smaller does. At the examined scheduling problem it is enough not to require the scheduling algorithm's using a list.

Exercises

17.3-1 Give parameters m, M, n, p and R so that the FIFO algorithm would cause at least three more page faults with a main memory of size M than with that of size m.

17.3-2 Give such parameters that using scheduling with list when increasing the number of processors the maximal stopping time increases at least to half as much again.

17.3-3 Give parameters with which the dumpling eating speed of a small giant is twice as big as that of a big giant.

17.4. Optimal file packing

In this section we will discuss a memory managing problem in which files with given sizes have to be placed onto discs with given sizes. The aim is to minimise the number of the discs used. The problem is the same as the bin-packing problem that can be found among the problems in Section *Approximation algorithms* in the book titled *Introduction to Algorithms*. Also scheduling theory uses this model in connection with minimising the number of processors. There is the number n of the files given, and array vector $\mathbf{t} = (t_1, t_2, \ldots, t_n)$ containing the sizes of the files to be stored, for the elements of which $0 < t_i \leq 1$ holds $(i = 1, 2, \ldots, n)$. The files have to be placed onto the discs taking into consideration that they cannot be divided and the capacity of the discs is a unit.

17.4.1. Approximation algorithms

The given problem is NP-complete. Therefore, different approaching algorithms are used in practice. The input data of these algorithms are: the number n of files, a vector $\mathbf{t} = \langle t_1, t_2, \ldots, t_n \rangle$ with the sizes of the files to be placed. And the output data are the number of discs needed (discnumber) and the level array $h = (h_1, h_2, \ldots, h_n)$ of discs.

Linear Fit (LF)

According to **L**inear **F**it file F_i is placed to disc D_i. The pseudocode of LF is the following.

LF$(n, \mathbf{t})$

```
1  for i ← 1 to n
2      do h[i] ← t[i]
3  number-of-discs ← n
4  return number-of-discs
```

Both the running time and the place requirement of this algorithm are $O(n)$. If, however, reading the sizes and printing the levels are carried out in the loop in rows 2–3, then the space requirement can be decreased to $O(1)$.

Next Fit (NF)
Next **F**it packs the files onto the disc next in line as long as possible. Its pseudocode is the following.

NF$(n, \mathbf{t})$

```
1  h[1] ← t[1]
2  number-of-discs ← 1
3  for i ← 2 to n
4      do if h[number-of-discs] + t[i] ≤ 1
5             then h[number-of-discs] ← h[number-of-discs] + t[i]
6             else number-of-discs ← number-of-discs + 1
7                  h[number-of-discs] ← t[i]
8  return number-of-discs
```

Both the running time and the place requirement of this algorithm are $O(n)$. If, however, reading the sizes and taking the levels out are carried out in the loop in rows 3–6, then the space requirement can be decreased to $O(1)$, but the running time is still $O(n)$.

First Fit (FF)
First **F**it packs each files onto the first disc onto which it fits.

FF$(n, \mathbf{t})$

```
 1  number-of-discs ← 1
 2  for i ← 1 to n
 3      do h[i] ← 0
 3  for i ← 1 to n
 4      do k ← 1
 5          while t[i] + h[k] > 1
 6                do k ← k + 1
 7          h[k] ← h[k] + t[i]
 8          if k > number-of-discs
 9             then number-of-discs ← number-of-discs + 1
10  return number-of-discs
```

The space requirement of this algorithm is $O(n)$, while its time requirement is $O(n^2)$. If, for example, every file size is 1, then the running time of the algorithm is $\Theta(n^2)$.

Best Fit (BF)
Best **F**it places each file onto the first disc on which the remaining capacity is the smallest.

BF$(n, \mathbf{t})$

```
 1  number-of-discs ← 1
 2  for i ← 1 to n
 3       do h[i] ← 0
 4  for i ← 1 to n
 5       do free ← 1.0
 6            ind ← 0
 7            for k ← 1 to number-of-discs
 8                 doif h[k] + t[i] ≤ 1 and 1 − h[k] − t[i] < free
 9                    then ind ← k
10                         szabad ← 1 − h[k] − t[i]
11       if ind > 0
12          then h[ind] ← h[ind] + t[i]
13          else  number-of-discs ← number-of-discs + 1
14                h[number-of-discs] ← t[i]
15  return number-of-discs
```

The space requirement of this algorithm is $O(n)$, while its time requirement is $O(n2)$.

Pairwise Fit (PF)
Pairwise Fit creates a pair of the first and the last element of the array of sizes, and places the two files onto either one or two discs—according to the sum of the two sizes. In the pseudocode there are two auxiliary variables: bind is the index of the first element of the current pair, and eind is the index of the second element of the current pair.

PF$(n, \mathbf{t})$

```
 1  number-of-discs ← 0
 2  beg-ind ← 1
 3  end-ind ← n
 4  while end-ind ≥ beg-ind
 5        do if end-ind − beg-ind ≥ 1
 6              then if t[beg-ind] + t[end-ind] > 1
 7                        then number-of-discs ← number-of-discs + 2
 8                             h[number-of-discs − 1] ← t[bind]
 9                             h[number-of-discs] ← t[eind]
10                        else number-of-discs ← number-of-discs + 1
11                             h[number-of-discs] ← t[beg-ind] + t[eind]
12             if end-ind = beg-ind
13              then number-of-discs ← number-of-discs + 1
14                   h[number-of-discs] ← t[end-ind]
15             beg-ind ← beg-ind + 1
16             end-ind ← end-ind − 1
17  return number-of-discs
```

The space requirement of this algorithm is $O(n)$, while its time requirement is $O(n^2)$. If, however, reading the sizes and taking the levels of the discs out are carried out online, then the space requirement will only be $O(1)$.

Next Fit Decreasing (NFD)
The following five algorithms consist of two parts: first they put the tasks into decreasing order according to their executing time, and then they schedule the ordered tasks. **N**ext **F**it **D**ecreasing operates according to NF after ordering. Therefore, both its space and time requirement are made up of that of the applied ordering algorithm and NF.

First Fit Decreasing (FFD)
First Fit Decreasing operates according to First Fit (FF) after ordering, therefore its space requirement is $O(n)$ and its time requirement is $O(n^2)$.

Best Fit Decreasing (BFD)
Best **F**it **D**ecreasing operates according to Best Fit (BF) after ordering, therefore its space requirement is $O(n)$ and its time requirement is $O(n2)$.

Pairwise Fit Decreasing (PFD)
Pairwise **F**it **D**ecreasing creates pairs of the first and the last tasks one after another, and schedules them possibly onto the same processor (if the sum of their executing time is not bigger than one). If it is not possible, then it schedules the given pair onto two processors.

Quick Fit Decreasing (QFD)
Quick **F**it **D**ecreasing places the first file after ordering onto the next empty disc,

and then adds the biggest possible files (found from the end of the ordered array of sizes) to this file as long as possible. The auxiliary variables used in the pseudocode are: bind is the index of the first file to be examined, and eind is the index of the last file to be examined.

QFD$(n, \mathbf{s})$

```
 1  beg-ind ← 1
 2  end-ind ← n
 4  number-of-discs ← 0
 5  while end-ind ≥ beg-ind
 6        do number-of-discs ← number-of-discs + 1
 7           h[number-of-discs] ← s[bind]
 8           beg-ind ← beg-ind + 1
 9           while end-ind ≥ beg-ind and h[number-of-discs] + s[eind] ≤ 1
10                 do ind ← end-ind
11                    while ind > beg-ind and h[number-of-discs] + s[ind − 1] ≤ 1
12                          do ind ← ind − 1
13  h[number-of-discs] ← h[number-of-discs] + s[ind]
14  if end-ind > ind
15    then for i ← ind to end-ind − 1
16               do s[i] ← s[i + 1]
17  end-ind ← end-ind − 1
18  return number-of-discs
```

The space requirement of this program is $O(n)$, and its running time in worst case is $\Theta(n_2)$, but in practice—in case of executing times of uniform distribution—it is $(n \lg n)$.

17.4.2. Optimal algorithms

Simple Power (SP)
This algorithm places each file—independently of each other—on each of the n discs, so it produces n^n placing, from which it chooses an optimal one. Since this algorithm produces all the different packing (supposing that two placing are the same if they allocate the same files to all of the discs), it certainly finds one of the optimal placing.

Factorial Algorithm (FACT)
This algorithm produces the permutations of all the files (the number of which is $n!$), and then it places the resulted lists using NF.

The algorithm being optimal can be proved as follows. Consider any file system and its optimal packing is $S_{\mathrm{OPT}}(t)$. Produce a permutation P of the files based on $S_{\mathrm{OPT}}(t)$ so that we list the files placed onto $P_1, P_2, \ldots, P_{\mathrm{OPT}}(t)$ respectively. If permutation P is placed by NF algorithm, then we get either S_{OPT} or another optimal placing (certain tasks might be placed onto processors with smaller indices).

Quick Power (QP)
This algorithm tries to decrease the time requirement of SP by placing 'large' files (the size of which is bigger than 0.5) on separate discs, and tries to place only the others (the 'small' ones) onto all the n discs. Therefore, it produces only n^K placing instead of n^n, where K is the number of small files.

Economic Power (EP)
This algorithm also takes into consideration that two small files always fit onto a disc—besides the fact that two large ones do not fit. Therefore, denoting the number of large files by N and that of the small ones by K it needs at most $N + (K+1)/2$ discs. So first we schedule the large discs to separate discs, and then the small ones to each of the discs of the number mentioned above. If, for instance, $N = K = n/2$, then according to this we only have to produce $(0.75n)^{0.5n}$.

17.4.3. Shortening of lists (SL)

With certain conditions it holds that list $\mathbf{t}$ can be split into lists $\mathbf{t}_1$ and $\mathbf{t}_2$ so that $\mathrm{OPT}(\mathbf{t}_1) + \mathrm{OPT}(\mathbf{t}_2) \leq \mathrm{OPT}(\mathbf{t})$ (in these cases the formula holds with equality). Its advantage is that usually shorter lists can be packed optimally in a shorter time than the original list. For example, let us assume that $t_i + t_j = 1$. Let $\mathbf{t}_1 = (t_i, t_j)$ and $\mathbf{t}_2 = \mathbf{t} \setminus \mathbf{t}_1$. In this case $\mathrm{OPT}(\mathbf{t}_1) = 1$ and $\mathrm{OPT}(\mathbf{t}_2) = \mathrm{OPT}(\mathbf{t}) - 1$. To prove this, consider the two discs onto which the elements of list $\mathbf{t}_1$ have been packed by an optimal algorithm. Since next to them there can be files whose sum is at most $1 - t_1$ and $1 - t_2$, their executing time can sum up to at most $2 - (t_1 + t_2)$, i.e., 1. Examining the lists on both ends at the same time we can sort out the pairs of files the sum of whose running time is 1 in $O(n)$. After that we order the list $\mathbf{t}$. Let the ordered list be $\mathbf{s}$. If, for example $s_1 + s_n < 1$, then the first file will be packed onto a different disc by every placing, so $\mathbf{t}_1 = (t_1, t_j)$ and $\mathbf{t}_2 = \mathbf{t} \setminus \mathbf{t}_1$ is a good choice. If for the ordered list $s_1 + s_n < 1$ and $s_1 + s_{n-1} + s_n > 1$ hold, then let s_j be the largest element of the list that can be added to s_1 without exceeding one. In this case with choices $\mathbf{t}_1 = (t_1, t_j)$ and $\mathbf{t}_2 = \mathbf{t} \setminus \mathbf{t}_1$ list $\mathbf{t_2}$ is two elements shorter than list $\mathbf{t}$. With the help of the last two operations lists can often be shortened considerably (in favourable case they can be shortened to such an extent that we can easily get the optimal number of processors for both lists). Naturally, the list remained after shortening has to be processed—for example with one of the previous algorithms.

17.4.4. Upper and lower estimations (ULE)

Algorithms based on upper and lower estimations operate as follows. Using one of the approaching algorithms they produce an upper estimation $\mathrm{A}(\mathbf{t})$ of $\mathrm{OPT}(\mathbf{t})$, and then they give a lower estimation for the value of $\mathrm{OPT}(\mathbf{t}$ as well. For this—among others—the properties of packing are suitable, according to which two large files cannot be placed onto the same disc, and the sum of the size cannot be more than 1 on any of the discs. Therefore, both the number of the large files and the sum of the size of the files, and so also their maximum $\mathrm{MAX}(\mathbf{t})$ is suitable as a lower estimation. If $\mathrm{A}(\mathbf{t} = \mathrm{MAX}(\mathbf{t})$, then algorithm A produced an optimal scheduling. Otherwise it

	LF	NF	FF	BF	PF	NFD	FFD	BFD	PFD	QFD	OPT
$\mathbf{t}_1$	4	3	3	3	3	3	2	2	2	2	2
$\mathbf{t}_2$	6	2	2	2	3	3	3	3	3	3	2
$\mathbf{t}_3$	7	3	2	3	4	3	2	3	4	2	2
$\mathbf{t}_4$	8	3	3	2	4	3	3	2	4	3	2
$\mathbf{t}_5$	5	3	3	3	3	2	2	2	3	2	2
$\mathbf{t}_6$	4	3	2	2	2	3	2	2	2	2	2
$\mathbf{t}_7$	4	3	3	3	2	3	2	2	2	2	2

Figure 17.18. Summary of the numbers of discs.

can be continued with one of the time-consuming optimum searching algorithms.

17.4.5. Pairwise comparison of the algorithms

If there are several algorithms known for a scheduling (or other) problem, then a simple way of comparing the algorithms is to examine whether the values of the parameters involved can be given so that the chosen output value is more favourable in the case of one algorithm than in the case of the other one.

In the case of the above discussed placing algorithm the number of processors discs allocated to size array $\mathbf{t}$ by algorithm A and B is denoted by A($\mathbf{t}$ and B($\mathbf{t}$, and we examine whether there are arrays $\mathbf{t}_1$ and $\mathbf{t}_2$ for which $\mathrm{A}(\mathbf{t}_1) < \mathrm{B}(\mathbf{t}_1)$ and $\mathrm{A}(\mathbf{t}_2) > \mathrm{B}(\mathbf{t}_2)$ hold. We answer this question in the case of the above defined ten approaching algorithms and for the optimal one. It follows from the definition of the optimal algorithms that for each $\mathbf{t}$ and each algorithm A holds $\mathrm{OPT}(\mathbf{t} \leq \mathrm{A}(\mathbf{t})$. In the following the elements of the arrays in the examples will be twentieth.

Consider the following seven lists:

$\mathbf{t_1} = (12/20, 6/20, 8/20, 14/20)$,
$\mathbf{t_2} = (8/20, 6/20, 6/20, 8/20, 6/20, 6/20)$,
$\mathbf{t_3} = (15/20, 8/20, 8/20, 3/20, 2/20, 2/20, 2/20)$,
$\mathbf{t_4} = (14/20, 8/20, 7/20, 3/20, 2/20, 2/20, 2/20, 2/20)$,
$\mathbf{t_5} = (10/20, 8/20, 10/20, 6/20, 6/20)$,
$\mathbf{t_6} = (12/20, 12/20, 8/20, 8/20)$,
$\mathbf{t_7} = (8/20, 8/20, 12/20, 12/20)$.

The packing results of these lists are summarised in Figure 17.18.

As shown in Figure 17.18, LF needs four discs for the first list, while the others need fewer than that. In addition, the row of list $\mathbf{t}_1$ shows that FFD, BFD, PFD, QFD and OPT need fewer discs than NF, FF, BF, PF and NFD. Of course, there are no lists for which any of the algorithms would use fewer discs than OPT. It is also obvious that there are no lists for which LF would use fewer discs than any of the other ten algorithms.

These facts are shown in Figure 17.19. In the figure symbols X in the main diagonal indicate that the algorithms are not compared to themselves. Dashes in the

	LF	NF	FF	BF	PF	NFD	FFD	BFD	PFD	QFD	OPT
LF	X	1	1	1	1	1	1	1	1	1	1
NF	–	X					1	1	1	1	1
FF	–		X				1	1	1	1	1
BF	–			X			1	1	1	1	1
PF	–				X		1	1	1	1	1
NFD	–					X	1	1	1	1	1
FFD	–						X				
BFD	–							X			
PFD	–								X		
QFD	–									X	
OPT	–	–	–	–	–	–	–	–	–	–	X

Figure 17.19. Pairwise comparison of algorithms.

	LF	NF	FF	BF	PF	NFD	FFD	BFD	PFD	QFD	OPT
LF	X	1	1	1	1	1	1	1	1	1	1
NF	–	X	3	4	7	5	1	1	1	1	1
FF	–	–	X	4	7	5	1	1	1	1	1
BF	–	–	3	X	8	5	1	1	1	1	1
PF	–	2	2	2	X	3	1	1	1	1	1
NFD	–	2	2	2	6	X	1	1	1	1	1
FFD	–	2	2	2		–	X	4		–	2
BFD	–	2	2	2		–	3	X		3	2
PFD	–	2	2	2	3	3	3	3	X	3	2
QFD	–	2	2	2		–	–	4		X	2
OPT	–	–	–	–	–	–	–	–	–	–	X

Figure 17.20. Results of the pairwise comparison of algorithms.

first column indicate that for the algorithm belonging to the given row there is no list which would be processed using more disc by this algorithm than by the algorithm belonging to the given column, i.e., LF. Dashes in the last column show that there is no list for which the optimal algorithm would use more discs than any of the examined algorithms. Finally, 1's indicate that for list $\mathbf{t}_1$ the algorithm belonging to the row of the given cell in the figure needs more discs than the algorithm belonging to the column of the given cell.

If we keep analysing the numbers of discs in Figure 17.19, we can make up this figure to Figure 17.20.

Since the first row and the first column of the table is filled, we do not deal more with algorithm LF.

For list $\mathbf{t}_2$ NF, FF, BF and OPT use two discs, while the other 6 algorithms use three ones. Therefore we write 2's in the points of intersection of the columns of the 'winners' and the rows of the 'losers' (but we do not rewrite the 1's given in the points of intersection of PF and OPT, and NFD and OPT, so we write 2's in $4 \times 6 - 2 = 22$ cells. Since both the row and the column of OPT have been filled in,

it is not dealt with any more in this section. The third list is disadvantageous for PF and PFD, therefore we write 3's in the empty cells in their rows. This list shows an example also for the fact that NF can be worse than FF, BF can be worse than FF, and BFD than FFD and QFD.

The fourth list can be processed only by BF and BFD optimally, i.e., using two discs. Therefore we can write 4's in the empty cells in the columns of these two algorithms. For the fifth list NFD, FFD, BFD and QFD use only two, while NF, FF, BF, PF and PDF use three discs. So we can fill the suitable cells with 5's. The 'losers' of list $\mathbf{t}_6$ are NF and NFD—therefore, we write 6's in the empty cells in their rows. PF performs better when processing list $\mathbf{t}_7$ than FF. The following theorem helps us filling in the rest of the cells.

Theorem 17.8 *If* $\mathbf{t} \in D$, *then*

$$\mathrm{FF}(\mathbf{t}) \leq \mathrm{NF}(\mathbf{t}) \ .$$

Proof We perform an induction according to the length of the list. Let $\mathbf{t} = \langle t_1, t_2, \ldots, t_n \rangle$ and $\mathbf{t}_i = \langle t_1, t_2, \ldots, t_i \rangle$ $(i = 1, 2, \ldots, n)$. Let $\mathrm{NF}(\mathbf{t}_i) = \mathrm{N}_i$ and $\mathrm{FF}(\mathbf{t}_i) = \mathrm{F}_i$, and let n_i be the level of the last disc according to NF, which means the sum of the lengths of the files placed onto the non empty disc with the higher index, when NF has just processed $\mathbf{t}_i$. Similarly, let f_i be the level of the last disc according to FF. We are going to prove the following invariant property for each i: either $F_i < N_i$, or $F_i = N_i$ and $f_i \leq n_i$. If $i = 1$, then $F_1 = N_1$ and $f_1 = n_1 = t_1$, i.e., the second part of the invariant property holds. Suppose that the property holds for the value $1 \leq i < n$. If the first part of the invariant property holds before packing t_{i+1}, then either inequality $F_i < N_i$ stays true, or the numbers of discs are equal, and $f_i < n_i$ holds. If the numbers of discs were equal before packing of t_{i+1}, then after placing it either the number of discs of FF is smaller, or the numbers of discs are equal and the level of the last disc of FF is at most as big as that of NF. ■

A similar statement can be proved for the pairs of algorithms NF-BF, NFD-FFD and NFD-BFD. Using an induction we could prove that FFD and QFD need the same number of discs for every list. The previous statements are summarised in Figure 11.20.

17.4.6. The error of approximate algorithms

The relative efficiency of two algorithms (A and B) is often described by the ratio of the values of the chosen efficiency measures, this time the relative number of processors $\mathrm{A}(\mathbf{t})/\mathrm{B}(\mathbf{t})$. Several different characteristics can be defined using this ratio. These can be divided into two groups: in the first group there are the quantities describing the worst case, while in the other group there are those describing the usual case. Only the worst case is going to be discussed here (the discussion of the usual case is generally much more difficult). Let D_n denote the real list of n elements and D the set of all the real lists, i.e.,

$$D = \cup_{i=1}^{\infty} D_i \ .$$

Let $\mathcal{A}_{nd}$ be the set of algorithms, determining the number of discs, that is of

algorithms, connecting a nonnegative real number to each list $\mathbf{t} \in D$, so implementing the mapping $D \to \mathbb{R}_0^+$).

Let $\mathcal{A}_{opt}$ be the set of the optimal algorithms, that is of algorithms ordering the optimal number of discs to each list, and OPT an element of this set (i.e., an algorithm that gives the number of discs sufficient and necessary to place the files belonging to the list for each list $\mathbf{t} \in D$).

Let $\mathcal{A}_{app}$ be the set of the approximation algorithms, that is of algorithms $\mathrm{A} \in \mathcal{A}_{nd}$ for which $\mathrm{A}(\mathbf{t}) \geq \mathrm{OPT}(\mathbf{t})$ for each list $\mathbf{t} \in D$, and there is a list $\mathbf{t} \in D$, for which $\mathrm{A}(\mathbf{t}) > \mathrm{OPT}(\mathbf{t})$..

Let A_{est} be the set of estimation algorithms, that is of algorithms $\mathrm{E} \in \mathcal{A}_{lsz}$ for which $\mathrm{E}(\mathbf{t}) \leq \mathrm{OPT}(\mathbf{t})$ for each list $\mathbf{t} \in D$, and there is a list $\mathbf{t} \in D$, for which $\mathrm{E}(\mathbf{t}) < \mathrm{OPT}(\mathbf{t})$.. Let F_n denote the set of real lists for which $\mathrm{OPT}(\mathbf{t}) = n$,, i.e., $F_n = \{\mathbf{t} | \mathbf{t} \in D \text{ and } \mathrm{OPT}(\mathbf{t}) = n\}$ $(n = 1, 2, \ldots)$.. In the following we discuss only algorithms contained in $\mathcal{A}_{nd}$. We define ($\mathrm{A}, \ \mathrm{B} \in \mathcal{A}$) $R_{\mathrm{A,B},n}$ error function, $R_{\mathrm{A,B}}$ error (absolute error) and $R_{A,\infty}$ asymptotic error of algorithms A and B ($\mathrm{A}, \ \mathrm{B} \in \mathcal{A}$) as follows:

$$R_{\mathrm{A,B},n} = \sup_{t \in F_n} \frac{\mathrm{A}(\mathbf{t})}{\mathrm{B}(\mathbf{t})} \ ,$$

$$R_{\mathrm{A,B}} = \sup_{t \in D} \frac{\mathrm{A}(\mathbf{t})}{\mathrm{B}(\mathbf{t})} \ ,$$

$$R_{\mathrm{A,B},\infty} = \limsup_{n \to \infty} R_{A,B,n} \ .$$

These quantities are interesting especially if $\mathrm{B} \in \mathcal{A}_{opt}$. In this case, to be as simple as possible, we omit B from the denotations, and speak about the error function, error and asymptotic error of algorithms $\mathrm{A} \in \mathcal{A}$, and $\mathrm{E} \in \mathcal{A}$. The characteristic values of NF file placing algorithm are known.

Theorem 17.9 *If* $\mathbf{t} \in F_n$, *then*

$$n = \mathrm{OPT}(\mathbf{t}) \leq \mathrm{NF}(\mathbf{t}) \leq 2\mathrm{OPT}(\mathbf{t}) - 1 = 2n - 1 \ . \tag{17.31}$$

Furthermore, if $k \in \mathbb{Z}$,, *then there are lists* $\boldsymbol{u}_k$ *and* $\boldsymbol{v}_k$ *for which*

$$k = \mathrm{OPT}(\mathbf{u}_k) = \mathrm{NF}(\mathbf{u}_k) \tag{17.32}$$

and

$$k = \mathrm{OPT}(\mathbf{v}_k) \ \textit{and} \ \mathrm{NF}(\mathbf{v}_k) = 2k - 1 \ . \tag{17.33}$$

From this statement follows the error function, absolute error and asymptotic error of NF placing algorithm.

Corollary 17.10 *If* $n \in \mathbb{Z}$, *then*

$$R_{\mathrm{NF},n} = 2 - \frac{1}{n} \ , \tag{17.34}$$

and

$$R_{\mathrm{NF}} = R_{\mathrm{NF},\infty} = 2 \ . \tag{17.35}$$

The following statement refers to the worst case of the FF and BF file packing algorithms.

Theorem 17.11 *If* $\mathbf{t} \in F_n$, *then*

$$\mathrm{OPT}(\mathbf{t}) \leq \mathrm{FF}(\mathbf{t}),\ \mathrm{BF}(\mathbf{t}) \leq 1.7\mathrm{OPT}(\mathbf{t}) + 2\ . \tag{17.36}$$

Furthermore, if $k \in \mathbb{Z},$, *then there are lists* u_k *and* v_k *for which*

$$k = \mathrm{OPT}(\mathbf{u}_k) = \mathrm{FF}(\mathbf{u}_k) = \mathrm{BF}(\mathbf{u}_k) \tag{17.37}$$

and

$$k = \mathrm{OPT}(\mathbf{v}_k)\ \textit{and}\ \mathrm{FF}(\mathbf{v}_k) = \mathrm{BF}(\mathbf{v}_k) = \lfloor 1.7k \rfloor\ . \tag{17.38}$$

For the algorithm FF holds the following stronger upper bound too.

Theorem 17.12 *If* $\mathbf{t} \in F_n$, *then*

$$\mathrm{OPT}(\mathbf{t}) \leq \mathrm{FF}(\mathbf{t}) < 1.7\mathrm{OPT}(\mathbf{t}) + 1\ . \tag{17.39}$$

From this statement follows the asymptotic error of FF and BF, and the good estimation of their error function.

Corollary 17.13 *If* $n \in \mathbb{Z}$, *then*

$$\frac{\lfloor 1.7n \rfloor}{n} \leq R_{\mathrm{FF},n} \leq \frac{\lceil 1.7n \rceil}{n} \tag{17.40}$$

and

$$\frac{\lfloor 1.7n \rfloor}{n} \leq R_{\mathrm{BF},n} \leq \frac{\lfloor 1.7n + 2 \rfloor}{n} \tag{17.41}$$

further

$$R_{\mathrm{FF},\infty} = R_{\mathrm{BF},\infty} = 1.7\ . \tag{17.42}$$

If n is divisible by 10, then the upper and lower limits in inequality (17.40) are equal, thus in this case $1.7 = \mathrm{R}_{\mathrm{FF},n} = R_{\mathrm{BF},n}$.

Exercises

17.4-1 Prove that the absolute error of the FF and BF algorithms is at least 1.7 by an example.

17.4-2 Implement the basic idea of the FF and BF algorithms so that the running time would be $O(n \lg n)$.

17.4-3 Complete Figure 11.20.

Problems

17-1 Smooth process selection for an empty partition

Modify the LONG-WAITING-OR-NOT-FIT-SMALLER algorithm in a way that instead of giving priority to processes with *points* above the *threshold*, selects the process with the highest *rank* + *points* among the processes fitting into the partition. Prove the correctness of the algorithm and give an upper bound for the waiting time of a process.

17-2 Partition search algorithms with restricted scope
Modify the Best-Fit, Limited-Best-Fit, Worst-Fit, Limited-Worst-Fit algorithms to only search for their optimal partitions among the next m suitable one following the last split partition, where m is a fixed positive number. Which algorithms do we get in the $m = 1$ and $m = \infty$ cases. Simulate both the original and the new algorithms, and compare their performance regarding execution time, average number of waiting processes and memory fragmentation.

17-3 Avoiding page replacement anomaly
Class the discussed page replacement algorithms based on whether they ensure to avoid the anomaly or not.

17-4 Optimal page replacement algorithm
Prove that for each demanding page replacement algorithm A, memory size m and reference string R holds

$$f_A(m, R) \leq f_{\mathrm{OPT}}(m, R) \ .$$

17-5 Anomaly
Plan (and implement) an algorithm with which it can occur that a given problem takes longer to solve on $q > p$ processors than on $p > 1$ ones.

17-6 Error of file placing algorithms
Give upper and lower limits for the error of the BF, BFD, FF and FFD algorithms.

Chapter notes

The basic algorithms for dynamic and fixed partitioning and page replacement are discussed according to textbooks by Silberschatz, Galvin and Gagne [302], and Tanenbaum and Woodhull [316].

Defining page replacement algorithms by a Mealy-automat is based on the summarising article by Denning [87], and textbooks by Ferenc Gécseg and István Peák [130], Hopcroft, Motwani and Ullman [166].

Optimizing the MIN algorithm was proved by Mihnovskiy and Shor in 1965 [243], after that by Mattson, Gecsei, Slutz and Traiger in 1970 [235].

The anomaly experienced in practice when using FIFO page replacement algorithm was first described by László Bélády [44] in 1966, after that he proved in a constructive way that the degree of the anomaly can approach two arbitrarily closely in his study he wrote together with Shedler. The conjecture that it cannot actually reach two can be found in the same article (written in 1969).

Péter Formai and Antal Iványi [115] showed that the ratio of the numbers of page replacements needed on a big and on a smaller computer can be arbitrarily large in 2002.

Examples for scheduling anomalies can be found in the books by Coffman [71], Iványi and Smelyanskiy [177] and Roosta [289], and in the article by Lai and Sahni [206].

Analysis of the interleaved memory derives from the article [173].

The bound $\mathrm{NF}(\mathbf{t}) \leq 2\mathrm{OPT}(\mathbf{t}) + 2$ can be found in D. S. Johnson's PhD dissertation [182], the precise Theorem 17.9. comes from [174]. The upper limit for FF and BF is a result by Johnson, Demers, Ullman, Garey and Graham [183], while the proof of the accuracy of the limit is that by [174, 175]. The source of the upper limit for FFD and BFD is [183], and that of the limit for NFD is [28]. The proof of the NP-completeness of the file packing problem—leading it back to the problem of partial sum—can be found in the chapter on approximation algorithms in *Introduction to Algorithms* [73].

18. Relational Database Design

The relational datamodel was introduced by Codd in 1970. It is the most widely used datamodel—extended with the possibilities of the World Wide Web—, because of its simplicity and flexibility. The main idea of the relational model is that data is organised in relational tables, where rows correspond to individual ***records*** and columns to ***attributes***. A ***relational schema*** consists of one or more relations and their attribute sets. In the present chapter only schemata consisting of one relation are considered for the sake of simplicity. In contrast to the mathematical concept of relations, in the relational schema the order of the attributes is not important, always ***sets*** of attributes are considered instead of ***lists***. Every attribute has an associated ***domain*** that is a set of elementary values that the attribute can take values from. As an example, consider the following schema.

Employee(Name,Mother's name,Social Security Number,Post,Salary)

The domain of attributes ***Name*** and ***Mother's name*** is the set of finite character strings (more precisely its subset containing all possible names). The domain of ***Social Security Number*** is the set of integers satisfying certain formal and parity check requirements. The attribute ***Post*** can take values from the set {Director,Section chief,System integrator,Programmer,Receptionist,Janitor,Handyman}. An ***instance*** of a schema R is a relation r if its columns correspond to the attributes of R and its rows contain values from the domains of attributes at the attributes' positions. A typical row of a relation of the Employee schema could be

(John Brown,Camille Parker,184-83-2010,Programmer,$172,000)

There can be dependencies between different data of a relation. For example, in an instance of the Employee schema the value of Social Security Number determines all other values of a row. Similarly, the pair (Name,Mother's name) is a unique identifier. Naturally, it may occur that some set of attributes do not determine all attributes of a record uniquely, just some of its subsets.

A relational schema has several ***integrity constraints*** attached. The most important kind of these is ***functional dependency***. Let U and V be two sets of attributes. V ***functionally depends*** on U, $U \to V$ in notation, means that whenever two records are identical in the attributes belonging to U, then they must agree

in the attribute belonging to V, as well. Throughout this chapter the attribute set $\{A_1, A_2, \ldots, A_k\}$ is denoted by $A_1A_2 \ldots A_k$ for the sake of convenience.

Example 18.1 *Functional dependencies* Consider the schema

$$\mathrm{R}(\mathbf{P}\text{rofessor}, \mathbf{Su}\text{bject}, \mathbf{R}\text{oom}, \mathbf{St}\text{udent}, \mathbf{G}\text{rade}, \mathbf{T}\text{ime}) \quad .$$

The meaning of an individual record is that a given student got a given grade of a given subject that was taught by a given professor at a given time slot. The following functional dependencies are satisfied.

Su→**P**: One subject is taught by one professor.
PT→**R**: A professor teaches in one room at a time.
StT→**R**: A student attends a lecture in one room at a time.
StT→**Su**: A student attends a lecture of one subject at a time.
SuSt→**G**: A student receives a unique final grade of a subject.

In Example 18.1 the attribute set **StT** uniquely determines the values of all other attributes, furthermore it is minimal such set with respect to containment. This kind attribute sets are called ***keys***. If all attributes are functionally dependent on a set of attributes X, then X is called a ***superkey***. It is clear that every superkey contains a key and that any set of attributes containing a superkey is also a superkey.

18.1. Functional dependencies

Some functional dependencies valid for a given relational schema are known already in the design phase, others are consequences of these. The **StT**→**P** dependency is implied by the **StT**→**Su** and **Su**→**P** dependencies in Example 18.1. Indeed, if two records agree on attributes **St** and **T**, then they must have the same value in attribute **Su**. Agreeing in **Su** and **Su**→**P** implies that the two records agree in **P**, as well, thus **StT**→**P** holds.

Definition 18.1 *Let R be a relational schema, F be a set of functional dependencies over R. The functional dependency $U \to V$ is* ***logically implied*** *by F, in notation $F \models U \to V$, if each instance of R that satisfies all dependencies of F also satisfies $U \to V$. The* ***closure*** *of a set F of functional dependencies is the set F^+ given by*

$$F^+ = \{U \to V : F \models U \to V\} \ .$$

18.1.1. Armstrong-axioms

In order to determine keys, or to understand logical implication between functional dependencies, it is necessary to know the closure F^+ of a set F of functional dependencies, or for a given $X \to Z$ dependency the question whether it belongs to F^+ must be decidable. For this, ***inference rules*** are needed that tell that from a set of functional dependencies what others follow. The ***Armstrong-axioms*** form a system of ***sound*** and ***complete*** inference rules. A system of rules is sound if only valid functional dependencies can be derived using it. It is complete, if every

dependency $X \to Z$ that is logically implied by the set F is derivable from F using the inference rules.

ARMSTRONG-AXIOMS

(A1) ***Reflexivity*** $Y \subseteq X \subseteq R$ implies $X \to Y$.

(A2) ***Augmentation*** If $X \to Y$, then for arbitrary $Z \subseteq R$, $XZ \to YZ$ holds.

(A3) ***Transitivity*** If $X \to Y$ and $Y \to Z$ hold, then $X \to Z$ holds, as well.

Example 18.2 *Derivation by the Armstrong-axioms* Let $R = ABCD$ and $F = \{A \to C, B \to D\}$, then AB is a key:

1. $A \to C$ is given.
2. $AB \to ABC$ 1. is augmented by (A2) with AB.
3. $B \to D$ is given.
4. $ABC \to ABCD$ 3. is augmented by (A2) with ABC.
5. $AB \to ABCD$ transitivity (A3) is applied to 2. and 4..

Thus it is shown that AB is superkey. That it is really a key, follows from algorithm CLOSURE(R, F, X).

There are other valid inference rules besides (A1)–(A3). The next lemma lists some, the proof is left to the Reader (Exercise 18.1-5).

Lemma 18.2

1. ***Union rule*** $\{X \to Y, X \to Z\} \models X \to YZ$.
2. ***Pseudo transitivity*** $\{X \to Y, WY \to Z\} \models XW \to YZ$.
3. ***Decomposition*** *If* $X \to Y$ *holds and* $Z \subseteq Y$*, then* $X \to Z$ *holds, as well.*

The soundness of system (A1)–(A3) can be proven by easy induction on the length of the derivation. The completeness will follow from the proof of correctness of algorithm CLOSURE(R, F, X) by the following lemma. Let X^+ denote the ***closure*** of the set of attributes $X \subseteq R$ with respect to the family of functional dependencies F, that is $X^+ = \{A \in R\colon X \to A$ follows from F by the Armstrong-axioms$\}$.

Lemma 18.3 *The functional dependency* $X \to Y$ *follows from the family of functional dependencies* F *by the Armstrong-axioms iff* $Y \subseteq X^+$.

Proof Let $Y = A_1 A_2 \dots A_n$ where A_i's are attributes, and assume that $Y \subseteq X^+$. $X \to A_i$ follows by the Armstrong-axioms for all i by the definition of X^+. Applying the union rule of Lemma 18.2 $X \to Y$ follows. On the other hand, assume that $X \to Y$ can be derived by the Armstrong-axioms. By the decomposition rule of Lemma 18.2 $X \to A_i$ follows by (A1)–(A3) for all i. Thus, $Y \subseteq X^+$. ■

18.1.2. Closures

Calculation of closures is important in testing equivalence or logical implication between systems of functional dependencies. The first idea could be that for a given

family F of functional dependencies in order to decide whether $F \models \{X \to Y\}$, it is enough to calculate F^+ and check whether $\{X \to Y\} \in F^+$ holds. However, the size of F^+ could be exponential in the size of input. Consider the family F of functional dependencies given by

$$F = \{A \to B_1, A \to B_2, \ldots, A \to B_n\} .$$

F^+ consists of all functional dependencies of the form $A \to Y$, where $Y \subseteq \{B_1, B_2, \ldots, B_n\}$, thus $|F^+| = 2^n$. Nevertheless, the closure X^+ of an attribute set X with respect to F can be determined in linear time of the total length of functional dependencies in F. The following is an algorithm that calculates the closure X^+ of an attribute set X with respect to F. The input consists of the schema R, that is a finite set of attributes, a set F of functional dependencies defined over R, and an attribute set $X \subseteq R$.

CLOSURE(R,F,X)

```
1  X^(0) ← X
2  i ← 0
3  G ← F                                  ▷ Functional dependencies not used yet.
4  repeat
5      X^(i+1) ← X^(i)
6      for all Y → Z in G
7          do if Y ⊆ X^(i)
8              then X^(i+1) ← X^(i+1) ∪ Z
9                   G ← G \ {Y → Z}
10     i ← i + 1
11 until X^(i−1) = X^(i)
```

It is easy to see that the attributes that are put into any of the $X^{(j)}$'s by CLOSURE(R,F,X) really belong to X^+. The harder part of the correctness proof of this algorithm is to show that each attribute belonging to X^+ will be put into some of the $X^{(j)}$'s.

Theorem 18.4 CLOSURE(R,F,X) *correctly calculates* X^+.

Proof First we prove by induction that if an attribute A is put into an $X^{(j)}$ during CLOSURE(R,F,X), then A really belongs to X^+.
Base case: $j = 0$. I this case $A \in X$ and by reflexivity (A1) $A \in X^+$.
Induction step: Let $j > 0$ and assume that $X^{(j-1)} \subseteq X^+$. A is put into $X^{(j)}$, because there is a functional dependency $Y \to Z$ in F, where $Y \subseteq X^{(j-1)}$ and $A \in Z$. By induction, $Y \subseteq X^+$ holds, which implies using Lemma 18.3 that $X \to Y$ holds, as well. By transitivity (A3) $X \to Y$ and $Y \to Z$ implies $X \to Z$. By reflexivity (A1) and $A \in Z$, $Z \to A$ holds. Applying transitivity again, $X \to A$ is obtained, that is $A \in X^+$.

On the other hand, we show that if $A \in X^+$, then A is contained in the result of CLOSURE(R,F,X). Suppose in contrary that $A \in X^+$, but $A \notin X^{(i)}$, where $X^{(i)}$ is the result of CLOSURE(R,F,X). By the stop condition in line 9 this means

$X^{(i)} = X^{(i+1)}$. An instance r of the schema R is constructed that satisfies every functional dependency of F, but $X \to A$ does not hold in r if $A \notin X^{(i)}$. Let r be the following two-rowed relation:

Attributes of $X^{(i)}$				Other attributes			
1	1	...	1	1	1	...	1
1	1	...	1	0	0	...	0

Let us suppose that the above r violates a $U \to V$ functional dependency of F, that is $U \subseteq X^{(i)}$, but V is not a subset of $X^{(i)}$. However, in this case CLOSURE(R,F,X) could not have stopped yet, since $X^{(i)} \neq X^{(i+1)}$.

$A \in X^+$ implies using Lemma 18.3 that $X \to A$ follows from F by the Armstrong-axioms. (A1)–(A3) is a sound system of inference rules, hence in every instance that satisfies F, $X \to A$ must hold. However, the only way this could happen in instance r is if $A \in X^{(i)}$. ■

Let us observe that the relation instance r given in the proof above provides the completeness proof for the Armstrong-axioms, as well. Indeed, the closure X^+ calculated by CLOSURE(R,F,X) is the set of those attributes for which $X \to A$ follows from F by the Armstrong-axioms. Meanwhile, for every other attribute B, there exist two rows of r that agree on X, but differ in B, that is $F \models X \to B$ ***does not*** hold.

The running tome of CLOSURE(R,F,X) is $O(n^2)$, where n is the length of he input. Indeed, in the **repeat** – **until** loop of lines 4–11 every not yet used dependency is checked, and the body of the loop is executed at most $|R \setminus X| + 1$ times, since it is started again only if $X^{(i-1)} \neq X^{(i)}$, that is a new attribute is added to the closure of X. However, the running time can be reduced to linear with appropriate bookkeeping.

1. For every yet unused $W \to Z$ dependency of F it is kept track of how many attributes of W are not yet included in the closure ($i[W, Z]$).
2. For every attribute A those yet unused dependencies are kept in a doubly linked list L_A whose left side contains A.
3. Those not yet used dependencies $W \to Z$ are kept in a linked list J, whose left side W's every attribute is contained in the closure already, that is for which $i[W, Z] = 0$.

It is assumed that the family of functional dependencies F is given as a set of attribute pairs (W, Z), representing $W \to Z$. The LINEAR-CLOSURE(R,F,X) algorithm is a modification of CLOSURE(R,F,X) using the above bookkeeping, whose running time is linear. R is the schema, F is the given family of functional dependencies, and we are to determine the closure of attribute set X.

Algorithm LINEAR-CLOSURE(R,F,X) consists of two parts. In the initialisation phase (lines 1–13) the lists are initialised. The loops of lines 2–5 and 6–8, respectively, take $O(\sum_{(W,Z)\in F} |W|)$ time. The loop in lines 9–11 means $O(|F|)$ steps. If the length of the input is denoted by n, then this is $O(n)$ steps altogether.

During the execution of lines 14–23, every functional dependency (W, Z) is examined at most once, when it is taken off from list J. Thus, lines 15–16 and 23 take

at most $|F|$ steps. The running time of the loops in line 17–22 can be estimated by observing that the sum $\sum i[W,Z]$ is decreased by one in each execution, hence it takes $O(\sum i_0[W,Z])$ steps, where $i_0[W,Z]$ is the $i[W,Z]$ value obtained in the initialisation phase. However, $\sum i_0[W,Z] \leq \sum_{(W,Z)\in F} |W|$, thus lines 14–23 also take $O(n)$ time in total.

LINEAR-CLOSURE(R,F,X)

```
 1                                                    ▷ Initialisation phase.
 2  for all (W,Z) ∈ F
 3      do for all A ∈ W
 4             do add (W,Z) to list L_A
 5      i[W,Z] ← 0
 6  for all A ∈ R \ X
 7      do for all (W,Z) of list L_A
 8             do i[W,Z] ← i[W,Z] + 1
 9  for all (W,Z) ∈ F
10      do if i[W,Z] = 0
11            then add (W,Z) to list J
12  X⁺ ← X
13                                                   ▷ End of initialisation phase.
14  while J is nonempty
15        do (W,Z) ← head(J)
16           delete (W,Z) from list J
17           for all A ∈ Z \ X⁺
18                do for all (W,Z) of list L_A
19                       do i[W,Z] ← i[W,Z] − 1
20                          if i[W,Z] = 0
21                             then add (W,Z) to list J
22                                  delete (W,Z)from list L_A
23        X⁺ ← X⁺ ∪ Z
24  return X⁺
```

18.1.3. Minimal cover

Algorithm LINEAR-CLOSURE(R,F,X) can be used to test equivalence of systems of dependencies. Let F and G be two families of functional dependencies. F and G are said to be ***equivalent,*** if exactly the same functional dependencies follow from both, that is $F^+ = G^+$. It is clear that it is enough to check for all functional dependencies $X \to Y$ in F whether it belongs to G^+, and vice versa, for all $W \to Z$ in G, whether it is in F^+. Indeed, if some of these is not satisfied, say $X \to Y$ is not in G^+, then surely $F^+ \neq G^+$. On the other hand, if all $X \to Y$ are in G^+, then a proof of a functional dependency $U \to V$ from F^+ can be obtained from dependencies in G in such a way that to the derivation of the dependencies $X \to Y$ of F from G, the derivation of $U \to V$ from F is concatenated. In order to decide that a dependency $X \to Y$ from F is in G^+, it is enough to construct the closure $X^+(G)$

of attribute set X with respect to G using LINEAR-CLOSURE(R,G,X), then check whether $Y \subseteq X^+(G)$ holds. The following special functional dependency system equivalent with F is useful.

Definition 18.5 *The system of functional dependencies G is a* ***minimal cover*** *of the family of functional dependencies F iff G is equivalent with F, and*

1. *functional dependencies of G are in the form $X \to A$, where A is an attribute and $A \notin X$,*
2. *no functional dependency can be dropped from G, i.e., $(G - \{X \to A\})^+ \subsetneqq G^+$,*
3. *the left sides of dependencies in G are minimal, that is $X \to A \in G$, $Y \subsetneqq X \Longrightarrow ((G - \{X \to A\}) \cup \{Y \to A\})^+ \neq G^+$.*

Every set of functional dependencies have a minimal cover, namely algorithm MINIMAL-COVER(R, F) constructs one.

MINIMAL-COVER(R, F)

```
 1 G ← ∅
 2 for all X → Y ∈ F
 3     do for all A ∈ Y − X
 4            do G ← G ∪ X → A
 5                          ▷ Each right hand side consists of a single attribute.
 6 for all X → A ∈ G
 7     do while there exists B ∈ X: A ∈ (X − B)⁺(G)
 8            X ← X − B
 9                                  ▷ Each left hand side is minimal.
10 for all X → A ∈ G
11     do if A ∈ X⁺(G − {X → A})
12          then G ← G − {X → A}
13                                  ▷ No redundant dependency exists.
```

After executing the loop of lines 2–4, the right hand side of each dependency in G consists of a single attribute. The equality $G^+ = F^+$ follows from the union rule of Lemma 18.2 and the reflexivity axiom. Lines 6–8 minimise the left hand sides. In line 11 it is checked whether a given functional dependency of G can be removed without changing the closure. $X^+(G - \{X \to A\})$ is the closure of attribute set X with respect to the family of functional dependencies $G - \{X \to A\}$.

Proposition 18.6 MINIMAL-COVER*(R, F) calculates a minimal cover of F.*

Proof It is enough to show that during execution of the loop in lines 10–12, no functional dependency $X \to A$ is generated whose left hand side could be decreased. Indeed, if a $X \to A$ dependency would exist, such that for some $Y \subsetneqq X$ $Y \to A \in G^+$ held, then $Y \to A \in G'^+$ would also hold, where G' is the set of dependencies considered when $X \to A$ is checked in lines 6–8. $G \subseteq G'$, which implies $G^+ \subseteq G'^+$ (see Exercise 18.1-1). Thus, X should have been decreased already during execution of the loop in lines 6–8. ■

18.1.4. Keys

In database design it is important to identify those attribute sets that uniquely determine the data in individual records.

Definition 18.7 *Let (R, F) be a relational schema. The set of attributes $X \subseteq R$ is called a* ***superkey****, if $X \to R \in F^+$. A superkey X is called a* ***key****, if it is minimal with respect to containment, that is no proper subset $Y \subsetneq X$ is key.*

The question is how the keys can be determined from (R, F)? What makes this problem hard is that the number of keys could be super exponential function of the size of (R, F). In particular, Yu and Johnson constructed such relational schema, where $|F| = k$, but the number of keys is $k!$. Békéssy and Demetrovics gave a beautiful and simple proof of the fact that starting from k functional dependencies, at most $k!$ key can be obtained. (This was independently proved by Osborne and Tompa.)

The proof of Békéssy and Demetrovics is based on the operation $*$ they introduced, which is defined for functional dependencies.

Definition 18.8 *Let $e_1 = U \to V$ and $e_2 = X \to Y$ be two functional dependencies. The binary operation $*$ is defined by*

$$e_1 * e_2 = U \cup ((R - V) \cap X) \to V \cup Y \ .$$

Some properties of operation $*$ is listed, the proof is left to the Reader (Exercise 18.1-3). Operation $*$ is associative, furthermore it is idempotent in the sense that if $e = e_1 * e_2 * \cdots * e_k$ and $e' = e * e_i$ for some $1 \leq i \leq k$, then $e' = e$.

Proposition 18.9 (Békéssy and Demetrovics). *Let (R, F) be a relational schema and let $F = \{e_1, e_2, \ldots, e_k\}$ be a listing of the functional dependencies. If X is a key, then $X \to R = e_{\pi_1} * e_{\pi_2} * \ldots * e_{\pi_s} * d$, where $(\pi_1, \pi_2, \ldots, \pi_s)$ is an ordered subset of the index set $\{1, 2, \ldots, k\}$, and d is a trivial dependency in the form $D \to D$.*

Proposition 18.9 bounds in some sense the possible sets of attributes in the search for keys. The next proposition gives lower and upper bounds for the keys.

Proposition 18.10 *Let (R, F) be a relational schema and let $F = \{U_i \to V_i \colon 1 \leq i \leq k\}$. Let us assume without loss of generality that $U_i \cap V_i = \emptyset$. Let $\mathcal{U} = \bigcup_{i=1}^{k} U_i$ and $\mathcal{V} = \bigcup_{i=1}^{k} V_i$. If K is a key in the schema (R, F), then*

$$\mathcal{H}_L = R - \mathcal{V} \subseteq K \subseteq (R - \mathcal{V}) \cup \mathcal{U} = \mathcal{H}_U.$$

The proof is not too hard, it is left as an exercise for the Reader (Exercise 18.1-4). The algorithm List-keys(R, F) that lists the keys of the schema (R, F) is based on the bounds of Proposition 18.10. The running time can be bounded by $O(n!)$, but one cannot expect any better, since to list the output needs that much time in worst case.

```
LIST-KEYS(R, F)
1                                   ▷ Let U and V be as defined in Proposition 18.10
2  if U ∩ V = ∅
3     then return R − V
4                                   ▷ R − V is the only key.
5  if (R − V)^+ = R
6     then return R − V
7                                   ▷ R − V is the only key.
8  K ← ∅
9  for all permutations A_1, A_2, ... A_h of the attributes of U ∩ V
10    do K ← (R − V) ∪ U
11       for i ← 1 to h
12          do Z ← K − A_i
13             if Z^+ = R
14                then K ← Z
15       K ← K ∪ {K}
16 return K
```

Exercises

18.1-1 Let R be a relational schema and let F and G be families of functional dependencies over R. Show that

a. $F \subseteq F^+$.

b. $(F^+)^+ = F^+$.

c. If $F \subseteq G$, then $F^+ \subseteq G^+$.

Formulate and prove similar properties of the closure X^+ – with respect to F – of an attribute set X.

18.1-2 Derive the functional dependency $AB \to F$ from the set of dependencies $G = \{AB \to C, A \to D, CD \to EF\}$ using Armstrong-axioms (A1)–(A3).

18.1-3 Show that operation $*$ is associative, furthermore if for functional dependencies $e_1, e_2, \ldots, e_k$ we have $e = e_1 * e_2 * \cdots * e_k$ and $e' = e * e_i$ for some $1 \leq i \leq k$, then $e' = e$.

18.1-4 Prove Proposition 18.10.

18.1-5 Prove the union, pseudo transitivity and decomposition rules of Lemma 18.2.

18.2. Decomposition of relational schemata

A ***decomposition*** of a relational schema $R = \{A_1, A_2, \ldots, A_n\}$ is a collection $\rho = \{R_1, R_2, \ldots, R_k\}$ of subsets of R such that

$$R = R_1 \cup R_2 \cup \cdots \cup R_k \ .$$

The R_i's need not be disjoint, in fact in most application they must not be. One important motivation of decompositions is to avoid ***anomalies***.

Example 18.3 *Anomalies* Consider the following schema

SUPPLIER-INFO(SNAME,ADDRESS,ITEM,PRICE)

This schema encompasses the following problems:

1. ***Redundancy.*** The address of a supplier is recorded with every item it supplies.
2. ***Possible inconsistency (update anomaly).*** As a consequence of redundancy, the address of a supplier might be updated in some records and might not be in some others, hence the supplier would not have a unique address, even though it is expected to have.
3. ***Insertion anomaly.*** The address of a supplier cannot be recorded if it does not supply anything at the moment. One could try to use `NULL` values in attributes ITEM and PRICE, but would it be remembered that it must be deleted, when a supplied item is entered for that supplier? More serious problem that SNAME and ITEM together form a key of the schema, and the `NULL` values could make it impossible to search by an index based on that key.
4. ***Deletion anomaly*** This is the opposite of the above. If all items supplied by a supplier are deleted, then as a side effect the address of the supplier is also lost.

All problems mentioned above are eliminated if schema SUPPLIER-INFO is replaced by two sub-schemata:

SUPPLIER(SNAME,ADDRESS),
SUPPLIES(SNAME,ITEM,PRICE).

In this case each suppliers address is recorded only once, and it is not necessary that the supplier supplies a item in order its address to be recorded. For the sake of convenience the attributes are denoted by single characters S (SNAME), A (ADDRESS), I (ITEM), P (PRICE).

Question is that is it correct to replace the schema $SAIP$ by SA and SIP? Let r be and instance of schema $SAIP$. It is natural to require that if SA and SIP is used, then the relations belonging to them are obtained projecting r to SA and SIP, respectively, that is $r_{SA} = \pi_{SA}(r)$ and $r_{SIP} = \pi_{SIP}(r)$. r_{SA} and r_{SIP} contains the same information as r, if r can be reconstructed using only r_{SA} and r_{SIP}. The calculation of r from r_{SA} and r_{SIP} can bone by the ***natural join*** operator.

Definition 18.11 *The **natural join** of relations r_i of schemata R_i $(i = 1, 2, \ldots n)$ is the relation s belonging to the schema $\cup_{i=1}^{n} R_i$, which consists of all rows μ that for all i there exists a row ν_i of relation r_i such that $\mu[R_i] = \nu_i[R_i]$. In notation $s = \bowtie_{i=1}^{n} r_i$.*

Example 18.4 Let $R_1 = AB$, $R_2 = BC$, $r_1 = \{ab, a'b', ab''\}$ and $r_2\{bc, bc', b'c''\}$. The natural join of r_1 and r_2 belongs to the schema $R = ABC$, and it is the relation $r_1 \bowtie r_2 = \{abc, abc', a'b'c''\}$.

If s is the natural join of r_{SA} and r_{SIP}, that is $s = r_{SA} \bowtie r_{SIP}$, then $\pi_{SA}(s) = r_{SA}$ és $\pi_{SIP}(s) = r_{SIP}$ by Lemma 18.12. If $r \neq s$, then the original relation could not be reconstructed knowing only r_{SA} and r_{SIP}.

18.2.1. Lossless join

Let $\rho = \{R_1, R_2, \ldots, R_k\}$ be a decomposition of schema R, furthermore let F be a family of functional dependencies over R. The decomposition ρ is said to have ***lossless join property*** (with respect to F), if every instance r of R that satisfies F also satisfies

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \cdots \bowtie \pi_{R_k}(r) \ .$$

That is, relation r is the natural join of its projections to attribute sets R_i, $i = 1, 2, \ldots, k$. For a decomposition $\rho = \{R_1, R_2, \ldots, R_k\}$, let m_ρ denote the the mapping which assigns to relation r the relation $m_\rho(r) = \bowtie_{i=1}^{k} \pi_{R_i}(r)$. Thus, the lossless join property with respect to a family of functional dependencies means that $r = m_\rho(r)$ for all instances r that satisfy F.

Lemma 18.12 *Let $\rho = \{R_1, R_2, \ldots, R_k\}$ be a decomposition of schema R, and let r be an arbitrary instance of R. Furthermore, let $r_i = \pi_{R_i}(r)$. Then*

1. $r \subseteq m_\rho(r)$.
2. *If $s = m_\rho(r)$, then $\pi_{R_i}(s) = r_i$.*
3. $m_\rho(m_\rho(r)) = m_\rho(r)$.

The proof is left to the Reader (Exercise 18.2-7).

18.2.2. Checking the lossless join property

It is relatively not hard to check that a decomposition $\rho = \{R_1, R_2, \ldots, R_k\}$ of schema R has the lossless join property. The essence of algorithm JOIN-TEST(R, F, ρ) is the following.
A $k \times n$ array T is constructed, whose column j corresponds to attribute A_j, while row i corresponds to schema R_i. $T[i, j] = 0$ if $A_j \in R_i$, otherwise $T[i, j] = i$.

The following step is repeated until there is no more possible change in the array. Consider a functional dependency $X \to Y$ from F. If a pair of rows i and j agree in all attributes of X, then their values in attributes of Y are made equal. More precisely, if one of the values in an attribute of Y is 0, then the other one is set to 0, as well, otherwise it is arbitrary which of the two values is set to be equal to the other one. If a symbol is changed, then ***each*** of its occurrences in that column must be changed accordingly. If at the end of this process there is an all 0 row in T, then the decomposition has the lossless join property, otherwise, it is lossy.

JOIN-TEST(R, F, ρ)

```
 1                                                          ▷ Initialisation phase.
 2  for i ← 1 to |ρ|
 3      do for j ← 1 to |R|
 4             do if A_j ∈ R_i
 5                   then T[i, j] ← 0
 6                   else T[i, j] ← i
 7                                                          ▷ End of initialisation phase.
 8  S ← T
 9  repeat
10          T ← S
11          for all {X → Y} ∈ F
12              do for i ← 1 to |ρ| − 1
13                     do for j ← i + 1 to |R|
14                            do if for all A_h in X (S[i, h] = S[j, h])
15                                  then EQUATE(i, j, S, Y)
16  until S = T
17  if there exist an all 0 row in S
18    then return "Lossless join"
19    else return "Lossy join"
```

Procedure EQUATE(i, j, S, Y) makes the appropriate symbols equal.

EQUATE(i, j, S, Y)

```
 1  for A_l ∈ Y
 2      do if S[i, l] · S[j, l] = 0
 3            then
 4                 for d ← 1 to k
 5                     do if S[d, l] = S[i, l] ∨ S[d, l] = S[j, l]
 6                           then S[d, l] ← 0
 7            else
 8                 for d ← 1 to k
 9                     do if S[d, l] = S[j, l]
10                           then S[d, l] ← S[i, l]
```

Example 18.5 *Checking lossless join property* Let $R = ABCDE$, $R_1 = AD$, $R_2 = AB$, $R_3 = BE$, $R_4 = CDE$, $R_5 = AE$, furthermore let the functional dependencies be $\{A \to C, B \to C, C \to D, DE \to C, CE \to A\}$. The initial array is shown on Figure 18.1(a). Using $A \to C$ values 1,2,5 in column C can be equated to 1. Then applying $B \to C$ value 3 of column C again can be changed to 1. The result is shown on Figure 18.1(b). Now $C \to D$ can be used to change values 2,3,5 of column D to 0. Then applying $DE \to C$ (the only nonzero) value 1 of column C can be set to 0. Finally, $CE \to A$ makes it possible to change values 3 and 4 in column A to be changed to 0. The final result is shown on Figure 18.1(c). The third row consists of only zeroes, thus the decomposition has the lossless join property.

A	B	C	D	E
0	1	1	0	1
0	0	2	2	2
3	0	3	3	0
4	4	0	0	0
0	5	5	5	0

(a)

A	B	C	D	E
0	1	1	0	1
0	0	1	2	2
3	0	1	3	0
4	4	0	0	0
0	5	1	5	0

(b)

A	B	C	D	E
0	1	0	0	1
0	0	0	2	2
0	0	0	0	0
0	4	0	0	0
0	5	0	0	0

(c)

Figure 18.1. Application of JOIN-TEST(R,F,ρ).

It is clear that the running time of algorithm JOIN-TEST(R,F,ρ) is polynomial in the length of the input. The important thing is that it uses only the schema, not the instance r belonging to the schema. Since the size of an instance is larger than the size of the schema by many orders of magnitude, the running time of an algorithm using the schema only is negligible with respect to the time required by an algorithm processing the data stored.

Theorem 18.13 *Procedure* JOIN-TEST*(R,F,ρ) correctly determines whether a given decomposition has the lossless join property.*

Proof Let us assume first that the resulting array T contains no all zero row. T itself can be considered as a relational instance over the schema R. This relation satisfies all functional dependencies from F, because the algorithm finished since there was no more change in the table during checking the functional dependencies. It is true for the starting table that its projections to every R_i's contain an all zero row, and this property does not change during the running of the algorithm, since a 0 is never changed to another symbol. It follows, that the natural join $m_\rho(T)$ contains the all zero row, that is $T \neq m_\rho(T)$. Thus the decomposition is lossy. The proof of the other direction is only sketched.

Logic, domain calculus is used. The necessary definitions can be found in the books of Abiteboul, Hull and Vianu, or Ullman, respectively. Imagine that variable a_j is written in place of zeroes, and b_{ij} is written in place of i's in column j, and

JOIN-TEST(R,F,ρ) is run in this setting. The resulting table contains row $a_1a_2\dots a_n$, which corresponds to the all zero row. Every table can be viewed as a shorthand notation for the following domain calculus expression

$$\{a_1a_2\dots a_n \mid (\exists b_{11})\dots(\exists b_{kn})\,(R(w_1)\wedge\dots\wedge R(w_k))\}\,, \tag{18.1}$$

where w_i is the ith row of T. If T is the starting table, then formula (18.1) defines m_ρ exactly. As a justification note that for a relation r, $m_\rho(r)$ contains the row $a_1a_2\dots a_n$ iff r contains for all i a row whose jth coordinate is a_j if A_j is an attribute of R_i, and arbitrary values represented by variables b_{il} in the other attributes.

Consider an arbitrary relation r belonging to schema R that satisfies the dependencies of F. The modifications (equating symbols) of the table done by JOIN-TEST(R,F,ρ) do not change the set of rows obtained from r by (18.1), if the modifications are done in the formula, as well. Intuitively it can be seen from the fact that only such symbols are equated in (18.1), that can only take equal values in a relation satisfying functional dependencies of F. The exact proof is omitted, since it is quiet tedious.

Since in the result table of JOIN-TEST(R,F,ρ) the all a's row occurs, the domain calculus formula that belongs to this table is of the following form:

$$\{a_1a_2\dots a_n \mid (\exists b_{11})\dots(\exists b_{kn})\,(R(a_1a_2\dots a_n)\wedge\dots)\}\,. \tag{18.2}$$

It is obvious that if (18.2) is applied to relation r belonging to schema R, then the result will be a subset of r. However, if r satisfies the dependencies of F, then (18.2) calculates $m_\rho(r)$. According to Lemma 18.12, $r\subseteq m_\rho(r)$ holds, thus if r satisfies F, then (18.2) gives back r exactly, so $r=m_\rho(r)$, that is the decomposition has the lossless join property. ■ Procedure JOIN-TEST(R,F,ρ) can be used independently of the number of parts occurring in the decomposition. The price of this generality is paid in the running time requirement. However, if R is to be decomposed only into ***two*** parts, then CLOSURE(R,F,X) or LINEAR-CLOSURE(R,F,X) can be used to obtain the same result faster, according to the next theorem.

Theorem 18.14 *Let $\rho=(R_1,R_2)$ be a decomposition of R, furthermore let F be a set of functional dependencies. Decomposition ρ has the lossless join property with respect to F iff*

$$(R_1\cap R_2)\to(R_1-R_2)\ \ \textit{or}\ \ (R_1\cap R_2)\to(R_2-R_1)\,.$$

These dependencies need not be in F, it is enough if they are in F^+.

Proof The starting table in procedure JOIN-TEST(R,F,ρ) is the following:

	$R_1\cap R_2$	R_1-R_2	R_2-R_1
row of R_1	$00\dots0$	$00\dots0$	$11\dots1$
row of R_2	$00\dots0$	$22\dots2$	$00\dots0$

(18.3)

It is not hard to see using induction on the number of steps done by JOIN-TEST(R,F,ρ) that if the algorithm changes both values of the column of an attribute A to 0, then $A\in(R_1\cap R_2)^+$. This is obviously true at the start. If at some time

values of column A must be equated, then by lines 11–14 of the algorithm, there exists $\{X \to Y\} \in F$, such that the two rows of the table agree on X, and $A \in Y$. By the induction assumption $X \subseteq (R_1 \cap R_2)^+$ holds. Applying Armstrong-axioms (transitivity and reflexivity), $A \in (R_1 \cap R_2)^+$ follows.

On the other hand, let us assume that $A \in (R_1 \cap R_2)^+$, that is $(R_1 \cap R_2) \to A$. Then this functional dependency can be derived from F using Armstrong-axioms. By induction on the length of this derivation it can be seen that procedure JOIN-TEST(R,F,ρ) will equate the two values of column A, that is set them to 0. Thus, the row of R_1 will be all 0 iff $(R_1 \cap R_2) \to (R_2 - R_1)$, similarly, the row of R_2 will be all 0 iff $(R_1 \cap R_2) \to (R_1 - R_2)$. ■

18.2.3. Dependency preserving decompositions

The lossless join property is important so that a relation can be recovered from its projections. In practice, usually not the relation r belonging to the underlying schema R is stored, but relations $r_i = r[R_i]$ for an appropriate decomposition $\rho = (R_1, R_2, \ldots, R_k)$, in order to avoid anomalies. The functional dependencies F of schema R are ***integrity constraints*** of the database, relation r is consistent if it satisfies all prescribed functional dependencies. When during the life time of the database updates are executed, that is rows are inserted into or deleted from the projection relations, then it may happen that the natural join of the new projections does not satisfy the functional dependencies of F. It would be too costly to join the projected relations – and then project them again – after each update to check the integrity constraints. However, the ***projection*** of the family of functional dependencies F to an attribute set Z can be defined: $\pi_Z(F)$ consists of those functional dependencies $\{X \to Y\} \in F^+$, where $XY \subseteq Z$. After an update, if relation r_i is changed, then it is relatively easy to check whether $\pi_{R_i}(F)$ still holds. Thus, it would be desired if family F would be logical implication of the families of functional dependencies $\pi_{R_i}(F)$ $i = 1, 2, \ldots, k$. Let $\pi_\rho(F) = \bigcup_{i=1}^{k} \pi_{R_i}(F)$.

Definition 18.15 *The decomposition ρ is said to be **dependency preserving**, if*

$$\pi_\rho(F)^+ = F^+.$$

Note that $\pi_\rho(F) \subseteq F^+$, hence $\pi_\rho(F)^+ \subseteq F^+$ always holds. Consider the following example.

Example 18.6 Let R = (**C**ity,**S**treet,**Z**ip code) be the underlying schema, furthermore let $F = \{CS \to Z, Z \to C\}$ be the functional dependencies. Let the decomposition ρ be $\rho = (CZ, SZ)$. This has the lossless join property by Theorem 18.14. $\pi_\rho(F)$ consists of $Z \to C$ besides the trivial dependencies. Let $R_1 = CZ$ and $R_2 = SZ$. Two rows are inserted into each of the projections belonging to schemata R_1 and R_2, respectively, so that functional dependencies of the projections are satisfied:

R_1	C	Z
	Fort Wayne	46805
	Fort Wayne	46815

R_2	S	Z
	Coliseum Blvd	46805
	Coliseum Blvd	46815

In this case R_1 and R_2 satisfy the dependencies prescribed for them separately, however in $R_1 \bowtie R_2$ the dependency $CS \to Z$ does not hold.

It is true as well, that none of the decompositions of this schema preserves the dependency $CS \to Z$. Indeed, this is the only dependency that contains Z on the right hand side, thus if it is to be preserved, then there has to be a subschema that contains C, S, Z, but then the decomposition would not be proper. This will be considered again when decomposition into normal forms is treated.

Note that it may happen that decomposition ρ preserves functional dependencies, but does not have the lossless join property. Indeed, let $R = ABCD$, $F = \{A \to B, C \to D\}$, and let the decomposition be $\rho = (AB, CD)$.

Theoretically it is very simple to check whether a decomposition $\rho = (R_1, R_2, \ldots R_k)$ is dependency preserving. Just F^+ needs to be calculated, then projections need to be taken, finally one should check whether the union of the projections is equivalent with F. The main problem with this approach is that even calculating F^+ may need exponential time.

Nevertheless, the problem can be solved without explicitly determining F^+. Let $G = \pi_\rho(F)$. G will not be calculated, only its equivalence with F will be checked. For this end, it needs to be decidable for all functional dependencies $\{X \to Y\} \in F$ that if X^+ is taken with respect to G, whether it contains Y. The trick is that X^+ is determined ***without*** full knowledge of G by repeatedly taking the effect to the closure of the projections of F onto the individual R_i's. That is, the concept of S-operation on an attribute set Z is introduced, where S is another set of attributes: Z is replaced by $Z \cup ((Z \cap S)^+ \cap S)$, where the closure is taken with respect to F. Thus, the closure of the part of Z that lies in S is taken with respect to F, then from the resulting attributes those are added to Z, which also belong to S.

It is clear that the running time of algorithm PRESERVE(ρ, F) is polynomial in the length of the input. More precisely, the outermost **for** loop is executed at most once for each dependency in F (it may happen that it turns out earlier that some dependency is not preserved). The body of the **repeat–until** loop in lines 3–7. requires linear number of steps, it is executed at most $|R|$ times. Thus, the body of the **for** loop needs quadratic time, so the total running time can be bounded by the cube of the input length.

PRESERVE(ρ, F)

```
 1 for all (X → Y) ∈ F
 2     do Z ← X
 3        repeat
 4                 W ← Z
 5                 for i ← 1 to k
 6                     do Z ← Z ∪ (LINEAR-CLOSURE(R, F, Z ∩ R_i) ∩ R_i)
 7        until Z = W
 8     if Y ⊈ Z
 9       then return "Not dependency preserving"
10 return "Dependency preserving"
```

Example 18.7 Consider the schema $R = ABCD$, let the decomposition be $\rho = \{AB, BC, CD\}$, and dependencies be $F = \{A \to B, B \to C, C \to D, D \to A\}$. That is, by the visible cycle of the dependencies, every attribute determines all others. Since D and A do not occur together in the decomposition one might think that the dependency $D \to A$ is not preserved, however this intuition is wrong. The reason is that during the projection to AB, not only the dependency $A \to B$ is obtained, but $B \to A$, as well, since not F, but F^+ is projected. Similarly, $C \to B$ and $D \to C$ are obtained, as well, but $D \to A$ is a logical implication of these by the transitivity of the Armstrong axioms. Thus it is expected that PRESERVE(ρ, F) claims that $D \to A$ is preserved.

Start from the attribute set $Y = \{D\}$. There are three possible operations, the AB-operation, the BC-operation and the CD-operation. The first two obviously does not add anything to $\{D\}^+$, since $\{D\} \cap \{A, B\} = \{D\} \cap \{B, C\} = \emptyset$, that is the closure of the empty set should be taken, which is empty (in the present example). However, using the CD-operation:

$$\begin{aligned} Z &= \{D\} \cup \ (\{D\} \cap \{C, D\})^+ \cap \{C, D\} \\ &= \{D\} \cup \ \{D\}^+ \cap \{C, D\} \\ &= \{D\} \cup (\{A, B, C, D\} \cap \{C, D\}) \\ &= \{C, D\}. \end{aligned}$$

In the next round using the BC-operation the actual $Z = \{C, D\}$ is changed to $Z = \{B, C, D\}$, finally applying the AB-operation on this, $Z = \{A, B, C, D\}$ is obtained. This cannot change, so procedure PRESERVE(ρ, F) stops. Thus, with respect to the family of functional dependencies

$$G = \pi_{AB}(F) \cup \pi_{BC}(F) \cup \pi_{CD}(F) \ ,$$

$\{D\}^+ = \{A, B, C, D\}$ holds, that is $G \models D \to A$. It can be checked similarly that the other dependencies of F are in G^+ (as a fact in G).

Theorem 18.16 *The procedure* PRESERVE(ρ, F) *determines correctly whether the decomposition ρ is dependency preserving.*

Proof It is enough to check for a single functional dependency $X \to Y$ whether whether the procedure decides correctly if it is in G^+. When an attribute is added to Z in lines 3–7, then Functional dependencies from G are used, thus by the soundness of the Armstrong-axioms if PRESERVE(ρ, F) claims that $X \to Y \in G^+$, then it is indeed so.

On the other hand, if $X \to Y \in G^+$, then LINEAR-CLOSURE(R, F, X) (run by G as input) adds the attributes of Y one-by-one to X. In every step when an attribute is added, some functional dependency $U \to V$ of G is used. This dependency is in one of $\pi_{R_i}(F)$'s, since G is the union of these. An easy induction on the number of functional dependencies used in procedure LINEAR-CLOSURE(R, F, X) shows that sooner or later Z becomes a subset of U, then applying the R_i-operation all attributes of V are added to Z. ■

18.2.4. Normal forms

The goal of transforming (decomposing) relational schemata into ***normal forms*** is to avoid the anomalies described in the previous section. Normal forms of many

different strengths were introduced in the course of evolution of database theory, here only the ***Boyce–Codd*** normal formát (BCNF) and the ***third,*** furthermore ***fourth*** normal form (3NF and 4NF) are treated in detail, since these are the most important ones from practical point of view.

Boyce-Codd normal form

Definition 18.17 *Let R be relational schema, F be a family of functional dependencies over R. (R, F) is said to be in* ***Boyce-Codd normal form*** *if $X \to A \in F^+$ and $A \not\subseteq X$ implies that A is a superkey.*

The most important property of BCNF is that it eliminates redundancy. This is based on the following theorem whose proof is left to the Reader as an exercise (Exercise 18.2-8).

Theorem 18.18 *Schema (R, F) is in BCNF iff for arbitrary attribute $A \in R$ and key $X \subset R$ there exists no $Y \subseteq R$, for which $X \to Y \in F^+$; $Y \to X \notin F^+$; $Y \to A \in F^+$ and $A \notin Y$.*

In other words, Theorem 18.18 states that "BCNF $\iff$ There is no transitive dependence on keys". Let us assume that a given schema is not in BCNF, for example $C \to B$ and $B \to A$ hold, but $B \to C$ does not, then the same B value could occur besides many different C values, but at each occasion the same A value would be stored with it, which is redundant. Formulating somewhat differently, the meaning of BCNF is that (only) using functional dependencies an attribute value in a row cannot be predicted from other attribute values. Indeed, assume that there exists a schema R, in which the value of an attribute can be determined using a functional dependency by comparison of two rows. That is, there exists two rows that agree on an attribute set X, differ on the set Y and the value of the remaining (unique) attribute A can be determined in one of the rows from the value taken in the other row.

X	Y	A
x	y_1	a
x	y_2	?

If the value ? can be determined by a functional dependency, then this value can only be a, the dependency is $Z \to A$, where Z is an appropriate subset of X. However, Z cannot be a superkey, since the two rows are distinct, thus R is not in BCNF.

3NF

Although BCNF helps eliminating anomalies, it is not true that every schema can be decomposed into subschemata in BCNF so that the decomposition is dependency preserving. As it was shown in Example 18.6, no proper decomposition of schema CSZ preserves the $CS \to Z$ dependency. At the same time, the schema is clearly not in BCNF, because of the $Z \to C$ dependency.

Since dependency preserving is important because of consistency checking of a database, it is practical to introduce a normal form that every schema has dependency preserving decomposition into that form, and it allows minimum possible

redundancy. An attribute is called ***prime attribute,*** if it occurs in a key.

Definition 18.19 *The schema (R, F) is in **third normal form,** if whenever $X \to A \in F^+$, then either X is a superkey, or A is a prime attribute.*

The schema $SAIP$ of Example 18.3 with the dependencies $SI \to P$ and $S \to A$ is not in 3NF, since SI is the only key and so A is not a prime attribute. Thus, functional dependency $S \to A$ violates the 3NF property.

3NF is clearly weaker condition than BCNF, since "or A is a prime attribute" occurs in the definition. The schema CSZ in Example 18.6 is trivially in 3NF, because every attribute is prime, but it was already shown that it is not in BCNF.

Testing normal forms

Theoretically every functional dependency in F^+ should be checked whether it violates the conditions of BCNF or 3NF, and it is known that F^+ can be exponentially large in the size of F. Nevertheless, it can be shown that if the functional dependencies in F are of the form that the right hand side is a single attribute always, then it is enough to check violation of BCNF, or 3NF respectively, for dependencies of F. Indeed, let $X \to A \in F^+$ be a dependency that violates the appropriate condition, that is X is not a superkey and in case of 3NF, A is not prime. $X \to A \in F^+ \iff A \in X^+$. In the step when $\textsc{Closure}(R, F, X)$ puts A into $X+$ (line 8) it uses a functional dependency $Y \to A$ from F that $Y \subset X^+$ and $A \notin Y$. This dependency is non-trivial and A is (still) not prime. Furthermore, if Y were a superkey, than by $R = Y^+ \subseteq (X^+)^+ = X^+$, X would also be a superkey. Thus, the functional dependency $Y \to A$ from F violates the condition of the normal form. The functional dependencies easily can be checked in polynomial time, since it is enough to calculate the closure of the left hand side of each dependency. This finishes checking for BCNF, because if the closure of each left hand side is R, then the schema is in BCNF, otherwise a dependency is found that violates the condition. In order to test 3NF it may be necessary to decide about an attribute whether it is prime or not. However this problem is NP-complete, see Problem *18-4*.

Lossless join decomposition into BCNF

Let (R, F) be a relational schema (where F is the set of functional dependencies). The schema is to be decomposed into union of subschemata $R_1, R_2, \ldots, R_k$, such that the decomposition has the lossless join property, furthermore each R_i endowed with the set of functional dependencies $\pi_{R_i}(F)$ is in BCNF. The basic idea of the decomposition is simple:

- If (R, F) is in BCNF, then ready.
- If not, it is decomposed into two proper parts (R_1, R_2), whose join is lossless.
- Repeat the above for R_1 and R_2.

In order to see that this works one has to show two things:

- If (R, F) is not in BCNF, then it has a lossless join decomposition into smaller parts.
- If a part of a lossless join decomposition is further decomposed, then the new decomposition has the lossless join property, as well.

Lemma 18.20 *Let (R, F) be a relational schema (where F is the set of functional dependencies), $\rho = (R_1, R_2, \ldots, R_k)$ be a lossless join decomposition of R. Furthermore, let $\sigma = (S_1, S_2)$ be a lossless join decomposition of R_1 with respect to $\pi_{R_1}(F)$. Then $(S_1, S_2, R_2, \ldots, R_k)$ is a lossless join decomposition of R.*

The proof of Lemma 18.20 is based on the associativity of natural join. The details are left to the Reader (Exercise 18.2-9).

This can be applied for a simple, but unfortunately exponential time algorithm that decomposes a schema into subschemata of BCNF property. The projections in lines 4–5 of NAÏV-BCNF(S, G) may be of exponential size in the length of the input. In order to decompose schema (R, F), the procedure must be called with parameters R, F. Procedure NAÏV-BCNF(S, G) is recursive, S is the actual schema with set of functional dependencies G. It is assumed that the dependencies in G are of the form $X \to A$, where A is a single attribute.

NAIV-BCNF(S, G)

```
1 while there exists {X → A} ∈ G, that violates BCNF
2       do S1 ← {XA}
3          S2 ← S − A
4          G1 ← π_{S1}(G)
5          G2 ← π_{S2}(G)
6          return (NAIV-BCNF(S1, G1), NAIV-BCNF(S2, G2))
7 return S
```

However, if the algorithm is allowed overdoing things, that is to decompose a schema even if it is already in BCNF, then there is no need for projecting the dependencies. The procedure is based on the following two lemmae.

Lemma 18.21

1. *A schema of only two attributes is in BCNF.*
2. *If R is not in BCNF, then there exists two attributes A and B in R, such that $(R - AB) \to A$ holds.*

Proof If the schema consists of two attributes, $R = AB$, then there are at most two possible non-trivial dependencies, $A \to B$ and $B \to A$. It is clear, that if some of them holds, then the left hand side of the dependency is a key, so the dependency does not violate the BCNF property. However, if none of the two holds, then BCNF is trivially satisfied.

On the other hand, let us assume that the dependency $X \to A$ violates the BCNF property. Then there must exists an attribute $B \in R - (XA)$, since otherwise X would be a superkey. For this B, $(R - AB) \to A$ holds. ■

Let us note, that the converse of the second statement of Lemma 18.21 is not true. It may happen that a schema R is in BCNF, but there are still two attributes $\{A, B\}$ that satisfy $(R - AB) \to A$. Indeed, let $R = ABC$, $F = \{C \to A, C \to B\}$.

This schema is obviously in BCNF, nevertheless $(R - AB) = C \to A$.

The main contribution of Lemma 18.21 is that the projections of functional dependencies need not be calculated in order to check whether a schema obtained during the procedure is in BCNF. It is enough to calculate $(R - AB)^+$ for pairs $\{A, B\}$ of attributes, which can be done by LINEAR-CLOSURE(R, F, X) in linear time, so the whole checking is polynomial (cubic) time. However, this requires a way of calculating $(R - AB)^+$ without actually projecting down the dependencies. The next lemma is useful for this task.

Lemma 18.22 *Let $R_2 \subset R_1 \subset R$ and let F be the set of functional dependencies of scheme R. Then*

$$\pi_{R_2}(\pi_{R_1}(F)) = \pi_{R_2}(F) \ .$$

The proof is left for the Reader (Exercise 18.2-10). The method of lossless join BCNF decomposition is as follows. Schema R is decomposed into two subschemata. One is XA that is in BCNF, satisfying $X \to A$. The other subschema is $R - A$, hence by Theorem 18.14 the decomposition has the lossless join property. This is applied recursively to $R - A$, until such a schema is obtained that satisfies property 2 of Lemma 18.21. The lossless join property of this recursively generated decomposition is guaranteed by Lemma 18.20.

POLYNOMIAL-BCNF(R, F)

```
 1 Z ← R
 2     ▷ Z is the schema that is not known to be in BCNF during the procedure.
 3 ρ ← ∅
 4 while there exist A, B in Z, such that A ∈ (Z − AB)^+ and |Z| > 2
 5       do Let A and B be such a pair
 6          E ← A
 7          Y ← Z − B
 8          while there exist C, D in Y, such that C ∈ (Z − CD)^+
 9                do Y ← Y − D
10                   E ← C
11          ρ ← ρ ∪ {Y}
12          Z ← Z − E
13 ρ ← ρ ∪ {Z}
14 return ρ
```

The running time of POLYNOMIAL-BCNF(R, F) is polynomial, in fact it can be bounded by $O(n^5)$, as follows. During each execution of the loop in lines 4–12 the size of Z is decreased by at least one, so the loop body is executed at most n times. $(Z - AB)^+$ is calculated in line 4 for at most $O(n^2)$ pairs that can be done in linear time using LINEAR-CLOSURE that results in $O(n^3)$ steps for each execution of the loop body. In lines 8–10 the size of Y is decreased in each iteration, so during each execution of lines 3–12, they give at most n iteration. The condition of the command **while** of line 8 is checked for $O(n^2)$ pairs of attributes, each checking is done in linear time. The running time of the algorithm is dominated by the time required by lines 8–10 that take $n \cdot n \cdot O(n^2) \cdot O(n) = O(n^5)$ steps altogether.

Dependency preserving decomposition into 3NF

We have seen already that its is not always possible to decompose a schema into subschemata in BCNF so that the decomposition is dependency preserving. Nevertheless, if only 3NF is required then a decomposition can be given using MINIMAL-COVER(R, F). Let R be a relational schema and F be the set of functional dependencies. Using MINIMAL-COVER(R, F) a minimal cover G of F is constructed. Let $G = \{X_1 \to A_1, X_2 \to A_2, \ldots, X_k \to A_k\}$.

Theorem 18.23 *The decomposition $\rho = (X_1A_1, X_2A_2, \ldots, X_kA_k)$ is dependency preserving decomposition of R into subschemata in 3NF.*

Proof Since $G^+ = F^+$ and the functional dependency $X_i \to A_i$ is in $\pi_{R_i}(F)$, the decomposition preserves every dependency of F. Let us suppose indirectly, that the schema $R_i = X_iA_i$ is not in 3NF, that is there exists a dependency $U \to B$ that violates the conditions of 3NF. This means that the dependency is non-trivial and U is not a superkey in R_i and B is not a prime attribute of R_i. There are two cases possible. If $B = A_i$, then using that U is not a superkey $U \subsetneqq X_i$ follows. In this case the functional dependency $U \to A_i$ contradicts to that $X_i \to A_i$ was a member of minimal cover, since its left hand side could be decreased. In the case when $B \neq A_i$, $B \in X_i$ holds. B is not prime in R_i, thus X_i is not a key, only a superkey. However, then X_i would contain a key Y such that $Y \subsetneqq X_i$. Furthermore, $Y \to A_i$ would hold, as well, that contradicts to the minimality of G since the left hand side of $X_i \to A_i$ could be decreased. ■

If the decomposition needs to have the lossless join property besides being dependency preserving, then ρ given in Theorem 18.23 is to be extended by a key X of R. Although it was seen before that it is not possible to list ***all*** keys in polynomial time, ***one*** can be obtained in a simple greedy way, the details are left to the Reader (Exercise 18.2-11).

Theorem 18.24 *Let (R, F) be a relational schema, and let $G = \{X_1 \to A_1, X_2 \to A_2, \ldots, X_k \to A_k\}$ be a minimal cover of F. Furthermore, let X be a key in (R, F). Then the decomposition $\tau = (X, X_1A_1, X_2A_2, \ldots, X_kA_k)$ is a lossless join and dependency preserving decomposition of R into subschemata in 3NF.*

Proof It was shown during the proof of Theorem 18.23 that the subschemata $R_i = X_iA_i$ are in 3NF for $i = 1, 2, \ldots, k$. There cannot be a non-trivial dependency in the subschema $R_0 = X$, because if it were, then X would not be a key, only a superkey.

The lossless join property of τ is shown by the use of JOIN-TEST(R, G, ρ) procedure. Note that it is enough to consider the minimal cover G of F. More precisely, we show that the row corresponding to X in the table will be all 0 after running JOIN-TEST(R, G, ρ). Let $A_1, A_2, \ldots, A_m$ be the order of the attributes of $R - X$ as CLOSURE(R,G,X) inserts them into X^+. Since X is a key, every attribute of $R - X$ is taken during CLOSURE(R,G,X). It will be shown by induction on i that the element in row of X and column of A_i is 0 after running JOIN-TEST(R, G, ρ).

The base case of $i = 0$ is obvious. Let us suppose that the statement is true for $i-$ and consider when and why A_i is inserted into X^+. In lines 6–8 of CLOSURE(R,G,X) such a functional dependency $Y \to A_i$ is used where $Y \subseteq X \cup \{A_1, A_2, \ldots, A_{i-1}\}$. Then $Y \to A_i \in G$, $YA_i = R_j$ for some j. The rows corresponding to X and

$YA_i = R_j$ agree in columns of X (all 0 by the induction hypothesis), thus the entries in column of A_i are equated by JOIN-TEST(R, G, ρ). This value is 0 in the row corresponding to $YA_i = R_j$, thus it becomes 0 in the row of X, as well. ■

It is interesting to note that although an arbitrary schema can be decomposed into subschemata in 3NF in polynomial time, nevertheless it is NP-complete to decide whether a given schema (R, F) is in 3NF, see Problem *18-4*. However, the BCNF property can be decided in polynomial time. This difference is caused by that in order to decide 3NF property one needs to decide about an attribute whether it is prime. This latter problem requires the listing of all keys of a schema.

18.2.5. Multivalued dependencies

Example 18.8 Besides functional dependencies, some other dependencies hold in Example 18.1, as well. There can be several lectures of a subject in different times and rooms. Part of an instance of the schema could be the following.

Professor	Subject	Room	Student	Grade	Time
Caroline Doubtfire	Analysis	MA223	John Smith	A^-	Monday 8–10
Caroline Doubtfire	Analysis	CS456	John Smith	A^-	Wednesday 12–2
Caroline Doubtfire	Analysis	MA223	Ching Lee	A^+	Monday 8–10
Caroline Doubtfire	Analysis	CS456	Ching Lee	A^+	Wednesday 12–2

A set of values of Time and Room attributes, respectively, belong to each given value of Subject, and all other attribute values are repeated with these. Sets of attributes SR and StG are independent, that is their values occur in each combination.

The set of attributes Y is said to be ***multivalued dependent*** on set of attributes X, in notation $X \twoheadrightarrow Y$, if for every value on X, there exists a set of values on Y that is not dependent in any way on the values taken in $R - X - Y$. The precise definition is as follows.

Definition 18.25 *The relational schema R satisfies the **multivalued dependency** $X \twoheadrightarrow Y$, if for every relation r of schema R and arbitrary tuples t_1, t_2 of r that satisfy $t_1[X] = t_2[X]$, there exists tuples $t_3, t_4 \in r$ such that*

- $t_3[XY] = t_1[XY]$
- $t_3[R - XY] = t_2[R - XY]$
- $t_4[XY] = t_2[XY]$
- $t_4[R - XY] = t_1[R - XY]$

holds.[1]

In Example 18.8 S$\twoheadrightarrow$TR holds.

Remark 18.26 *Functional dependency is **equality generating** dependency, that is from the equality of two objects it deduces the equality of other other two objects. On the other hand, multivalued dependency is **tuple generating dependency**, that*

[1] It would be enough to require the existence of t_3, since the existence of t_4 would follow. However, the symmetry of multivalued dependency is more apparent in this way.

is the existence of two rows that agree somewhere implies the existence of some other rows.

There exists a sound and complete axiomatisation of multivalued dependencies similar to the Armstrong-axioms of functional dependencies. Logical implication and inference can be defined analogously. The multivalued dependency $X \twoheadrightarrow Y$ is ***logically implied*** by the set M of multivalued dependencies, in notation $M \models X \twoheadrightarrow Y$, if every relation that satisfies all dependencies of M also satisfies $X \twoheadrightarrow Y$.

Note, that $X \rightarrow Y$ implies $X \twoheadrightarrow Y$. The rows t_3 and t_4 of Definition 18.25 can be chosen as $t_3 = t_2$ and $t_4 = t_1$, respectively. Thus, functional dependencies and multivalued dependencies admit a common axiomatisation. Besides Armstrong-axioms (A1)–(A3), five other are needed. Let R be a relational schema.

(A4) ***Complementation***: $\{X \twoheadrightarrow Y\} \models X \twoheadrightarrow (R - X - Y)$.

(A5) ***Extension***: If $X \twoheadrightarrow Y$ holds, and $V \subseteq W$, then $WX \twoheadrightarrow VY$.

(A6) ***Transitivity***: $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \models X \twoheadrightarrow (Z - Y)$.

(A7) $\{X \rightarrow Y\} \models X \twoheadrightarrow Y$.

(A8) If $X \twoheadrightarrow Y$ holds, $Z \subseteq Y$, furthermore for some W disjoint from Y $W \rightarrow Z$ holds, then $X \rightarrow Z$ is true, as well.

Beeri, Fagin and Howard proved that (A1)–(A8) is sound and complete system of axioms for functional and Multivalued dependencies together. Proof of soundness is left for the Reader (Exercise 18.2-12), the proof of the completeness exceeds the level of this book. The rules of Lemma 18.2 are valid in exactly the same way as when only functional dependencies were considered. Some further rules are listed in the next Proposition.

Proposition 18.27 *The followings are true for multivalued dependencies.*

1. ***Union rule:*** $\{X \twoheadrightarrow Y, X \twoheadrightarrow Z\} \models X \twoheadrightarrow YZ$.
2. ***Pseudotransitivity:*** $\{X \twoheadrightarrow Y, WY \twoheadrightarrow Z\} \models WX \twoheadrightarrow (Z - WY)$.
3. ***Mixed pseudotransitivity:*** $\{X \twoheadrightarrow Y, XY \rightarrow Z\} \models X \rightarrow (Z - Y)$.
4. ***Decomposition rule*** *for multivalued dependencies: ha* $X \twoheadrightarrow Y$ *and* $X \twoheadrightarrow Z$ *holds, then* $X \twoheadrightarrow (Y \cap Z)$, $X \twoheadrightarrow (Y - Z)$ *and* $X \twoheadrightarrow (Z - Y)$ *holds, as well.*

Th proof of Proposition 18.27 is left for the Reader (Exercise 18.2-13).

Dependency basis

Important difference between functional dependencies and multivalued dependencies is that $X \rightarrow Y$ immediately implies $X \rightarrow A$ for all A in Y, however $X \twoheadrightarrow A$ is deduced by the decomposition rule for multivalued dependencies from $X \twoheadrightarrow Y$ only if there exists a set of attributes Z such that $X \twoheadrightarrow Z$ and $Z \cap Y = A$, or $Y - Z = A$. Nevertheless, the following theorem is true.

Theorem 18.28 *Let R be a relational schema, $X \subset R$ be a set of attributes. Then there exists a partition $Y_1, Y_2, \ldots, Y_k$ of the set of attributes $R - X$ such that for $Z \subseteq R - X$ the multivalued dependency $X \twoheadrightarrow Z$ holds if and only if Z is the union of some Y_i's.*

Proof We start from the one-element partition $W_1 = R - X$. This will be refined successively, while the property that $X \twoheadrightarrow W_i$ holds for all W_i in the actual decomposition, is kept. If $X \twoheadrightarrow Z$ and Z is not a union of some of the W_i's, then replace every W_i such that neither $W_i \cap Z$ nor $W_i - Z$ is empty by $W_i \cap Z$ and $W_i - Z$. According to the decomposition rule of Proposition 18.27, both $X \twoheadrightarrow (W_i \cap Z)$ and $X \twoheadrightarrow (W_i - Z)$ hold. Since $R - X$ is finite, the refinement process terminates after a finite number of steps, that is for all Z such that $X \twoheadrightarrow Z$ holds, Z is the union of some blocks of the partition. In order to complete the proof one needs to observe only that by the union rule of Proposition 18.27, the union of some blocks of the partition depends on X in multivalued way. ■

Definition 18.29 *The partition $Y_1, Y_2, \ldots, Y_k$ constructed in Theorem 18.28 from a set D of functional and multivalued dependencies is called the* ***dependency basis*** *of X (with respect to D).*

Example 18.9 Consider the familiar schema

R(**P**rofessor,**Su**bject,**R**oom,**St**udent,**G**rade,**T**ime)

of Examples 18.1 and 18.8. **Su** ↠ **RT** was shown in Example 18.8. By the complementation rule **Su** ↠ **PStG** follows. **Su**→**P** is also known. This implies by axiom (A7) that **Su** ↠ **P**. By the decomposition rule **Su** ↠ **Stg** follows. It is easy to see that no other one-element attribute set is determined by **Su** via multivalued dependency. Thus, the dependency basis of **Su** is the partition{**P**,**RT**,**StG**}.

We would like to compute the set D^+ of logical consequences of a given set D of functional and multivalued dependencies. One possibility is to apply axioms (A1)–(A8) to extend the set of dependencies repeatedly, until no more extension is possible. However, this could be an exponential time process in the size of D. One cannot expect any better, since it was shown before that even D^+ can be exponentially larger than D. Nevertheless, in many applications it is not needed to compute the whole set D^+, one only needs to decide whether a given functional dependency $X \to Y$ or multivalued dependency $X \twoheadrightarrow Y$ belongs to D^+ or not. In order to decide about a multivalued dependency $X \twoheadrightarrow Y$, it is enough to compute the dependency basis of X, then to check whether $Z - X$ can be written as a union of some blocks of the partition. The following is true.

Theorem 18.30 (Beeri). *In order to compute the dependency basis of a set of attributes X with respect to a set of dependencies D, it is enough to consider the following set M of multivalued dependencies:*

1. *All multivalued dependencies of D and*
2. *for every $X \to Y$ in D the set of multivalued dependencies $X \twoheadrightarrow A_1, X \twoheadrightarrow A_2, \ldots, X \twoheadrightarrow A_k$, where $Y = A_1 A_2 \ldots A_k$, and the A_i's are single attributes.*

The only thing left is to decide about functional dependencies based on the dependency basis. CLOSURE(R,F,X) works correctly only if multivalued dependencies are not considered. The next theorem helps in this case.

Theorem 18.31 (Beeri). *Let us assume that $A \notin X$ and the dependency basis of X with respect to the set M of multivalued dependencies obtained in Theorem 18.30 is known. $X \to A$ holds if and only if*

1. *A forms a single element block in the partition of the dependency basis, and*
2. *There exists a set Y of attributes that does not contain A, $Y \to Z$ is an element of the originally given set of dependencies D, furthermore $A \in Z$.*

Based on the observations above, the following polynomial time algorithm can be given to compute the dependency basis of a set of attributes X.

DEPENDENCY-BASIS(R, M, X)

```
1 S ← {R − X}                          ▷ The collection of sets in the dependency basis is S.
2 repeat
3       for all V ↠ W ∈ M
4           do if there exists Y ∈ S such that Y ∩ W ≠ ∅ ∧ Y ∩ V = ∅
5                 then S ← S − {{Y}} ∪ {{Y ∩ W}, {Y − W}}
6 until S does not change
7 return S
```

It is immediate that if $\mathcal{S}$ changes in lines 3–5. of DEPENDENCY-BASIS(R, M, X), then some block of the partition is cut by the algorithm. This implies that the running time is a polynomial function of the sizes of M and R. In particular, by careful implementation one can make this polynomial to $O(|M| \cdot |R|^3)$, see Problem 18-5.

Fourth normal form 4NF

The Boyce-Codd normal form can be generalised to the case where multivalued dependencies are also considered besides functional dependencies, and one needs to get rid of the redundancy caused by them.

Definition 18.32 *Let R be a relational schema, D be a set of functional and multivalued dependencies over R. R is in **fourth normal form (4NF)**, if for arbitrary multivalued dependency $X \twoheadrightarrow Y \in D^+$ for which $Y \not\subseteq X$ and $R \neq XY$, holds that X is superkey in R.*

Observe that 4NF$\Longrightarrow$BCNF. Indeed, if $X \to A$ violated the BCNF condition, then $A \notin X$, furthermore XA could not contain all attributes of R, because that would imply that X is a superkey. However, $X \to A$ implies $X \twoheadrightarrow A$ by (A8), which in turn would violate the 4NF condition.

Schema R together with set of functional and multivalued dependencies D can be decomposed into $\rho = (R_1, R_2, \ldots, R_k)$, where each R_i is in 4NF and the decomposition has the lossless join property. The method follows the same idea as the decomposition into BCNF subschemata. If schema S is not in 4NF, then there exists a multivalued dependency $X \twoheadrightarrow Y$ in the projection of D onto S that violates the 4NF condition. That is, X is not a superkey in S, Y neither is empty, nor is a subset of X, furthermore the union of X and Y is not S. It can be assumed without loss of generality that X and Y are disjoint, since $X \twoheadrightarrow (Y - X)$ is implied by $X \twoheadrightarrow Y$

using (A1), (A7) and the decomposition rule. In this case S can be replaced by subschemata $S_1 = XY$ and $S_2 = S - Y$, each having a smaller number of attributes than S itself, thus the process terminates in finite time.

Two things has to be dealt with in order to see that the process above is correct.

- Decomposition S_1, S_2 has the lossless join property.
- How can the projected dependency set $\pi_S(D)$ be computed?

The first problem is answered by the following theorem.

Theorem 18.33 *The decomposition $\rho = (R_1, R_2)$ of schema R has the lossless join property with respect to a set of functional and multivalued dependencies D iff*

$$(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2).$$

Proof The decomposition $\rho = (R_1, R_2)$ of schema R has the lossless join property iff for any relation r over the schema R that satisfies all dependencies from D holds that if μ and ν are two tuples of r, then there exists a tuple φ satisfying $\varphi[R_1] = \mu[R_1]$ and $\varphi[R_2] = \nu[R_2]$, then it is contained in r. More precisely, φ is the natural join of the projections of μ on R_1 and of ν on R_2, respectively, which exist iff $\mu[R_1 \cap R_2] = \nu[R_1 \cap R_2]$. Thus the fact that φ is always contained in r is equivalent with that $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$. ■

To compute the projection $\pi_S(D)$ of the dependency set D one can use the following theorem of Aho, Beeri and Ullman. $\pi_S(D)$ is the set of multivalued dependencies that are logical implications of D and use attributes of S only.

Theorem 18.34 (Aho, Beeri és Ullman). *$\pi_S(D)$ consists of the following dependencies:*

- *For all $X \to Y \in D^+$, if $X \subseteq S$, then $X \to (Y \cap S) \in \pi_S(D)$.*
- *For all $X \twoheadrightarrow Y \in D^+$, if $X \subseteq S$, then $X \twoheadrightarrow (Y \cap S) \in \pi_S(D)$.*

Other dependencies cannot be derived from the fact that D holds in R.

Unfortunately this theorem does not help in computing the projected dependencies in polynomial time, since even computing D^+ could take exponential time. Thus, the algorithm of 4NF decomposition is not polynomial either, because the 4NF condition must be checked with respect to the projected dependencies in the subschemata. This is in deep contrast with the case of BCNF decomposition. The reason is, that to check BCNF condition one does not need to compute the projected dependencies, only closures of attribute sets need to be considered according to Lemma 18.21.

Exercises

18.2-1 Are the following inference rules sound?

a. If $XW \to Y$ and $XY \to Z$, then $X \to (Z - W)$.
b. If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow Z$.
c. If $X \twoheadrightarrow Y$ and $XY \to Z$, then $X \to Z$.

18.2-2 Prove Theorem 18.30, that is show the following. Let D be a set of functional

and multivalued dependencies, and let $m(D) = \{X \twoheadrightarrow Y : X \twoheadrightarrow Y \in D\} \cup \{X \twoheadrightarrow A : A \in Y \text{ for some } X \to Y \in D\}$. Then

a. $D \models X \to Y \Longrightarrow m(D) \models X \twoheadrightarrow Y$, and

b.$D \models X \twoheadrightarrow Y \iff m(D) \models X \twoheadrightarrow Y$.

Hint. Use induction on the inference rules to prove b.

18.2-3 Consider the database of an investment firm, whose attributes are as follows: B (stockbroker), O (office of stockbroker), I (investor), S (stock), A (amount of stocks of the investor), D (dividend of the stock). The following functional dependencies are valid: $S \to D$, $I \to B$, $IS \to A$, $B \to O$.

a. Determine a key of schema $R = BOISAD$.

b. How many keys are in schema R?

c. Give a lossless join decomposition of R into subschemata in BCNF.

d. Give a dependency preserving and lossless join decomposition of R into subschemata in 3NF.

18.2-4 The schema R of Exercise 18.2-3 is decomposed into subschemata SD, IB, ISA and BO. Does this decomposition have the lossless join property?

18.2-5 Assume that schema R of Exercise 18.2-3 is represented by ISA, IB, SD and ISO subschemata. Give a minimal cover of the projections of dependencies given in Exercise 18.2-3. Exhibit a minimal cover for the union of the sets of projected dependencies. Is this decomposition dependency preserving?

18.2-6 Let the functional dependency $S \to D$ of Exercise 18.2-3 be replaced by the multivalued dependency $S \twoheadrightarrow D$. That is , D represents the stock's dividend "history".

a. Compute the dependency basis of I.

b. Compute the dependency basis of BS.

c. Give a decomposition of R into subschemata in 4NF.

18.2-7 Consider the decomposition $\rho = \{R_1, R_2, \ldots, R_k\}$ of schema R. Let $r_i = \pi_{R_i}(r)$, furthermore $m_\rho(r) = \bowtie_{i=1}^{k} \pi_{R_i}(r)$. Prove:

a. $r \subseteq m_\rho(r)$.

b. If $s = m_\rho(r)$, then $\pi_{R_i}(s) = r_i$.

c. $m_\rho(m_\rho(r)) = m_\rho(r)$.

18.2-8 Prove that schema (R, F) is in BCNF iff for arbitrary $A \in R$ and key $X \subset R$, it holds that there exists no $Y \subseteq R$, for which $X \to Y \in F^+$; $Y \to X \notin F^+$; $Y \to A \in F^+$ and $A \notin Y$.

18.2-9 Prove Lemma 18.20.

18.2-10 Let us assume that $R_2 \subset R_1 \subset R$ and the set of functional dependencies of schema R is F. Prove that $\pi_{R_2}(\pi_{R_1}(F)) = \pi_{R_2}(F)$.

18.2-11 Give a $O(n^2)$ running time algorithm to find a key of the relational schema (R, F). *Hint.* Use that R is superkey and each superkey contains a key. Try to drop attributes from R one-by-one and check whether the remaining set is still a key.

18.2-12 Prove that axioms (A1)–(A8) are sound for functional and multivalued dependencies.

18.2-13 Derive the four inference rules of Proposition 18.27 from axioms (A1)–(A8).

18.3. Generalised dependencies

Two such dependencies will be discussed in this section that are generalizations of the previous ones, however cannot be axiomatised with axioms similar to (A1)–(A8).

18.3.1. Join dependencies

Theorem 18.33 states that multivalued dependency is equivalent with that some decomposition the schema into two parts has the lossless join property. Its generalisation is the ***join dependency***.

Definition 18.35 *Let R be a relational schema and let $R = \bigcup_{i=1}^{k} X_i$. The relation r belonging to R is said to satisfy the* ***join dependency***

$$\bowtie[X_1, X_2, \ldots, X_k]$$

if

$$r = \bowtie_{i=1}^{k} \pi_{X_i}(r) \ .$$

In this setting r satisfies multivalued dependency $X \twoheadrightarrow Y$ iff it satisfies the join dependency $\bowtie[XY, X(R-Y)]$. The join dependency $\bowtie[X_1, X_2, \ldots, X_k]$ expresses that the decomposition $\rho = (X_1, X_2, \ldots, X_k)$ has the lossless join property. One can define the ***fifth normal form, 5NF***.

Definition 18.36 *The relational schema R is in* ***fifth normal form,*** *if it is in 4NF and has no non-trivial join dependency.*

The fifth normal form has theoretical significance primarily. The schemata used in practice usually have ***primary keys***. Using that the schema could be decomposed into subschemata of two attributes each, where one of the attributes is a superkey in every subschema.

Example 18.10 Consider the database of clients of a bank (Clientnumber,Name,Address,accountBalance). Here C is unique identifier, thus the schema could be decomposed into (CN,CA,CB), which has the lossless join property. However, it is not worth doing so, since no storage place can be saved, furthermore no anomalies are avoided with it.

There exists an ***axiomatisation*** of a dependency system if there is a finite set of inference rules that is sound and complete, i.e. logical implication coincides with being derivable by using the inference rules. For example, the Armstrong-axioms give an axiomatisation of functional dependencies, while the set of rules (A1)–(A8) is the same for functional and multivalued dependencies considered together. Unfortunately, the following negative result is true.

Theorem 18.37 *The family of join dependencies has no finite axiomatisation.*

In contrary to the above, Abiteboul, Hull and Vianu show in their book that the logical implication problem can be decided by an algorithm for the family of functional and join dependencies taken together. The complexity of the problem is as follows.

Theorem 18.38

- *It is NP-complete to decide whether a given join dependency is implied by another given join dependency and a functional dependency.*
- *It is NP-hard to decide whether a given join dependency is implied by given set of multivalued dependencies.*

18.3.2. Branching dependencies

A generalisation of functional dependencies is the family of ***branching dependencies***. Let us assume that $A, B \subset R$ and there exists no $q+1$ rows in relation r over schema R, such that they contain at most p distinct values in columns of A, but all $q+1$ values are pairwise distinct in some column of B. Then B is said to be (p,q)***-dependent*** on A, in notation $A \xrightarrow{p,q} B$. In particular, $A \xrightarrow{1,1} B$ holds if and only if functional dependency $A \to B$ holds.

Example 18.11 Consider the database of the trips of an international transport truck.

- One trip: four distinct countries.
- One country has at most five neighbours.
- There are 30 countries to be considered.

Let x_1, x_2, x_3, x_4 be the attributes of the countries reached in a trip. In this case $x_i \xrightarrow{1,1} x_{i+1}$ does not hold, however another dependency is valid:

$$x_i \xrightarrow{1,5} x_{i+1} \ .$$

The storage space requirement of the database can be significantly reduced using these dependencies. The range of each element of the original table consists of 30 values, names of countries or some codes of them (5 bits each, at least). Let us store a little table ($30 \times 5 \times 5 = 750$ bits) that contains a numbering of the neighbours of each country, which assigns to them the numbers 0,1,2,3,4 in some order. Now we can replace attribute x_2 by these numbers (x_2^*), because the value of x_1 gives the starting country and the value of x_2^* determines the second country with the help of the little table. The same holds for the attribute x_3, but we can decrease the number of possible values even further, if we give a table of numbering the possible third countries for each x_1, x_2 pair. In this case, the attribute x_3^* can take only 4 different values. The same holds for x_4, too. That is, while each element of the original table could be encoded by 5 bits, now for the cost of two little auxiliary tables we could decrease the length of the elements in the second column to 3 bits, and that of the elements in the third and fourth columns to 2 bits.

The (p,q)-closure of an attribute set $X \subset R$ can be defined:

$$C_{p,q}(X) = \{A \in R \colon X \xrightarrow{p,q} A\} \ .$$

In particular, $C_{1,1}(X) = X^+$. In case of branching dependencies even such basic questions are hard as whether there exists an ***Armstrong-relation*** for a given family of dependencies.

Definition 18.39 *Let R be a relational schema, F be a set of dependencies of some dependency family $\mathcal{F}$ defined on R. A relation r over schema R is* ***Armstrong-relation*** *for F, if the set of dependencies from $\mathcal{F}$ that r satisfies is exactly F, that is $F = \{\sigma \in \mathcal{F} \colon r \models \sigma\}$.*

Armstrong proved that for an arbitrary set of functional dependencies F there exists Armstrong-relation for F^+. The proof is based on the three properties of closures of attributes sets with respect to F, listed in Exercise 18.1-1 For branching dependencies only the first two holds in general.

Lemma 18.40 *Let $0 < p \leq q$, furthermore let R be a relational schema. For $X, Y \subseteq R$ one has*

1. $X \subseteq C_{p,q}(X)$ *and*
2. $X \subseteq Y \Longrightarrow C_{p,q}(X) \subseteq C_{p,q}(Y)$.

There exists such $C \colon 2^R \to 2^R$ mapping and natural numbers p, q that there exists no Armstrong-relation for C in the family if (p, q)-dependencies.

Grant Minker investigated ***numerical dependencies*** that are similar to branching dependencies. For attribute sets $X, Y \subseteq R$ the dependency $X \xrightarrow{k} Y$ holds in a relation r over schema R if for every tuple value taken on the set of attributes X, there exists at most k distinct tuple values taken on Y. This condition is stronger than that of $X \xrightarrow{1,k} Y$, since the latter only requires that in each column of Y there are at most k values, independently of each other. That allows $k^{|Y-X|}$ different Y projections. Numerical dependencies were axiomatised in some special cases, based on that Katona showed that branching dependencies have no finite axiomatisation. It is still an open problem whether logical implication is algorithmically decidable amongst branching dependencies.

Exercises

18.3-1 Prove Theorem 18.38.

18.3-2 Prove Lemma 18.40.

18.3-3 Prove that if $p = q$, then $C_{p,p}\left(C_{p,p}(X)\right) = C_{p,p}(X)$ holds besides the two properties of Lemma 18.40.

18.3-4 A $C \colon 2^R \to 2^R$ mapping is called a ***closure***, if it satisfies the two properties of Lemma 18.40 and and the third one of Exercise 18.3-3. Prove that if $C \colon 2^R \to 2^R$ is a closure, and F is the family of dependencies defined by $X \to Y \iff Y \subseteq C(X)$, then there exists an Armstrong-relation for F in the family of $(1, 1)$-dependencies (functional dependencies) and in the family of $(2, 2)$-dependencies, respectively.

18.3-5 Let C be the closure defined by

$$C(X) = \begin{cases} X, & \text{if } |X| < 2 \\ R & \text{otherwise} \end{cases}.$$

Prove that there exists no Armstrong-relation for C in the family of (n, n)-dependencies, if $n > 2$.

Problems

18-1 External attributes

Maier calls attribute A an ***external attribute*** in the functional dependency $X \to Y$ with respect to the family of dependencies F over schema R, if the following two conditions hold:

1. $(F - \{X \to Y\}) \cup \{X \to (Y - A)\} \models X \to Y$, or
2. $(F - \{X \to Y\}) \cup \{(X - A) \to Y\} \models X \to Y$.

Design an $O(n^2)$ running time algorithm, whose input is schema (R, F) and output is a set of dependencies G equivalent with F that has no external attributes.

18-2 The order of the elimination steps in the construction of minimal cover is important

In the procedure MINIMAL-COVER(R, F) the set of functional dependencies was changed in two ways: either by dropping redundant dependencies, or by dropping redundant attributes from the left hand sides of the dependencies. If the latter method is used first, until there is no more attribute that can be dropped from some left hand side, then the first method, this way a minimal cover is obtained really, according to Proposition 18.6. Prove that if the first method applied first and then the second, until there is no more possible applications, respectively, then the obtained set of dependencies is not necessarily a minimal cover of F.

18-3 BCNF subschema

Prove that the following problem is coNP-complete: Given a relational schema R with set of functional dependencies F, furthermore $S \subset R$, decide whether $(S, \pi_S(F))$ is in BCNF.

18-4 3NF is hard to recognise

Let (R, F) be a relational schema, where F is the system of functional dependencies. The k ***size key problem*** is the following: given a natural number k, determine whether there exists a key of size at most k.
The ***prime attribute problem*** is the following: for a given $A \in R$, determine whether it is a prime attribute.

a. Prove that the k size key problem is NP-complete. *Hint.* Reduce the ***vertex cover*** problem to the prime attribute problem.

b. Prove that the prime attribute problem is NP-complete by reducing the k size key problem to it.

c. Prove that determining whether the relational schema (R, F) is in 3NF is NP-complete. *Hint.* Reduce the prime attribute problem to it.

***18-5 Running time of* DEPENDENCY-BASIS**

Give an implementation of procedure DEPENDENCY-BASIS, whose running time is $O(|M| \cdot |R|^3)$.

Chapter notes

The relational data model was introduced by Codd [66] in 1970. Functional dependencies were treated in his paper of 1972 [69], their axiomatisation was completed by Armstrong [20]. The logical implication problem for functional dependencies were investigated by Beeri and Bernstein [33], furthermore Maier [231]. Maier also treats the possible definitions of minimal covers, their connections and the complexity of their computations in that paper. Maier, Mendelzon and Sagiv found method to decide logical implications among general dependencies [232]. Beeri, Fagin and Howard proved that axiom system (A1)–(A8) is sound and complete for functional and multivalued dependencies taken together [35]. Yu and Johnson [352] constructed such relational schema, where $|F| = k$ and the number of keys is $k!$. Békéssy and Demetrovics [43] gave a simple and beautiful proof for the statement, that from k functional dependencies at most $k!$ keys can be obtained, thus Yu and Johnson's construction is extremal.

Armstrong-relations were introduced and studied by Fagin [102, 101], furthermore by Beeri, Fagin, Dowd and Statman [34].

Multivalued dependencies were independently discovered by Zaniolo [355], Fagin [100] and Delobel [83].

The necessity of normal forms was recognised by Codd while studying update anomalies [67, 68]. The Boyce–Codd normal form was introduced in [70]. The definition of the third normal form used in this chapter was given by Zaniolo [356]. Complexity of decomposition into subschemata in certain normal forms was studied by Lucchesi and Osborne [224], Beeri and Bernstein [33], furthermore Tsou and Fischer [327].

Theorems 18.30 and 18.31 are results of Beeri [32]. Theorem 18.34 is from a paper of Aho, Beeri és Ullman [6].

Theorems 18.37 and 18.38 are from the book of Abiteboul, Hull and Vianu [3], the non-existence of finite axiomatisation of join dependencies is Petrov's result [270].

Branching dependencies were introduced by Demetrovics, Katona and Sali, they studied existence of Armstrong-relations and the size of minimal Armstrong-relations [84, 85, 86, 291]. Katona showed that there exists no finite axiomatisation of branching dependencies in (ICDT'92 Berlin, invited talk) but never published.

Possibilities of axiomatisation of numerical dependencies were investigated by Grant and Minker [141, 142].

Good introduction of the concepts of this chapter can be found in the books of Abiteboul, Hull and Vianu [3], Ullman [332] furthermore Thalheim [321], respectively.

19. Query Rewriting in Relational Databases

In chapter "Relational database design" basic concepts of relational databases were introduced, such as relational schema, relation, instance. Databases were studied from the designer point of view, the main question was how to avoid redundant data storage, or various anomalies arising during the use of the database.

In the present chapter the schema is considered to be given and focus is on fast and efficient ways of answering user queries. First, basic (theoretical) query languages and their connections are reviewed in Section 19.1.

In the second part of this chapter (Section 19.2) views are considered. Informally, a view is nothing else, but result of a query. Use of views in query efficiency, providing physical data independence and data integration is explained.

Finally, the third part of the present chapter (Section 19.3) introduces query rewriting.

19.1. Queries

Consider the database of cinemas in Budapest. Assume that the schema consists of three relations:

$$\mathbf{CinePest} = \{\mathit{Film}, \mathit{Theater}, \mathit{Show}\} \ . \tag{19.1}$$

The schemata of individual relations are as follows:

$$\begin{aligned} \mathit{Film} &= \{\mathbf{T}\mathit{itle}, \mathbf{D}\mathit{irector}, \mathbf{A}\mathit{ctor}\} \ , \\ \mathit{Theater} &= \{\mathbf{Th}\mathit{eater}, \mathbf{A}\mathit{ddress}, \mathbf{P}\mathit{hone}\} \ , \\ \mathit{Show} &= \{\mathbf{Th}\mathit{eater}, \mathbf{T}\mathit{itle}, \mathbf{Ti}\mathit{me}\} \ . \end{aligned} \tag{19.2}$$

Possible values of instances of each relation are shown on Figure 19.1.

Typical user queries could be:

19.1 Who is the director of "Control"?

19.2 List the names address of those theatres where Kurosawa films are played.

19.3 Give the names of directors who played part in some of their films.

These queries define a mapping from the relations of database schema **CinePest** to

Film

T*itle*	**D***irector*	**A***ctor*
Control	Antal, Nimród	Csányi, Sándor
Control	Antal, Nimród	Mucsi, Zoltán
Control	Antal, Nimród	Pindroch, Csaba
⋮	⋮	⋮
Rashomon	Akira Kurosawa	Toshiro Mifune
Rashomon	Akira Kurosawa	Machiko Kyo
Rashomon	Akira Kurosawa	Mori Masayuki

Theatre

Th*eater*	**A***ddress*	**P***hone*
Bem	II., Margit Blvd. 5/b.	316-8708
Corvin	VIII., Corvin alley 1.	459-5050
Európa	VII., Rákóczi st. 82.	322-5419
Művész	VI., Teréz blvd. 30.	332-6726
⋮	⋮	⋮
Uránia	VIII., Rákóczi st. 21.	486-3413
Vörösmarty	VIII., Üllői st. 4.	317-4542

Show

Th*eater*	**T***itle*	**Ti***me*
Bem	Rashomon	19:00
Bem	Rashomon	21:30
Uránia	Control	18:15
Művész	Rashomon	16:30
Művész	Control	17:00
⋮	⋮	⋮
Corvin	Control	10:15

Figure 19.1. The database **CinePest**.

some other schema (in the present case to schemata of single relations). Formally, ***query*** and ***query mapping*** should be distinguished. The former is a syntactic concept, the latter is a mapping from the set of instances over the input schema to the set of instances over the output schema, that is determined by the query according to some semantic interpretation. However, for both concepts the word "query" is used for the sake of convenience, which one is meant will be clear from the context.

Definition 19.1 *Queries q_1 and q_2 over schema R are said to be* ***equivalent,*** *in*

notation $q_1 \equiv q_2$, if they have the same output schema and for every instance $\mathcal{I}$ over schema R $q_1(\mathcal{I}) = q_2(\mathcal{I})$ holds.

In the remaining of this chapter the most important query languages are reviewed. The expressive powers of query languages need to be compared.

Definition 19.2 *Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be query languages (with appropriate semantics). $\mathcal{Q}_2$ is **dominated by** $\mathcal{Q}_1$ ($\mathcal{Q}_1$ is **weaker,** than $\mathcal{Q}_2$), in notation $\mathcal{Q}_1 \sqsubseteq \mathcal{Q}_2$, if for every query q_1 of $\mathcal{Q}_1$ there exists a query $q_2 \in \mathcal{Q}_2$, such that $q_1 \equiv q_2$. $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are **equivalent,** if $\mathcal{Q}_1 \sqsubseteq \mathcal{Q}_2$ and $\mathcal{Q}_1 \sqsupseteq \mathcal{Q}_2$.*

Example 19.1 *Query.* Consider Question 19.2. As a first try the next solution is obtained:

> **if** there exist in relations *Film*, *Theater* and *Show* tuples $(x_T, \text{"Akira Kurosawa"}, x_A)$, (x_{Th}, x_{Ad}, x_P) and (x_{Th}, x_T, x_{Ti})
> **then** put the tuple $(\mathit{Theater} : x_{Th}, \mathit{Address} : x_A)$ into the output relation.

$x_T, x_A, x_{Th}, x_{Ad}, x_P, x_{Ti}$ denote different variables that take their values from the domains of the corresponding attributes, respectively. Using the same variables implicitly marked where should stand identical values in different tuples.

19.1.1. Conjunctive queries

Conjunctive queries are the simplest kind of queries, but they are the easiest to handle and have the most good properties. Three equivalent forms will be studied, two of them based on logic, the third one is of algebraic nature. The name comes from first order logic expressions that contain only existential quantors ($\exists$), furthermore consist of atomic expressions connected with logical "and", that is conjunction.

Datalog – rule based queries
The tuple $(x_1, x_2, \ldots, x_m)$ is called ***free tuple*** if the x_i's are variables or constants. This is a generalisation of a tuple of a relational instance. For example, the tuple $(x_T, \text{"Akira Kurosawa"}, x_A)$ in Example 19.1 is a free tuple.

Definition 19.3 *Let $\mathbf{R}$ be a relational database schema. **Rule based conjunctive query** is an expression of the following form*

$$ans(u) \leftarrow R_1(u_1), R_2(u_2), \ldots, R_n(u_n) , \tag{19.3}$$

where $n \geq 0$, $R_1, R_2, \ldots, R_n$ are relation names from $\mathbf{R}$, ans is a relation name not in $\mathbf{R}$, $u, u_1, u_2, \ldots, u_n$ are free tuples. Every variable occurring in u must occur in one of $u_1, u_2, \ldots, u_n$, as well.

The rule based conjunctive query is also called a ***rule*** for the sake of simplicity. $ans(u)$ is the ***head*** of the rule, $R_1(u_1), R_2(u_2), \ldots, R_n(u_n)$ is the ***body*** of the rule, $R_i(u_i)$ is called a ***(relational) atom***. It is assumed that each variable of the head also occurs in some atom of the body.

A rule can be considered as some tool that tells how can we deduce newer and newer ***facts,*** that is tuples, to include in the output relation. If the variables of the rule can be assigned such values that each atom $R_i(u_i)$ is true (that is the appropriate tuple is contained in the relation R_i), then tuple u is added to the relation *ans*. Since all variables of the head occur in some atoms of the body, one never has to consider ***infinite*** domains, since the variables can take values from the actual instance queried. Formally. let $\mathcal{I}$ be an instance over relational schema $\mathbf{R}$, furthermore let q be a the query given by rule (19.3). Let $var(q)$ denote the set of variables occurring in q, and let $dom(\mathcal{I})$ denote the set of constants that occur in $\mathcal{I}$. The ***image of*** $\mathcal{I}$ ***under*** q is given by

$$q(\mathcal{I}) = \{\nu(u) | \nu \colon var(q) \ \rightarrow \ dom(\mathcal{I}) \text{ and } \nu(u_i) \in R_i \ i = 1, 2, \ldots, n\} \ . \qquad (19.4)$$

An immediate way of calculating $q(\mathcal{I})$ is to consider all possible valuations ν in some order. There are more efficient algorithms, either by equivalent rewriting of the query, or by using some indices.

An important difference between atoms of the body and the head is that relations $R_1, R_2, \ldots, R_n$ are considered given, (physically) stored, while relation *ans* is not, it is thought to be calculated by the help of the rule. This justifies the names: R_i's are ***extensional relations*** and *ans* is ***intensional relation***.

Query q over schema $\mathbf{R}$ is ***monotone,*** if for instances $\mathcal{I}$ and $\mathcal{J}$ over $\mathbf{R}$, $\mathcal{I} \subseteq \mathcal{J}$ implies $q(\mathcal{I}) \subseteq q(\mathcal{J})$. q is ***satisfiable,*** if there exists an instance $\mathcal{I}$, such that $q(\mathcal{I}) \neq \emptyset$. The proof of the next simple observation is left for the Reader (Exercise 19.1-1).

Proposition 19.4 *Rule based queries are monotone and satisfiable.*

Proposition 19.4 shows the limitations of rule based queries. For example, the query *Which theatres play only Kurosawa films?* is obviously not monotone, hence cannot be expressed by rules of form (19.3).

Tableau queries

If the difference between variables and constants is not considered, then the body of a rule can be viewed as an instance over the schema. This leads to a tabular form of conjunctive queries that is most similar to the visual queries (QBE: Query By Example) of database management system Microsoft Access.

Definition 19.5 *A* ***tableau*** *over the schema* $\mathbf{R}$ *is a generalisation of an instance over* $\mathbf{R}$*, in the sense that variables may occur in the tuples besides constants. The pair* $(\mathbf{T}, u)$ *is a* ***tableau query*** *if* $\mathbf{T}$ *is a tableau and* u *is a free tuple such that all variables of* u *occur in* $\mathbf{T}$*, as well. The free tuple* u *is the* ***summary****.*

The summary row u of tableau query $(\mathbf{T}, u)$ shows which tuples form the result of the query. The essence of the procedure is that the pattern given by tableau $\mathbf{T}$ is searched for in the database, and if found then the tuple corresponding to is included in the output relation. More precisely, the mapping $\nu \colon var(\mathbf{T}) \ \rightarrow \ dom(\mathcal{I})$ is an ***embedding*** of tableau $(\mathbf{T}, u)$ into instance $\mathcal{I}$, if $\nu(\mathbf{T}) \subseteq \mathcal{I}$. The output relation of tableau query $(\mathbf{T}, u)$ consists of all tuples $\nu(u)$ that ν is an embedding of tableau $(\mathbf{T}, u)$ into instance $\mathcal{I}$.

Example 19.2 *Tableau query* Let $\mathbf{T}$ be the following tableau.

Film	*Title*	*Director*	*Actor*
	x_T	"Akira Kurosawa"	x_A

Theater	*Theater*	*Address*	*Phone*
	x_{Th}	x_{Ad}	x_P

Show	*Theater*	*Title*	*Time*
	x_{Th}	x_T	x_{Ti}

The tableau query $(\mathbf{T}, \langle \mathit{Theater}\colon x_{Th}, \mathit{Address}\colon x_{Ad}\rangle)$ answers question 19.2. of the introduction.

The syntax of tableau queries is similar to that of rule based queries. It will be useful later that conditions for one query to contain another one can be easily formulated in the language of tableau queries.

Relational algebra*

A database consists of relations, and a relation is a set of tuples. The result of a query is also a relation with a given attribute set. It is a natural idea that output of a query could be expressed by algebraic and other operations on relations. The ***relational algebra**** consists of the following operations.[1]

Selection: It is of form either $\sigma_{A=c}$ or $\sigma_{A=B}$, where A and B are attributes while c is a constant. The operation can be applied to all such relations R that has attribute A (and B), and its result is relation *ans* that has the same set of attributes as R has, and consists of all tuples that satisfy the ***selection condition***.

Projection: The form of the operation is $\pi_{A_1,A_2,\ldots,A_n}$, $n \geq 0$, where A_i's are distinct attributes. It can be applied to all such relations whose attribute set includes each A_i and its result is the relation *ans* that has attribute set $\{A_1, A_2, \ldots, A_n\}$,

$$val = \{t[A_1, A_2, \ldots, A_n] | t \in R\} ,$$

that is it consists of the restrictions of tuples in R to the attribute set $\{A_1, A_2, \ldots, A_n\}$.

Natural join: This operation has been defined earlier in chapter "Relational database design". Its notation is $\bowtie$, its input consists of two (or more) relations R_1, R_2, with attribute sets V_1, V_2, respectively. The attribute set of the output relation is $V_1 \cup V_2$.

$$R_1 \bowtie R_2 = \{t \text{ tuple over } V_1 \cup V_2 | \exists v \in R_1, \exists w \in R_2, t[V_1] = v \text{ és } t[V_2] = w\} .$$

Renaming: Attribute renaming is nothing else, but an injective mapping from a finite set of attributes U into the set of all attributes. Attribute renaming f can be given by the list of pairs $(A, f(A))$, where $A \neq f(A)$, which is written usually

[1] The relational algebra* is the ***monotone*** part of the (full) relational algebra introduced later.

in the form $A_1A_2\dots A_n \ \rightarrow \ B_1B_2\dots B_n$. The ***renaming operator*** δ_f maps from inputs over U to outputs over$f[U]$. If R is a relation over U, then

$$\delta_f(R) = \{v \text{ over } f[U] | \exists u \in R, v(f(A)) = u(A), \ \forall A \in U\} \ .$$

Relational algebra* queries are obtained by finitely many applications of the operations above from ***relational algebra base queries,*** which are

Input relation: R.

Single constant: $\{\langle A : a\rangle\}$, where a is a constant, A is an attribute name.

Example 19.3 *Relational algebra* query.* The question 19.2. of the introduction can be expressed with relational algebra operations as follows.

$$\pi_{Theater,Address}\left((\sigma_{Director=\text{“Akira Kurosawa”}}(Film) \bowtie Show) \bowtie Theater\right) \ .$$

The mapping given by a relational algebra* query can be easily defined via induction on the operation tree. It is easy to see (Exercise 19.1-2) that non-satisfiable queries can be given using relational algebra*. There exist no rule based or tableau query equivalent with such a non-satisfiable query. Nevertheless, the following is true.

Theorem 19.6 *Rule based queries, tableau queries and satisfiable relational algebra* are equivalent query languages.*

The proof of Theorem 19.6 consists of three main parts:

1. Rule based $\equiv$ Tableau
2. Satisfiable relational algebra* $\sqsubseteq$ Rule based
3. Rule based $\sqsubseteq$ Satisfiable relational algebra*

The first (easy) step is left to the Reader (Exercise 19.1-3). For the second step, it has to be seen first, that the language of rule based queries is closed under composition. More precisely, let $\mathbf{R} = \{R_1, R_2, \dots, R_n\}$ be a database, q be a query over $\mathbf{R}$. If the output relation of q is S_1, then in a subsequent query S_1 can be used in the same way as any extensional relation of $\mathbf{R}$. Thus relation S_2 can be defined, then with its help relation S_3 can be defined, and so on. Relations S_i are ***intensional*** relations. The ***conjunctive query program*** P is a list of rules

$$\begin{array}{rcl} S_1(u_1) & \leftarrow & body_1 \\ S_2(u_2) & \leftarrow & body_2 \\ & \vdots & \\ S_m(u_m) & \leftarrow & body_m \ , \end{array} \tag{19.5}$$

where S_i's are pairwise distinct and not contained in $\mathbf{R}$. In rule body $body_i$ only relations $R_1, R_2, \dots R_n$ and $S_1, S_2, \dots, S_{i-1}$ can occur. S_m is considered to be the output relation of P, its evaluation is is done by computing the results of the rules one-by-one in order. It is not hard to see that with appropriate renaming the variables P can be substituted by a single rule, as it is shown in the following example.

Example 19.4 *Conjunctive query program.* Let $\mathbf{R} = \{Q, R\}$, and consider the following conjunctive query program

$$\begin{array}{rcl} S_1(x,z) & \leftarrow & Q(x,y), R(y,z,w) \\ S_2(x,y,z) & \leftarrow & S_1(x,w), R(w,y,v), S_1(v,z) \\ S_3(x,z) & \leftarrow & S_2(x,u,v), Q(v,z)\ . \end{array} \tag{19.6}$$

S_2 can be written using Q and R only by the first two rules of (19.6)

$$S_2(x,y,z) \leftarrow Q(x,y_1), R(y_1,w,w_1), R(w,y,v), Q(v,y_2), R(y_2,z,w_2)\ . \tag{19.7}$$

It is apparent that some variables had to be renamed to avoid unwanted interaction of rule bodies. Substituting expression (19.7) into the third rule of (19.6) in place of S_2, and appropriately renaming the variables

$$S_3(x,z) \leftarrow Q(x,y_1), R(y_1,w,w_1), R(w,u,v_1), Q(v_1,y_2), R(y_2,v,w_2), Q(v,z). \tag{19.8}$$

is obtained.

Thus it is enough to realise each single relational algebra* operation by an appropriate rule.

$P \bowtie Q$: Let $\overrightarrow{x}$ denote the list of variables (and constants) corresponding to the common attributes of P and Q, let $\overrightarrow{y}$ denote the variables (and constants) corresponding to the attributes occurring only in P, while $\overrightarrow{z}$ denotes those of corresponding to Q's own attributes. Then rule $ans(\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{z}) \leftarrow P(\overrightarrow{x}, \overrightarrow{y}), Q(\overrightarrow{x}, \overrightarrow{z})$ gives exactly relation $P \bowtie Q$.

$\sigma_F(R)$: Assume that $R = R(A_1, A_2, \ldots, A_n)$ and the selection condition F is of form either $A_i = a$ or $A_i = A_j$, where A_i, A_j are attributes a is constant. Then

$$ans(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n) \leftarrow R(x_1, \ldots, x_{i-1}, a, x_{i+1}, \ldots, x_n)\ ,$$

respectively,

$$ans(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_{j-1}, y, x_{j+1}, \ldots, x_n) \leftarrow R(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_{j-1}, y, x_{j+1}, \ldots, x_n)$$

are the rules sought. The satisfiability of relational algebra* query is used here. Indeed, during composition of operations we never obtain an expression where two distinct constants should be equated.

$\pi_{A_{i_1}, A_{i_2}, \ldots, A_{i_m}}(R)$: If $R = R(A_1, A_2, \ldots, A_n)$, then

$$ans(x_{i_1}, x_{i_2}, \ldots, x_{i_m}) \leftarrow R(x_1, x_2, \ldots, x_n)$$

works.

$A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_n$: The renaming operation of relational algebra* can be achieved by renaming the appropriate variables, as it was shown in Example 19.4.

For the proof of the third step let us consider rule

$$ans(\overrightarrow{x}) \leftarrow R_1(\overrightarrow{x_1}), R_2(\overrightarrow{x_2}), \ldots, R_n(\overrightarrow{x_n}) . \tag{19.9}$$

By renaming the attributes of relations R_i's, we may assume without loss of generality that all attribute names are distinct. Then $R = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$ can be constructed that is really a direct product, since the the attribute names are distinct. The constants and multiple occurrences of variables of rule (19.9) can be simulated by appropriate selection operators. The final result is obtained by projecting to the set of attributes corresponding to the variables of relation *ans*.

19.1.2. Extensions

Conjunctive queries are a class of query languages that has many good properties. However, the set of expressible questions are rather narrow. Consider the following.

19.4. List those pairs where one member directed the other member in a film, and vice versa, the other member also directed the first in a film.

19.5. Which theatres show "La Dolce Vita" or "Rashomon"?

19.6. Which are those films of Hitchcock that Hitchcock did not play a part in?

19.7. List those films whose every actor played in some film of Fellini.

19.8. Let us recall the game "Chain-of-Actors". The first player names an actor/actress, the next another one who played in some film together with the first named. This is continued like that, always a new actor/actress has to be named who played together with the previous one. The winner is that player who could continue the chain last time. List those actors/actresses who could be reached by "Chain-of-Actors" starting with "Marcello Mastroianni".

Equality atoms

Question **19.4.** can be easily answered if equalities are also allowed rule bodies, besides relational atoms:

$$ans(y_1, y_2) \leftarrow Film(x_1, y_1, z_1), Film(x_2, y_2, z_2), y_1 = z_2, y_2 = z_1. \tag{19.10}$$

Allowing equalities raises two problems. First, the result of the query could become infinite. For example, the rule based query

$$ans(x, y) \leftarrow R(x), y = z \tag{19.11}$$

results in an infinite number of tuples, since variables y and z are not bounded by relation R, thus there can be an infinite number of evaluations that satisfy the rule body. Hence, the concept of ***domain restricted*** query is introduced. Rule based query q is ***domain restricted,*** if all variables that occur in the rule body also occur in some relational atom.

The second problem is that equality atoms may cause the body of a rule become

unsatisfiable, in contrast to Proposition 19.4. For example, query

$$ans(x) \leftarrow R(x), x = a, x = b \tag{19.12}$$

is domain restricted, however if a and b are distinct constants, then the answer will be empty. It is easy to check whether a rule based query with equality atoms is satisfiable.

SATISFIABLE(q)

1 Compute the transitive closure of equalities of the body of q.
2 **if** Two distinct constants should be equal by transitivity
3 **then return** "Not satisfiable."
4 **else return** "Satisfiable."

It is also true (Exercise 19.1-4) that if a rule based query q that contains equality atoms is satisfiable, then there exists a another rule based query q' without equalities that is equivalent with q.

Disjunction – union

The question **19.5**. cannot be expressed with conjunctive queries. However, if the union operator is added to relational algebra, then **19.5**. can be expressed in that extended relational algebra:

$$\pi_{Theater}(\sigma_{Title=\text{"La Dolce Vita"}}(Show) \cup \sigma_{Title=\text{" Rashomon"}}(Show)) \ . \tag{19.13}$$

Rule based queries are also capable of expressing question **19.5**. if it is allowed that the same relation is in the head of many distinct rules:

$$\begin{aligned} ans(x_M) &\leftarrow Show(x_{Th}, \text{"La Dolce Vita"}, x_{Ti}) \ , \\ ans(x_M) &\leftarrow Show(x_{Th}, \text{"Rashomon"}, x_{Ti}) \ . \end{aligned} \tag{19.14}$$

Non-recursive datalog program is a generalisation of this.

Definition 19.7 *A **non-recursive datalog program** over schema* $\mathbf{R}$ *is a set of rules*

$$\begin{aligned} S_1(u_1) &\leftarrow body_1 \\ S_2(u_2) &\leftarrow body_2 \\ &\vdots \\ S_m(u_m) &\leftarrow body_m \ , \end{aligned} \tag{19.15}$$

where no relation of $\mathbf{R}$ *occurs in a head, the same relation may occur in the head of several rules, furthermore there exists an ordering* $r_1, r_2, \ldots, r_m$ *of the rules such that the relation in the head of* r_i *does not occur in the body of any rule* r_j *for*$j \leq i$.

The semantics of the non-recursive datalog program (19.15) is similar to the conjunctive query program (19.5). The rules are evaluated in the order $r_1, r_2, \ldots, r_m$ of Definition 19.7, and if a relation occurs in more than one head then the union of the sets of tuples given by those rules is taken.

The union of tableau queries $(\mathbf{T}_i, u)$ $i = 1, 2, \ldots, n$ is denoted by $(\{\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_n\}, u)$. It is evaluated by individually computing the result of each tableau query $(\mathbf{T}_i, u)$, then the union of them is taken. The following holds.

Theorem 19.8 *The language of non-recursive datalog programs with unique output relation and the relational algebra extended with the union operator are equivalent.*

The proof of Theorem 19.8 is similar to that of Theorem 19.6 and it is left to the Reader (Exercise 19.1-5). Let us note that the expressive power of the union of tableau queries is weaker. This is caused by the requirement having the same summary row for each tableau. For example, the non-recursive datalog program query

$$\begin{array}{l} ans(a) \leftarrow \\ ans(b) \leftarrow \end{array} \tag{19.16}$$

cannot be realised as union of tableau queries.

Negation

The query **19.6**. is obviously not monotone. Indeed, suppose that in relation *Film* there exist tuples about Hitchcock's film *Psycho*, for example ("Psycho","A. Hitchcock","A. Perkins"), ("Psycho","A. Hitchcock","J. Leigh"), ... , however, the tuple ("Psycho","A. Hitchcock","A. Hitchcock") is not included. Then the tuple ("Psycho") occurs in the output of query **19.6**. With some effort one can realize however, that Hitchcock appears in the film *Psycho*, as "a man in cowboy hat". If the tuple ("Psycho","A. Hitchcock","A. Hitchcock") is added to relation *Film* as a consequence, then the instance of schema **CinePest** gets larger, but the output of query **19.6**. becomes smaller.

It is not too hard to see that the query languages discussed so far are monotone, hence query **19.6**. cannot be formulated with non-recursive datalog program or with some of its equivalents. Nevertheless, if the difference $(-)$operator is also added to relation algebra, then it becomes capable of expressing queries of type **19.6**. For example,

$$\pi_{Title}(\sigma_{Director="A.\ Hitchcock"}(Film)) - \pi_{Title}(\sigma_{Actor="A.\ Hitchcock"}(Film)) \tag{19.17}$$

realises exactly query **19.6**. Hence, the (full) relational algebra consists of operations $\{\sigma, \pi, \bowtie, \delta, \cup, -\}$. The importance of the relational algebra is shown by the fact, that Codd calls a query language $\mathcal{Q}$ ***relationally complete,*** exactly if for all relational algebra query q there exists $q' \in \mathcal{Q}$, such that $q \equiv q'$.

If ***negative literals***, that is atoms of the form $\neg R(u)$ are also allowed in rule bodies, then the obtained ***non-recursive datalog with negation,*** in notation ***nr-datalog***$^\neg$ is relationally complete.

Definition 19.9 *A non-recursive datalog*$^\neg$ *(nr-datalog*$^\neg$*) rule is of form*

$$q: \quad S(u) \leftarrow L_1, L_2, \ldots, L_n \ , \tag{19.18}$$

where S is a relation, u is a free tuple, L_i's are ***literals,****, that is expression of form $R(v)$ or $\neg R(v)$, such that v is a free tuple for $i = 1, 2, \ldots, n$. S does not occur in the body of the rule. The rule is* ***domain restricted,*** *if each variable x that occurs in the rule also occurs in a* ***positive literal*** *(expression of the form $R(v)$) of the body. Every nr-datalog*$^\neg$ *rule is considered domain restricted, unless it is specified otherwise.*

The semantics of rule (19.18) is as follows. Let $\mathbf{R}$ be a relational schema that contains all relations occurring in the body of q, furthermore, let $\mathcal{I}$ be an instance over $\mathbf{R}$. The ***image of*** $\mathcal{I}$ ***under*** q is

$$q(\mathcal{I}) = \{\nu(u) |\ \nu \text{ is an valuation of the variables and for } i = 1, 2, \ldots, n \quad \nu(u_i) \in \mathcal{I}(R_i), \text{ if } L_i = R_i(u_i) \text{ and} \quad \nu(u_i) \notin \mathcal{I}(R_i), \text{ if } L_i = \neg R_i(u_i)\}. \tag{19.19}$$

A ***nr-datalog$^\neg$ program*** over schema $\mathbf{R}$ is a collection of nr-datalog$^\neg$ rules

$$\begin{array}{rcl} S_1(u_1) & \leftarrow & body_1 \\ S_2(u_2) & \leftarrow & body_2 \\ & \vdots & \\ S_m(u_m) & \leftarrow & body_m\ , \end{array} \tag{19.20}$$

where relations of schema $\mathbf{R}$ do not occur in heads of rules, the same relation may appear in more than one rule head, furthermore there exists an ordering $r_1, r_2, \ldots, r_m$ of the rules such that the relation of the head of rule r_i does not occur in the head of any rule r_j if $j \leq i$.

The computation of the result of nr-datalog$^\neg$ program (19.20) applied to instance $\mathcal{I}$ over schema $\mathbf{R}$ can be done in the same way as the evaluation of non-recursive datalog program (19.15), with the difference that the individual nr-datalog$^\neg$ rules should be interpreted according to (19.19).

Example 19.5 *Nr-datalog$^\neg$ program.* Let us assume that all films that are included in relation *Film* have only one director. (It is not always true in real life!) The nr-datalog$^\neg$ rule

$$ans(x) \leftarrow Film(x, \text{“A. Hitchcock”}, z), \neg Film(x, \text{“A. Hitchcock”}, \text{“A. Hitchcock”}) \tag{19.21}$$

expresses query **19.6**. Query **19.7.** is realised by the nr-datalog$^\neg$ program

$$\begin{array}{rcl} Fellini\text{-}actor(z) & \leftarrow & Film(x, \text{”F. Fellini”}, z) \\ Not\text{-}the\text{-}answer(x) & \leftarrow & Film(x, y, z), \neg Fellini\text{-}actor(z) \\ Answer(x) & \leftarrow & Film(x, y, z), \neg Not\text{-}the\text{-}answer(x)\ . \end{array} \tag{19.22}$$

One has to be careful in writing nr-datalog$^\neg$ programs. If the first two rules of program (19.22) were to be merged like in Example 19.4

$$\begin{array}{rcl} Bad\text{-}not\text{-}ans(x) & \leftarrow & Film(x, y, z), \neg Film(x', \text{”F. Fellini”}, z), Film(x', \text{”F. Fellini”}, z') \\ Answer(x) & \leftarrow & Film(x, y, z), \neg Bad\text{-}not\text{-}ans(x), \end{array} \tag{19.23}$$

then (19.23) answers the following query (assuming that all films have unique director)

19.9. List all those films whose every actor played in ***each*** film of Fellini,

instead of query **19.7.**

It is easy to see that every satisfiable nr-datalog$^\neg$ program that contains equality atoms can be replaced by one without equalities. Furthermore the following proposition is true, as well.

Proposition 19.10 *The satisfiable (full) relational algebra and the nr-datalog$^\neg$ programs with single output relation are equivalent query languages.*

Recursion

Query **19.8.** cannot be formulated using the query languages introduced so far. Some ***a priori*** information would be needed about how long a *chain-of-actors* could be formed starting with a given actor/actress. Let us assume that the maximum length of a chain-of-actors starting from "Marcello Mastroianni" is 117. (It would be interesting to know the ***real*** value!) Then, the following non-recursive datalog program gives the answer.

$$\begin{array}{rcl}
\mathit{Film\text{-}partner}(z_1, z_2) & \leftarrow & \mathit{Film}(x, y, z_1), \mathit{Film}(x, y, z_2), z_1 < z_2 \text{[2]} \\
\mathit{Partial\text{-}answer}_1(z) & \leftarrow & \mathit{Film\text{-}partner}(z, \text{“Marcello Mastroianni”}) \\
\mathit{Partial\text{-}answer}_1(z) & \leftarrow & \mathit{Film\text{-}partner}(\text{“Marcello Mastroianni”}, z) \\
\mathit{Partial\text{-}answer}_2(z) & \leftarrow & \mathit{Film\text{-}partner}(z, y), \mathit{Partial\text{-}answer}_1(y) \\
\mathit{Partial\text{-}answer}_2(z) & \leftarrow & \mathit{Film\text{-}partner}(y, z), \mathit{Partial\text{-}answer}_1(y) \\
\vdots & & \vdots \\
\mathit{Partial\text{-}answer}_{117}(z) & \leftarrow & \mathit{Film\text{-}partner}(z, y), \mathit{Partial\text{-}answer}_{116}(y) \\
\mathit{Partial\text{-}answer}_{117}(z) & \leftarrow & \mathit{Film\text{-}partner}(y, z), \mathit{Partial\text{-}answer}_{116}(y) \\
\mathit{Mastroianni\text{-}chain}(z) & \leftarrow & \mathit{Partial\text{-}answer}_1(z) \\
\mathit{Mastroianni\text{-}chain}(z) & \leftarrow & \mathit{Partial\text{-}answer}_2(z) \\
\vdots & & \vdots \\
\mathit{Mastroianni\text{-}chain}(z) & \leftarrow & \mathit{Partial\text{-}answer}_{117}(z)
\end{array} \tag{19.24}$$

It is much easier to express query **19.8.** using ***recursion***. In fact, the ***transitive closure*** of the graph *Film-partner* needs to be calculated. For the sake of simplicity the definition of *Film-partner* is changed a little (thus approximately doubling the storage requirement).

$$\begin{array}{rcl}
\mathit{Film\text{-}partner}(z_1, z_2) & \leftarrow & \mathit{Film}(x, y, z_1), \mathit{Film}(x, y, z_2) \\
\mathit{Chain\text{-}partner}(x, y) & \leftarrow & \mathit{Film\text{-}partner}(x, y) \\
\mathit{Chain\text{-}partner}(x, y) & \leftarrow & \mathit{Film\text{-}partner}(x, z), \mathit{Chain\text{-}partner}(z, y)\ .
\end{array} \tag{19.25}$$

The datalog program (19.25) is ***recursive,*** since the definition of relation *Chain-partner* uses the relation itself. Let us suppose for a moment that this is meaningful, then query **19.8.** is answered by rule

$$\mathit{Mastroianni\text{-}chain}(x) \leftarrow \mathit{Chain\text{-}partner}(x, \text{“Marcello Mastroianni”}) \tag{19.26}$$

Definition 19.11 *The expression*

$$R_1(u_1) \leftarrow R_2(u_2), R_3(u_3), \ldots, R_n(u_n) \tag{19.27}$$

is a ***datalog rule,*** *if $n \geq 1$, the R_i's are relation names, the u_i's are free tuples of appropriate length. Every variable of u_1 has to occur in one of $u_2, \ldots u_n$, as well. The head of the rule is $R_1(u_1)$, the body of the rule is $R_2(u_2), R_3(u_3), \ldots, R_n(u_n)$. A*

[2] Arbitrary comparison atoms can be used, as well, similarly to equality atoms. Here $z_1 < z_2$ makes it sure that all pairs occur at most once in the list.

datalog program *is a finite collection of rules of type (19.27). Let P be a datalog program. The relation R occurring in P is* ***extensional*** *if it occurs in only rule bodies, and it is* ***intensional*** *if it occurs in the head of some rule.*

If ν is a valuation of the variables of rule (19.27), then $R_1(\nu(u_1)) \leftarrow R_2(\nu(u_2)), R_3(\nu(u_3)), \ldots, R_n(\nu(u_n))$ is a ***realisation*** of rule (19.27). The ***extensional (database) schema*** of P consists of the extensional relations of P, in notation $edb(P)$. The ***intensional schema*** of P, in notation $idb(P)$ is defined similarly as consisting of the intensional relations of P. Let $sch(P) = edb(P) \cup idb(P)$. The semantics of datalog program P is a mapping from the set of instances over $edb(P)$ to the set of instances over $idb(P)$. This can be defined proof theoretically, model theoretically or as a fixpoint of some operator. This latter one is equivalent with the first two, so to save space only the fixpoint theoretical definition is discussed.

There are no negative literals used in Definition 19.11. The main reason of this is that recursion and negation together may be meaningless, or contradictory. Nevertheless, sometimes negative atoms might be necessary. In those cases the semantics of the program will be defined specially.

Fixpoint semantics

Let P be a datalog program, $\mathcal{K}$ be an instance over $sch(P)$. ***Fact*** A, that is a tuple consisting of constants is an ***immediate consequence*** of $\mathcal{K}$ and P, if either $A \in \mathcal{K}(R)$ for some relation $R \in sch(P)$, or $A \leftarrow A_1, A_2, \ldots, A_n$ is a realisation of a rule in P and each A_i is in $\mathcal{K}$. The ***immediate consequence operator*** T_P is a mapping from the set of instances over $sch(P)$ to itself. $T_P(\mathcal{K})$ consists of all immediate consequences of $\mathcal{K}$ and P.

Proposition 19.12 *The immediate consequence operator T_P is monotone.*

Proof Let $\mathcal{I}$ and $\mathcal{J}$ be instances over $sch(P)$, such that $\mathcal{I} \subseteq \mathcal{J}$. Let A be a fact of$T_P(\mathcal{I})$. If $A \in \mathcal{I}(R)$ for some relation $R \in sch(P)$, then $A \in \mathcal{J}(R)$ is implied by $\mathcal{I} \subseteq \mathcal{J}$. on the other hand, if $A \leftarrow A_1, A_2, \ldots, A_n$ is a realisation of a rule in P and each A_i is in $\mathcal{I}$, then $A_i \in \mathcal{J}$ also holds. ■

The definition of T_P implies that $\mathcal{K} \subseteq T_P(\mathcal{K})$. Using Proposition 19.12 it follows that

$$\mathcal{K} \subseteq T_P(\mathcal{K}) \subseteq T_P(T_P(\mathcal{K})) \subseteq \ldots \ . \tag{19.28}$$

Theorem 19.13 *For every instance $\mathcal{I}$ over schema $sch(P)$ there exists a unique minimal instance $\mathcal{I} \subseteq \mathcal{K}$ that is a* ***fixpoint*** *of T_P, i.e. $\mathcal{K} = T_P(\mathcal{K})$.*

Proof Let $T_P^i(\mathcal{I})$ denote the consecutive application of operator T_P i-times, and let

$$\mathcal{K} = \bigcup_{i=0}^{\infty} T_P^i(\mathcal{I}) \ . \tag{19.29}$$

By the monotonicity of T_P and (19.29) we have

$$T_P(\mathcal{K}) = \bigcup_{i=1}^{\infty} T_P^i(\mathcal{I}) \subseteq \bigcup_{i=0}^{\infty} T_P^i(\mathcal{I}) = \mathcal{K} \subseteq T_P(\mathcal{K}) \ , \tag{19.30}$$

that is $\mathcal{K}$ is a fixpoint. It is easy to see that every fixpoint that contains $\mathcal{I}$, also contains $T_P^i(\mathcal{I})$ for all $i = 1, 2, \ldots$, that is it contains $\mathcal{K}$, as well. ■

Definition 19.14 *The **result** of datalog program P on instance $\mathcal{I}$ over $edb(P)$ is the unique minimal fixpoint of T_P containing $\mathcal{I}$, in notation $P(\mathcal{I})$.*

It can be seen, see Exercise 19.1-6, that the chain in (19.28) is finite, that is there exists an n, such that $T_P(T_P^n(\mathcal{I})) = T_P^n(\mathcal{I})$. The naive evaluation of the result of the datalog program is based on this observation.

NAIV-DATALOG(P,$\mathcal{I}$)

1 $\mathcal{K} \leftarrow \mathcal{I}$
2 **while** $T_P(\mathcal{K}) \neq \mathcal{K}$
3 **do** $\mathcal{K} \leftarrow T_P(\mathcal{K})$
4 **return** $\mathcal{K}$

Procedure NAIV-DATALOG is not optimal, of course, since every fact that becomes included in $\mathcal{K}$ is calculated again and again at every further execution of the **while** loop.

The idea of SEMI-NAIV-DATALOG is that it tries to use only recently calculated new facts in the **while** loop, as much as it is possible, thus avoiding recomputation of known facts. Consider datalog program P with $edb(P) = \mathbf{R}$, and $idb(P) = \mathbf{T}$. For a rule

$$S(u) \leftarrow R_1(v_1), \ldots, R_n(v_n), T_1(w_1), \ldots, T_m(w_m) \tag{19.31}$$

of P where $R_k \in \mathbf{R}$ and $T_j \in \mathbf{T}$, the following rules are constructed for $j = 1, 2, \ldots, m$ and $i \geq 1$

$$\begin{aligned} temp_S^{i+1}(u) \leftarrow\ & R_1(v_1), \ldots, R_n(v_n), \\ & T_1^i(w_1), \ldots, T_{j-1}^i(w_{j-1}), \Delta_{T_j}^i(w_j), T_{j+1}^{i-1}(w_{j+1}), \ldots, T_m^{i-1}(w_m)\ . \end{aligned} \tag{19.32}$$

Relation $\Delta_{T_j}^i$ denotes the change of T_j in iteration i. The union of rules corresponding to S in layer i is denoted by P_S^i, that is rules of form (19.32) for $temp_S^{i+1}$, $j = 1, 2, \ldots, m$. Assume that the list of *idb* relations occurring in rules defining the *idb* relation S is $T_1, T_2, \ldots, T_\ell$. Let

$$P_S^i(\mathcal{I}, T_1^{i-1}, \ldots, T_\ell^{i-1}, T_1^i, \ldots, T_\ell^i, \Delta_{T_1}^i, \ldots, \Delta_{T_\ell}^i) \tag{19.33}$$

denote the set of facts (tuples) obtained by applying rules (19.32) to input instance $\mathcal{I}$ and to *idb* relations $T_j^{i-1}, T_j^i, \Delta_{T_j}^i$. The input instance $\mathcal{I}$ is the actual value of the *edb* relations of P.

SEMI-NAIV-DATALOG(P,$\mathcal{I}$)

```
1  P' ← those rules of P whose body does not contain idb relation
2  for S ∈ idb(P)
3      do S^0 ← ∅
4         Δ^1_S ← P'(I)(S)
5  i ← 1
6  repeat
7          for S ∈ idb(P)
8                              ▷ T_1, ..., T_ℓ are the idb relations of the rules defining S.
9              do S^i ← S^{i-1} ∪ Δ^i_S
10                Δ^{i+1}_S ← P^i_S(I, T_1^{i-1}, ..., T_ℓ^{i-1}, T_1^i, ..., T_ℓ^i, Δ^i_{T_1}, ..., Δ^i_{T_ℓ}) − S^i
11         i ← i + 1
12 until Δ^i_S = ∅ for all S ∈ idb(P)
13 for S ∈ idb(P)
14     do S ← S^i
15 return S
```

Theorem 19.15 *Procedure* SEMI-NAIV-DATALOG *correctly computes the result of program P on instance $\mathcal{I}$.*

Proof We will show by induction on i that after execution of the loop of lines 6–12 i^{th} times the value of S^i is $T_P^i(\mathcal{I})(S)$, while Δ_S^{i+1} is equal to $T_P^{i+1}(\mathcal{I})(S) - T_P^i(\mathcal{I})(S)$ for arbitrary $S \in idb(P)$. $T_P^i(\mathcal{I})(S)$ is the result obtained for S starting from $\mathcal{I}$ and applying the immediate consequence operator T_P i-times.

For $i = 0$, line 4 calculates exactly $T_P(\mathcal{I})(S)$ for all $S \in idb(P)$. In order to prove the induction step, one only needs to see that $P_S^i(\mathcal{I}, T_1^{i-1}, \ldots, T_\ell^{i-1}, T_1^i, \ldots, T_\ell^i, \Delta_{T_1}^i, \ldots, \Delta_{T_\ell}^i) \cup S^i$ is exactly equal to $T_P^{i+1}(\mathcal{I})(S)$, since in lines 9–10 procedure SEMI-NAIV-DATALOG constructs S^i-t and Δ_S^{i+1} using that. The value of S^i is $T_P^i(\mathcal{I})(S)$, by the induction hypothesis. Additional new tuples are obtained only if that for some *idb* relation defining S such tuples are considered that are constructed at the last application of T_P, and these are in relations $\Delta_{T_1}^i, \ldots, \Delta_{T_\ell}^i$, also by the induction hypothesis.

The halting condition of line 12 means exactly that all relations $S \in idb(P)$ are unchanged during the application of the immediate consequence operator T_P, thus the algorithm has found its minimal fixpoint. This latter one is exactly the result of datalog program P on input instance $\mathcal{I}$ according to Definition 19.14. ■

Procedure SEMI-NAIV-DATALOG eliminates a large amount of unnecessary calculations, nevertheless it is not optimal on some datalog programs (Exercise gy:snaiv). However, analysis of the datalog program and computation based on that can save most of the unnecessary calculations.

Definition 19.16 *Let P be a datalog program. The* ***precedence graph*** *of P is the directed graph G_P defined as follows. Its vertex set consists of the relations of $idb(P)$, and (R, R') is an arc for $R, R' \in idb(P)$ if there exists a rule in P whose head is R' and whose body contains R. P is* ***recursive,*** *if G_P contains a directed cycle. Relations R and R' are* ***mutually recursive*** *if the belong to the same strongly*

connected component of G_P.

Being mutually recursive is an equivalence relation on the set of relations $idb(P)$. The main idea of procedure IMPROVED-SEMI-NAIV-DATALOG is that for a relation $R \in idb(P)$ only those relations have to be computed "simultaneously" with R that are in its equivalence class, all other relations defining R can be calculated "in advance" and can be considered as *edb* relations.

IMPROVED-SEMI-NAIV-DATALOG(P,$\mathcal{I}$)

```
1  Determine the equivalence classes of idb(P) under mutual recursivity.
2  List the equivalence classes [R_1], [R_2], ..., [R_n]
       according to a topological order of G_P.
3                ▷ There exists no directed path from R_j to R_i in G_P for all i < j.
4  for i ← 1 to n
5      do Use SEMI-NAIV-DATALOG to compute relations of [R_i]
          taking relations of [R_j] as edb relations for j < i.
```

Lines 1–2 can be executed in time $O(v_{G_P} + e_{G_P})$ using depth first search, where v_{G_P} and e_{G_P} denote the number of vertices and edges of graph G_P, respectively. Proof of correctness of the procedure is left to the Reader (Exercise 19.1-8).

19.1.3. Complexity of query containment

In the present section we return to conjunctive queries. The costliest task in computing result of a query is to generate the natural join of relations. In particular, if there are no indexes available on the common attributes, hence only procedure FULL-TUPLEWISE-JOIN is applicable.

FULL-TUPLEWISE-JOIN(R_1, R_2)

```
1  S ← ∅
2  for all u ∈ R_1
3      do for all v ∈ R_2
4             do if u and v can be joined
5                    then S ← S ∪ {u ⋈ v}
6  return S
```

It is clear that the running time of FULL-TUPLEWISE-JOIN is $O(|R_1| \times |R_2|)$. Thus, it is important that in what order is a query executed, since during computation natural joins of relations of various sizes must be calculated. In case of tableau queries the ***Homomorphism Theorem*** gives a possibility of a query rewriting that uses less joins than the original.

Let q_1, q_2 be queries over schema $\mathbf{R}$. q_2 ***contains*** q_1, in notation $q_1 \sqsubseteq q_2$, if for all instances $\mathcal{I}$ over schema $\mathbf{R}$ $q_1(\mathcal{I}) \subseteq q_2(\mathcal{I})$ holds. $q_1 \equiv q_2$ according to Definition 19.1 iff $q_1 \sqsubseteq q_2$ and $q_1 \sqsupseteq q_2$. A generalisation of valuations will be needed. ***Substitution*** is a mapping from the set of variables to the union of sets of variables and sets of constants that is extended to constants as identity. Extension of substitution to free tuples and tableaux can be defined naturally.

Definition 19.17 *Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be two tableau queries overs schema $\mathbf{R}$. Substitution θ is a* ***homomorphism*** *from q' to q, if $\theta(\mathbf{T}') = \mathbf{T}$ and $\theta(u') = u$.*

Theorem 19.18 (Homomorphism Theorem). *Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be two tableau queries overs schema $\mathbf{R}$. $q \sqsubseteq q'$ if and only if, there exists a homomorphism from q' to q.*

Proof Assume first, that θ is a homomorphism from q' to q, and let $\mathcal{I}$ be an instance over schema $\mathbf{R}$. Let $w \in q(\mathcal{I})$. This holds exactly if there exists a valuation ν that maps tableau $\mathbf{T}$ into $\mathcal{I}$ and $\nu(u) = w$. It is easy to see that $\theta \circ \nu$ maps tableau $\mathbf{T}'$ into $\mathcal{I}$ and $\theta \circ \nu(u') = w$, that is $w \in q'(\mathcal{I})$. Hence, $w \in q(\mathcal{I}) \Longrightarrow w \in q'(\mathcal{I})$, which in turn is equivalent with $q \sqsubseteq q'$.

On the other hand, let us assume that $q \sqsubseteq q'$. The idea of the proof is that both, q and q' are applied to the "instance" $\mathbf{T}$. The output of q is free tuple u, hence the output of q' also contains u, that is there exists a θ embedding of $\mathbf{T}'$ into $\mathbf{T}$ that maps u' to u. To formalise this argument the instance $\mathcal{I}_{\mathbf{T}}$ isomorphic to $\mathbf{T}$ is constructed.

Let V be the set of variables occurring in $\mathbf{T}$. For all $x \in V$ let a_x be constant that differs from all constants appearing in $\mathbf{T}$ or $\mathbf{T}'$, furthermore $x \neq x' \Longrightarrow a_x \neq a_{x'}$. Let μ be the valuation that maps $x \in V$ to a_x, furthermore let $\mathcal{I}_{\mathbf{T}} = \mu(\mathbf{T})$. μ is a bijection from V to $\mu(V)$ and there are no constants of $\mathbf{T}$ appearing in $\mu(V)$, hence μ^{-1} well defined on the constants occurring in $\mathcal{I}_{\mathbf{T}}$.

It is clear that $\mu(u) \in q(\mathcal{I}_{\mathbf{T}})$, thus using $q \sqsubseteq q'$ $\mu(u) \in q'(\mathcal{I}_{\mathbf{T}})$ is obtained. That is, there exists a valuation ν that embeds tableau $\mathbf{T}'$ into $\mathcal{I}_{\mathbf{T}}$, such that $\nu(u') = \mu(u)$. It is not hard to see that $\nu \circ \mu^{-1}$ is a homomorphism from q' to q. ■

Query optimisation by tableau minimisation

According to Theorem 19.6 tableau queries and satisfiable relational algebra (without subtraction) are equivalent. The proof shows that the relational algebra expression equivalent with a tableau query is of form $\pi_{\overline{ae}}(\sigma_F(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k))$, where k is the number of rows of the tableau. It implies that if the number of joins is to be minimised, then the number of rows of an equivalent tableau must be minimised.

The tableau query $(\mathbf{T}, u)$ is ***minimal,*** if there exists no tableau query $(\mathbf{S}, v)$ that is equivalent with $(\mathbf{T}, u)$ and $|\mathbf{S}| < |\mathbf{T}|$, that is S has fewer rows. It may be surprising, but it is true, that a minimal tableau query equivalent with $(\mathbf{T}, u)$ can be obtained by simply dropping some rows from $\mathbf{T}$.

Theorem 19.19 *Let $q = (\mathbf{T}, u)$ be a tableau query. There exists a subset $\mathbf{T}'$ of $\mathbf{T}$, such that query $q' = (\mathbf{T}', u)$ is minimal and equivalent with $q = (\mathbf{T}, u)$.*

Proof Let $(\mathbf{S}, v)$ be a minimal query equivalent with q. According to the Homomorphism Theorem there exist homomorphisms θ from q to $(\mathbf{S}, v)$, and λ from $(\mathbf{S}, v)$ to q. Let $\mathbf{T}' = \theta \circ \lambda(\mathbf{T})$. It is easy to check that $(\mathbf{T}', u) \equiv q$ and $|\mathbf{T}'| \leq |\mathbf{S}|$. But $(\mathbf{S}, v)$ is minimal, hence $(\mathbf{T}', u)$ is minimal, as well. ■

Example 19.6 *Application of tableau minimisation* Consider the relational algebra expression

$$q = \pi_{AB}\left(\sigma_{B=5}(R)\right) \bowtie \pi_{BC}\left(\pi_{AB}(R) \bowtie \pi_{AC}\left(\sigma_{B=5}(R)\right)\right) \tag{19.34}$$

over the schema $\mathbf{R}$ of attribute set $\{A, B, C\}$. The tableau query corresponding to q is the following tableau $\mathbf{T}$:

$$\begin{array}{c|ccc} R & A & B & C \\ \hline & x & 5 & z_1 \\ & x_1 & 5 & z_2 \\ & x_1 & 5 & z \\ \hline u & x & 5 & z \end{array} \tag{19.35}$$

Such a homomorphism is sought that maps some rows of $\mathbf{T}$ to some other rows of $\mathbf{T}$, thus sort of "folding" the tableau. The first row cannot be eliminated, since the homomorphism is an identity on free tuple u, thus x must be mapped to itself. The situation is similar with the third row, as the image of z is itself under any homomorphism. However the second row can be eliminated by mapping x_1 to x and z_2 to z, respectively. Thus, the minimal tableau equivalent with $\mathbf{T}$ consists of the first and third rows of $\mathbf{T}$. Transforming back to relational algebra expression,

$$\pi_{AB}(\sigma_{B=5}(R)) \bowtie \pi_{BC}(\sigma_{B=5}(R)) \tag{19.36}$$

is obtained. Query (19.36) contains one less join operator than query (19.34).

The next theorem states that the question of tableau containment and equivalence is NP-complete, hence tableau minimisation is an NP-hard problem.

Theorem 19.20 *For given tableau queries q and q' the following decision problems are NP-complete:*

19.10. $q \sqsubseteq q'$*?*

19.11. $q \equiv q'$*?*

19.12. Assume that q' is obtained from q by removing some free tuples. Is it true then that $q \equiv q'$?

Proof The Exact-Cover problem will be reduced to the various tableau problems. The input of Exact-Cover problem is a finite set $X = \{x_1, x_2, \ldots, x_n\}$, and a collection of its subsets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$. It has to be determined whether there exists $\mathcal{S}' \sqsubseteq \mathcal{S}$, such that subsets in $\mathcal{S}'$ cover X exactly (that is, for all $x \in X$ there exists exactly one $S \in \mathcal{S}'$ such that $x \in S$). Exact-Cover is known to be an NP-complete problem.

Let $\mathcal{E} = (X, \mathcal{S})$ be an input of Exact cover. A construction is sketched that produces a pair $q_\mathcal{E}, q'_\mathcal{E}$ of tableau queries to $\mathcal{E}$ in polynomial time. This construction can be used to prove the various NP-completeness results.

Let the schema $\mathbf{R}$ consist of the pairwise distinct attributes $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$. $q_\mathcal{E} = (\mathbf{T}_\mathcal{E}, t)$ and $q'_\mathcal{E} = (\mathbf{T}'_\mathcal{E}, t)$ are tableau queries over schema $\mathbf{R}$ such that the summary row of both of them is free tuple $t = \langle A_1 : a_1, A_2 : a_2, \ldots, A_n : a_n \rangle$, where $a_1, a_2, \ldots, a_n$ are pairwise distinct variables.

Let $b_1, b_2, \ldots, b_m, c_1, c_2, \ldots c_m$ be another set of pairwise distinct variables. Tableau $\mathbf{T}_\mathcal{E}$ consists of n rows, for each element of X corresponds one. a_i stands in column of attribute A_i in the row of x_i, while b_j stands in column of attribute B_j for all such j that $x_i \in S_j$ holds. In other positions of tableau $\mathbf{T}_\mathcal{E}$ pairwise distinct

new variables stand.

Similarly, $\mathbf{T}'_{\mathcal{E}}$ consists of m rows, one corresponding to each element of $\mathcal{S}$. a_i stands in column of attribute A_i in the row of S_j for all such i that $x_i \in S_j$, furthermore $c_{j'}$ stands in the column of attribute $B_{j'}$, for all $j' \neq j$. In other positions of tableau $\mathbf{T}'_{\mathcal{E}}$ pairwise distinct new variables stand.

The NP-completeness of problem 19.10. follows from that X has an exact cover using sets of $\mathcal{S}$ if and only if $q'_{\mathcal{E}} \sqsubseteq q_{\mathcal{E}}$ holds. The proof, and the proof of the NP-completeness of problems 19.11. and 19.12. are left to the Reader (Exercise 19.1-9). ■

Exercises

19.1-1 Prove Proposition 19.4, that is every rule based query q is monotone and satisfiable. *Hint.* For the proof of satisfiability let K be the set of constants occurring in query q, and let $a \notin K$ be another constant. For every relation schema R_i in rule (19.3) construct all tuples $(a_1, a_2, \ldots, a_r)$, where $a_i \in K \cup \{a\}$, and r is the arity of R_i. Let $\mathcal{I}$ be the instance obtained so. Prove that $q(\mathcal{I})$ is nonempty.

19.1-2 Give a relational schema $\mathbf{R}$ and a relational algebra query q over schema $\mathbf{R}$, whose result is empty to arbitrary instance over $\mathbf{R}$.

19.1-3 Prove that the languages of rule based queries and tableau queries are equivalent.

19.1-4 Prove that every rule based query q with equality atoms is either equivalent with the empty query $q^{\emptyset}$, or there exists a rule based query q' without equality atoms such that $q \equiv q'$. Give a polynomial time algorithm that determines for a rule based query q with equality atoms whether $q \equiv q^{\emptyset}$ holds, and if not, then constructs a rule based query q' without equality atoms, such that $q \equiv q'$.

19.1-5 Prove Theorem 19.8 by generalising the proof idea of Theorem 19.6.

19.1-6 Let P be a datalog program, $\mathcal{I}$ be an instance over $edb(P)$, $inC(P, \mathcal{I})$ be the (finite) set of constants occurring in $\mathcal{I}$ and P. Let $\mathbf{B}(P, \mathcal{I})$ be the following instance over $sch(P)$:

1. For every relation R of $edb(P)$ the fact $R(u)$ is in $\mathbf{B}(P, \mathcal{I})$ iff it is in $\mathcal{I}$, furthermore
2. for every relation R of $idb(P)$ every $R(u)$ fact constructed from constants of $C(P, \mathcal{I})$ is in $\mathbf{B}(P, \mathcal{I})$.

Prove that

$$\mathcal{I} \subseteq T_P(\mathcal{I}) \subseteq T_P^2(\mathcal{K}) \subseteq T_P^3(\mathcal{K}) \subseteq \cdots \subseteq \mathbf{B}(P, \mathcal{I}). \tag{19.37}$$

19.1-7 Give an example of an input, that is a datalog program P and instance $\mathcal{I}$ over $edb(P)$, such that the same tuple is produced by distinct runs of the loop of Semi-naiv-datalog.

19.1-8 Prove that procedure Improved-Semi-Naiv-Datalog stops in finite time for all inputs, and gives correct result. Give an example of an input on which Improved-Semi-Naiv-Datalog calculates less number of rows multiple times than Semi-naiv-datalog.

19.1-9

1. Prove that for tableau queries $q_{\mathcal{E}} = (\mathbf{T}_{\mathcal{E}}, t)$ and $q'_{\mathcal{E}} = (\mathbf{T}'_{\mathcal{E}}, t)$ of the proof of

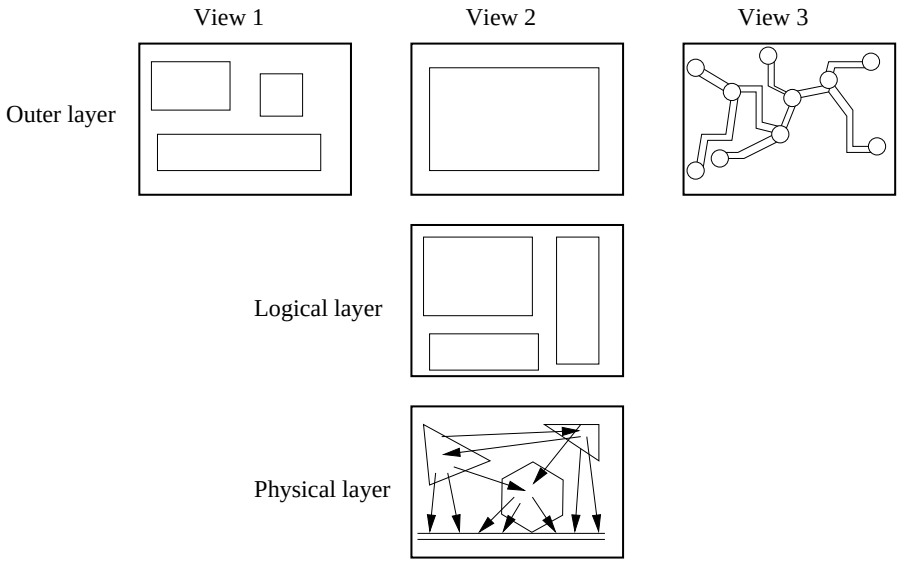


Figure 19.2. The three levels of database architecture.

Theorem 19.20 there exists a homomorphism from $(\mathbf{T}_{\mathcal{E}}, t)$ to $(\mathbf{T}'_{\mathcal{E}}, t)$ if and only if the EXACT-COVER problem $\mathcal{E} = (X, \mathcal{S})$ has solution.

2. Prove that the decision problems 19.11. and 19.12. are NP-complete.

19.2. Views

A database system architecture has three main levels:

- physical layer;
- logical layer;
- outer (user) layer.

The goal of the separation of levels is to reach data independence and the convenience of users. The three views on Figure 19.2 show possible user interfaces: multirelational, universal relation and graphical interface.

The ***physical layer*** consists of the actually stored data files and the dense and sparse indices built over them.

The separation of the ***logical layer*** from the physical layer makes it possible for the user to concentrate on the logical dependencies of the data, which approximates the image of the reality to be modelled better. The logical layer consists of the database schema description together with the various integrity constraints, dependencies. This the layer where the database administrators work with the system. The connection between the physical layer and the logical layer is maintained by the database engine.

The goal of the separation of the logical layer and the ***outer layer*** is that the endusers can see the database according to their (narrow) needs and requirements. For example, a very simple view of the outer layer of a bank database could be the automatic teller machine, or a much more complex view could be the credit history of a client for loan approval.

19.2.1. View as a result of a query

The question is that how can the views of different layers be given. If a query given by relational algebra expression is considered as a formula that will be applied to relational instances, then the ***view*** is obtained. Datalog rules show the difference between views and relations, well. The relations defined by rules are called ***intensional***, because these are the relations that do not have to exist on external storage devices, that is to exist extensionally, in contrast to the ***extensional*** relations.

Definition 19.21 *The $\mathcal{V}$ expression given in some query language $\mathcal{Q}$ over schema* **R** *is called a* ***view****.*

Similarly to intensional relations, views can be used in definition of queries or other views, as well.

Example 19.7 *SQL view.* Views in database manipulation language SQL can be given in the following way. Suppose that the only interesting data for us from schema **CinePest** is where and when are Kurosawa's film shown. The view **KurosawaTimes** is given by the SQL command

KUROSAWATIMES

```
1  create view KurosawaTimes as
2          select Theater, Time
3          from Film, Show
4          where Film.Title=Show.Title and Film.Director="Akira Kurosawa"
```

Written in relational algebra is as follows.

$$KurosawaTimes(Theater,\ Time) = \pi_{Theater,\ Time}(Theater \bowtie \sigma_{Director=\text{"Akira Kurosawa"}}(Film)) \tag{19.38}$$

Finally, the same by datalog rule is:

$$KurosawaTimes(x_{Th}, x_{Ti}) \leftarrow Theater(x_{Th}, x_T, x_{Ti}), Film(x_T, \text{"Akira Kurosawa"}, x_A)\ . \tag{19.39}$$

Line 2 of KUROSAWATIMES marks the selection operator used, line 3 marks that which two relations are to be joined, finally the condition of line 4 shows that it is a natural join, not a direct product.

Having defined view $\mathcal{V}$, it can be used in further queries or view definitions like any other (extensional) relation.

Advantages of using views

- Automatic data hiding: Such data that is not part of the view used, is not shown to the user, thus the user cannot read or modify them without having proper access rights to them. So by providing access to the database through views, a simple, but effective security mechanism is created.
- Views provide simple "macro capabilities". Using the view *KurosawaTimes* defined in Example 19.7 it is easy to find those theatres where Kurosawa films are shown in the morning:

$$KurosawaMorning(Theater) \leftarrow KurosawaTimes(Theater, x_{Ti}), x_{Ti} < 12\ . \tag{19.40}$$

  Of course the user could include the definition of *KurosawaTimes* in the code directly, however convenience considerations are first here, in close similarity with macros.
- Views make it possible that the same data could be seen in different ways by different users at the same time.
- Views provide ***logical data independence.*** The essence of logical data independence is that users and their programs are protected from the structural changes of the database schema. It can be achieved by defining the relations of the schema before the structural change as views in the new schema.
- Views make controlled data input possible. The **with check option** clause of command **create view** is to do this in SQL.

Materialised view

Some view could be used in several different queries. It could be useful in these cases that if the tuples of the relation(s) defined by the view need not be calculated again and again, but the output of the query defining the view is stored, and only read in at further uses. Such stored output is called a ***materialised view.***

Exercises

19.2-1 Consider the following schema:

$$FilmStar(Name, Address, Gender, BirthDate)$$
$$FilmMogul(Name, Address, Certificate\#, Assets)$$
$$Studio(Name, Address, PresidentCert\#)\ .$$

Relation *FilmMogul* contains data of the big people in film business (studio presidents, producers, etc.). The attribute names speak for themselves, *Certificate#* is the number of the certificate of the filmmogul, *PresidentCert#*) is the certificate number of the president of the studio. Give the definitions of the following views using datalog rules, relational algebra expressions, furthermore SQL:

1. *RichMogul*: Lists the names, addresses,certificate numbers and assets of those filmmoguls, whose asset value is over 1 million dollars.
2. *StudioPresident*: Lists the names, addresses and certificate numbers of those filmmoguls, who are studio presidents, as well.
3. *MogulStar*: Lists the names, addresses,certificate numbers and assets of those people who are filmstars and filmmoguls at the same time.

19.2-2 Give the definitions of the following views over schema **CinePest** using datalog rules, relational algebra expressions, furthermore SQL:

1. *Marilyn*(*Title*): Lists the titles of Marilyn Monroe's films.
2. *CorvinInfo*(*Title*,*Time*,*Phone*): List the titles and show times of films shown in theatre *Corvin*, together with the phone number of the theatre.

19.3. Query rewriting

Answering queries using views, in other words query rewriting using views has become a problem attracting much attention and interest recently. The reason is its applicability in a wide range of data manipulation problems: query optimisation, providing physical data independence, data and information integration, furthermore planning data warehouses.

The problem is the following. Assume that query Q is given over schema $\mathbf{R}$, together with views $V_1, V_2, \ldots, V_n$. Can one answer Q using only the results of views $V_1, V_2, \ldots, V_n$? Or, which is the largest set of tuples that can be determined knowing the views? If the views and the relations of the base schema can be accessed both, what is the cheapest way to compute the result of Q?

19.3.1. Motivation

Before query rewriting algorithms are studied in detail, some motivating examples of applications are given. The following university database is used throughout this section.

$$\mathbf{University} = \{Professor, Course, Teach, Registered, Major, Affiliation, Supervisor\} \ . \tag{19.41}$$

The schemata of the individual relations are as follows:

$$\begin{aligned} Professor &= \{PName, Area\} \\ Course &= \{C\text{-}Number, Title\} \\ Teaches &= \{PName, C\text{-}Number, Semester, Evaluation\} \\ Registered &= \{Student, C\text{-}Number, Semester\} \\ Major &= \{Student, Department\} \\ Affiliation &= \{PName, Department\} \\ Advisor &= \{PName, Student\} \ . \end{aligned} \tag{19.42}$$

It is supposed that professors, students and departments are uniquely identified by their names. Tuples of relation *Registered* show that which student took which course in what semester, while *Major* shows which department a student choose in majoring (for the sake of convenience it is assumed that one department has one subject as possible major).

Query optimisation
If the computation necessary to answer a query was performed in advance and the results are stored in some materialised view, then it can be used to speed up the query answering.

Consider the query that looks for pairs (*Student, Title*), where the student registered for the given Ph.D.-level course, the course is taught by a professor of the *Database* area (the C-number of graduate courses is at least 400, and the Ph.D.-level courses are those with C-number at least 500).

$$\begin{aligned} val(x_S, x_T) \quad \leftarrow \quad & Teach(x_P, x_C, x_{Se}, y_1), Professor(x_P, \text{“database”}), \\ & Registered(x_S, x_C, x_{Se}), Course(x_C, x_T), x_C \geq 500 \ . \end{aligned} \tag{19.43}$$

Suppose that the following materialised view is available that contains the registration data of graduate courses.

$$Graduate(x_S, x_T, x_C, x_{Se}) \ \leftarrow \ Registered(x_S, x_C, x_{Se}), Course(x_X, x_T), x_C \geq 400 \ . \tag{19.44}$$

View *Graduate* can be used to answer query (19.43).

$$\begin{aligned} val(x_S, x_T) \quad \leftarrow \quad & Teaches(x_P, x_C, x_{Se}, y_1), Professor(x_P, \text{“database”}), \\ & (x_S, x_T, x_C, x_{Se}), x_C \geq 500 \ . \end{aligned} \tag{19.45}$$

It will be faster to compute (19.45) than to compute (19.43), because the natural join of relations *Registered* and *Course* has already be done by view *Graduate*, furthermore it shelled off the undergraduate courses (that make up for the bulk of registration data at most universities). It worth noting that view *Graduate* could be used event hough ***syntactically*** did not agree with any part of query (19.43).

On the other hand, it may happen that the the original query can be answered faster. If relations *Registered* and *Course* have an index on attribute *C-Number*, but there exists no index built for *Graduate*, then it could be faster to answer query (19.43) directly from the database relations. Thus, the real challenge is not only that to decide about a materialised view whether it could be used to answer some query logically, but a thorough cost analysis must be done when is it worth using the existing views.

Physical data independence
One of the principles underlying modern database systems is the separation between the logical view of data and the physical view of data. With the exception of horizontal or vertical partitioning of relations into multiple files, relational database systems are still largely based on a one-to-one correspondence between relations in the schema and the files in which they are stored. In object-oriented systems, maintaining the separation is necessary because the logical schema contains significant redundancy, and does not correspond to a good physical layout. Maintaining physical data independence becomes more crucial in applications where the logical model is introduced as an intermediate level after the physical representation has already been determined. This is common in storage of XML data in relational databases, and in data integration. In fact, the Stored System stores XML documents in a relational database, and uses views to describe the mapping from XML into relations

def.gmap G1 **as** b$^+$-tree **by**
given *Student.name*
select *Department*
where *Student* major *Department*

def.gmap G2 **as** b$^+$-tree **by**
given *Student.name*
select *Course.c-number*
where *Student* registered *Course*

def.gmap G3 **as** b$^+$-tree **by**
given *Course.c-number*
select *Department*
where *Student* registered *Course* **and** *Student* major *Department*

Figure 19.3. GMAPs for the university domain.

in the database.

To maintain physical data independence, a widely accepted method is to use views over the logical schema as mechanism for describing the physical storage of data. In particular, Tsatalos, Solomon and Ioannidis use GMAPs (*Generalised Multi-level Access Paths*) to describe data storage.

A GMAP describes the physical organisation and the indexes of the storage structure. The first clause of the GMAP (**as**) describes the actual data structure used to store a set of tuples (e.g., a B$^+$-tree, hash index, etc.) The remaining clauses describe the content of the structure, much like a view definition. The **given** and **select** clauses describe the available attributes, where the **given** clause describes the attributes on which the structure is indexed. The definition of the view, given in the **where** clause uses infix notation over the conceptual model.

In the example shown in Figure 19.3, the GMAP G1 stores a set of pairs containing students and the departments in which they major, and these pairs are indexed by a B$^+$-tree on attribute *Student.name*. The GMAP G2 stores an index from names of students to the numbers of courses in which they are registered. The GMAP G3 stores an index from course numbers to departments whose majors are enrolled in the course.

Given that the data is stored in structures described by the GMAPs, the question that arises is how to use these structures to answer queries. Since the logical content of the GMAPs are described by views, answering a query amounts to finding a way of rewriting the query using these views. If there are multiple ways of answering the query using the views, we would like to find the cheapest one. Note that in contrast to the query optimisation context, we ***must*** use the views to answer the given query, because all the data is stored in the GMAPs.

Consider the following query, which asks for names of students registered for

Ph.D.-level courses and the departments in which these students are majoring.

$$\begin{aligned} ans(Student,Department) \leftarrow & Registered(Student,C\text{-}number,y), \\ & Major(Student,Department), \\ & C\text{-}number \geq 500\ . \end{aligned} \tag{19.46}$$

The query can be answered in two ways. First, since *Student.name* uniquely identifies a student, we can take the join of G! and G2, and then apply selection operator $Course.c\text{-}number \geq 500$, finally a projection eliminates the unnecessary attributes. The other execution plan could be to join G3 with G2 and select $Course.c\text{-}number \geq 500$. In fact, this solution may even be more efficient because G3 has an index on the course number and therefore the intermediate joins may be much smaller.

Data integration

A ***data integration system*** (also known as ***mediator system***) provides a ***uniform*** query interface to a multitude of autonomous heterogeneous data sources. Prime examples of data integration applications include enterprise integration, querying multiple sources on the World-Wide Web, and integration of data from distributed scientific experiments.

To provide a uniform interface, a data integration system exposes to the user a ***mediated schema.*** A mediated schema is a set of ***virtual*** relations, in the sense that they are not stored anywhere. The mediated schema is designed manually for a particular data integration application. To be able to answer queries, the system must also contain a set of ***source descriptions.*** A description of a data source specifies the contents of the source, the attributes that can be found in the source, and the (integrity) constraints on the content of the source. A widely adopted approach for specifying source descriptions is to describe the contents of a data source as a ***view*** over the mediated schema. This approach facilitates the addition of new data sources and the specification of constraints on the contents of sources.

In order to answer a query, a data integration system needs to translate a query formulated on the mediated schema into one that refers directly to the schemata of the data sources. Since the contents of the data sources are described as views, the translation problem amounts finding a way to answer a query using a set of views.

Consider as an example the case where the mediated schema exposed to the user is schema **University**, except that the relations *Teaches* and *Course* have an additional attribute identifying the university at which the course is being taught:

$$\begin{aligned} Course &= \{C\text{-}number, Title, Univ\} \\ Teaches &= \{PName, C\text{-}number, Semester, Evaluation, Univ.\} \end{aligned} \tag{19.47}$$

Suppose we have the following two data sources. The first source provides a listing of all the courses entitled "Database Systems" taught anywhere and their instructors. This source can be described by the following view definition:

$$\begin{aligned} DBcourse(Title,PName,C\text{-}number,Univ) \leftarrow & Course(\ C\text{-}number,Title,Univ), \\ & Teaches(PName,C\text{-}number,Semester, \\ & \quad Evaluation,Univ), \\ & Title = \text{“Database Systems”}\ . \end{aligned} \tag{19.48}$$

The second source lists Ph.D.-level courses being taught at The Ohio State University (OSU), and is described by the following view definition:

$$\begin{aligned} OSUPhD(Title,PName,C\text{-}number,Univ) \leftarrow & Course(\ C\text{-}number,Title,Univ), \\ & Teaches(PName,C\text{-}number,Semester, \\ & \qquad Evaluation,Univ), \\ & Univ = \text{“OSU”}, C\text{-}number \geq 500. \end{aligned} \tag{19.49}$$

If we were to ask the data integration system who teaches courses titled "Database Systems" at OSU, it would be able to answer the query by applying a selection on the source *DB-courses*:

$$ans(PName) \ \leftarrow \ DBcourse(Title,PName,C\text{-}number,Univ), Univ = \text{"OSU"} \ . \tag{19.50}$$

On the other hand, suppose we ask for all the graduate-level courses (not just in databases) being offered at OSU. Given that only these two sources are available, the data integration system cannot find ***all*** tuples in the answer to the query. Instead, the system can attempt to find the maximal set of tuples in the answer available from the sources. In particular, the system can obtain graduate ***database*** courses at OSU from the *DB-course* source, and the Ph.D.-level courses at OSU from the *OSUPhD* source. Hence, the following non-recursive datalog program gives the maximal set of answers that can be obtained from the two sources:

$$\begin{aligned} ans(Title,C\text{-}number) \leftarrow & DBcourse(Title,PName,C\text{-}number,Univ), \\ & Univ = \text{"OSU"}, C\text{-}number \geq 400 \\ ans(Title,C\text{-}number) \leftarrow & OSUPhD(Title,PName,C\text{-}number,Univ) \ . \end{aligned} \tag{19.51}$$

Note that courses that are not PH.D.-level courses or database courses will not be returned as answers. Whereas in the contexts of query optimisation and maintaining physical data independence the focus is on finding a query expression that is ***equivalent*** with the original query, here finding a query expression that provides the ***maximal answers*** from the views is attempted.

Semantic data caching
If the database is accessed via client-server architecture, the data cached at the client can be semantically modelled as results of certain queries, rather than at the physical level as a set of data pages or tuples. Hence, deciding which data needs to be shipped from the server in order to answer a given query requires an analysis of which parts of the query can be answered by the cached views.

19.3.2. Complexity problems of query rewriting

In this section the ***theoretical*** complexity of query rewriting is studied. Mostly conjunctive queries are considered. ***Minimal,*** and ***complete*** rewriting will be distinguished. It will be shown that if the query is conjunctive, furthermore the materialised views are also given as results of conjunctive queries, then the rewriting problem is *NP-complete,* assuming that neither the query nor the view definitions contain comparison atoms. Conjunctive queries will be considered in rule-based form for the

sake of convenience.

Assume that query Q is given over schema $\mathbf{R}$.

Definition 19.22 *The conjunctive query Q' is a **rewriting** of query Q using views $\mathcal{V} = V_1, V_2, \ldots, V_m$, if*

- *Q and Q' are equivalent, and*
- *Q' contains one or more literals from $\mathcal{V}$.*

*Q' is said to be **locally minimal** if no literal can be removed from Q' without violating the equivalence. The rewriting is **globally minimal,** if there exists no rewriting using a smaller number of literals. (The comparison atoms $=, \neq, \leq, <$ are not counted in the number of literals.)*

Example 19.8 *Query rewriting.* Consider the following query Q and view V.

$$\begin{array}{lll} Q\colon q(X,U) & \leftarrow & p(X,Y), p_0(Y,Z), p_1(X,W), p_2(W,U) \\ V\colon v(A,B) & \leftarrow & p(A,C), p_0(C,B), p_1(A,D)\ . \end{array} \tag{19.52}$$

Q can be rewritten using V:

$$Q'\colon q(X,U) \ \leftarrow \ v(X,Z), p_1(X,W), p_2(W,U)\ . \tag{19.53}$$

View V replaces the first two literals of query Q. Note that the view certainly satisfies the third literal of the query, as well. However, it cannot be removed from the rewriting, since variable D does not occur in the head of V, thus if literal p_1 were to be removed, too, then the natural join of p_1 and p_2 would not be enforced anymore.

Since in some of the applications the database relations are inaccessible, only the views can be accessed, for example in case of data integration or data warehouses, the concept of ***complete rewriting*** is introduced.

Definition 19.23 *A rewriting Q' of query Q using views $\mathcal{V} = V_1, V_2, \ldots, V_m$ is called a **complete rewriting,** if Q' contains only literals of $\mathcal{V}$ and comparison atoms.*

Example 19.9 *Complete rewriting.* Assume that besides view V of Example 19.8 the following view is given, as well:

$$V_2\colon v_2(A,B) \ \leftarrow \ p_1(A,C), p_2(C,B), p_0(D,E) \tag{19.54}$$

A complete rewriting of query Q is:

$$Q''\colon q(X,U) \ \leftarrow \ v(X,Z), v_2(X,U)\ . \tag{19.55}$$

It is important to see that this rewriting cannot be obtained ***step-by-step,*** first using only V, then trying to incorporate V_2, (or just in the opposite order) since relation p_0 of V_2 does not occur in Q'. Thus, in order to find the complete rewriting, use of the two view must be considered ***parallel,*** at the same time.

There is a close connection between finding a rewriting and the problem of query containment. This latter one was discussed for tableau queries in section 19.1.3. Homomorphism between tableau queries can be defined for rule based queries, as well.

The only difference is that it is not required in this section that a homomorphism maps the head of one rule to the head of the other. (The head of a rule corresponds to the summary row of a tableau.) According to Theorem 19.20 it is NP-complete to decide whether conjunctive query Q_1 contains another conjunctive query Q_2. This remains true in the case when Q_2 may contain comparison atoms, as well. However, if both, Q_1 and Q_2 may contain comparison atoms, then the existence of a homomorphism from Q_1 to Q_2 is only a sufficient but not necessary condition for the containment of queries, which is a Π_2^p-complete problem in that case. The discussion of this latter complexity class is beyond the scope of this chapter, thus it is omitted. The next proposition gives a necessary and sufficient condition whether there exists a rewriting of query Q using view V.

Proposition 19.24 *Let Q and V be conjunctive queries that may contain comparison atoms. There exists a a rewriting of query Q using view V if and only if $\pi_\emptyset(Q) \subseteq \pi_\emptyset(V)$, that is the projection of V to the empty attribute set contains that of Q.*

Proof Observe that $\pi_\emptyset(Q) \subseteq \pi_\emptyset(V)$ is equivalent with the following proposition: If the output of V is empty for some instance, then the same holds for the output of Q, as well.

Assume first that there exists a rewriting, that is a rule equivalent with Q that contains V in its body. If r is such an instance, that the result of V is empty on it, then every rule that includes V in its body results in empty set over r, too.

In order to prove the other direction, assume that if the output of V is empty for some instance, then the same holds for the output of Q, as well. Let

$$\begin{array}{lll} Q\colon q(\tilde{x}) & \leftarrow & q_1(\tilde{x}), q_2(\tilde{x}), \ldots, q_m(\tilde{x}) \\ V\colon v(\tilde{a}) & \leftarrow & v_1(\tilde{a}), v_2(\tilde{a}), \ldots, v_n(\tilde{a})\ . \end{array} \tag{19.56}$$

Let $\tilde{y}$ be a list of variables disjoint from variables of $\tilde{x}$. Then the query Q' defined by

$$Q'\colon q'(\tilde{x}) \ \leftarrow \ q_1(\tilde{x}), q_2(\tilde{x}), \ldots, q_m(\tilde{x}), v_1(\tilde{y}), v_2(\tilde{y}), \ldots, v_n(\tilde{y}) \tag{19.57}$$

satisfies $Q \equiv Q'$. It is clear that $Q' \subseteq Q$. On the other hand, if there exists a valuation of the variables of $\tilde{y}$ that satisfies the body of V over some instance r, then fixing it, for arbitrary valuation of variables in $\tilde{x}$ a tuple is obtained in the output of Q, whenever a tuple is obtained in the output of Q' together with the previously fixed valuation of variables of $\tilde{y}$. ■

As a corollary of Theorem 19.20 and Proposition 19.24 the following theorem is obtained.

Theorem 19.25 *Let Q be a conjunctive query that may contain comparison atoms, and let $\mathcal{V}$ be a set of views. If the views in $\mathcal{V}$ are given by conjunctive queries that do not contain comparison atoms, then it is NP-complete to decide whether there exists a rewriting of Q using $\mathcal{V}$.*

The proof of Theorem 19.25 is left for the Reader (Exercise 19.3-1).

The proof of Proposition 19.24 uses new variables. However, according to the next lemma, this is not necessary. Another important observation is that it is enough

to consider a subset of database relations occurring in the original query when locally minimal rewriting is sought, new database relations need not be introduced.

Lemma 19.26 *Let Q be a conjunctive query that does not contain comparison atoms*

$$Q\colon q(\tilde{X}) \leftarrow p_1(\tilde{U}_1), p_2(\tilde{U}_2), \ldots, p_n(\tilde{U}_n)\ , \tag{19.58}$$

furthermore let $\mathcal{V}$ be a set of views that do not contain comparison atoms either.

1. *If Q' is a locally minimal rewriting of Q using $\mathcal{V}$, then the set of database literals in Q' is isomorphic to a subset of database literals occurring in Q.*
2. *If*

$$q(\tilde{X}) \leftarrow p_1(\tilde{U}_1), p_2(\tilde{U}_2), \ldots, p_n(\tilde{U}_n), v_1(\tilde{Y}_1), v_2(\tilde{Y}_2), \ldots v_k(\tilde{Y}_k) \tag{19.59}$$

is a rewriting of Q using the views, then there exists a rewriting

$$q(\tilde{X}) \leftarrow p_1(\tilde{U}_1), p_2(\tilde{U}_2), \ldots, p_n(\tilde{U}_n), v_1(\tilde{Y'}_1), v_2(\tilde{Y'}_2), \ldots v_k(\tilde{Y'}_k) \tag{19.60}$$

such that $\{\tilde{Y'}_1 \cup \cdots \cup \tilde{Y'}_k\} \subseteq \{\tilde{U}_1 \cup \cdots \cup \tilde{U}_n\}$, that is the rewriting does not introduce new variables.

The details of the proof of Lemma 19.26 are left for the Reader (Exercise 19.3-2). The next lemma is of fundamental importance: A minimal rewriting of Q using $\mathcal{V}$ cannot ***increase*** the number of literals.

Lemma 19.27 *Let Q be conjunctive query, $\mathcal{V}$ be set of views given by conjunctive queries, both without comparison atoms. If the body of Q contains p literals and Q' is a locally minimal rewriting of Q using $\mathcal{V}$, then Q' contains at most p literals.*

Proof Replacing the view literals of Q' by their definitions query Q'' is obtained. Let φ be a homomorphism from the body of Q to Q''. The existence of φ follows from $Q \equiv Q''$ by the Homomorphism Theorem (Theorem 19.18). Each of the literals $l_1, l_2, \ldots, l_p$ of the body of Q is mapped to at most one literal obtained from the expansion of view definitions. If Q' contains more than p view literals, then the expansion of some view literals in the body of Q'' is disjoint from the image of φ. These view literals can be removed from the body of Q' without changing the equivalence. ■

Based on Lemma 19.27 the following theorem can be stated about complexity of minimal rewritings.

Theorem 19.28 *Let Q be conjunctive query, $\mathcal{V}$ be set of views given by conjunctive queries, both without comparison atoms. Let the body of Q contain p literals.*

1. *It is NP-complete to decide whether there exists a rewriting Q' of Q using $\mathcal{V}$ that uses at most k $(\leq p)$ literals.*
2. *It is NP-complete to decide whether there exists a rewriting Q' of Q using $\mathcal{V}$ that uses at most k $(\leq p)$ database literals.*

3. *It is NP-complete to decide whether there exists a complete rewriting of Q using $\mathcal{V}$.*

Proof The first statement is proved, the proof of the other two is similar. According to Lemmas 19.27 and 19.26, only such rewritings need to be considered that have at most as many literals as the query itself, contain a subset of the literals of the query and do not introduce new variables. Such a rewriting and the homomorphisms proving the equivalence can be tested in polynomial time, hence the problem is in NP. In order to prove that it is NP-hard, Theorem 19.25 is used. For a given query Q and view V let V' be the view, whose head is same as the head of V, but whose body is the conjunction of the bodies of Q and V. It is easy to see that there exists a rewriting using V' with a single literal if and only if there exists a rewriting (with no restriction) using V. ■

19.3.3. Practical algorithms

In this section only complete rewritings are studied. This does not mean real restriction, since if database relations are also to be used, then views mirroring the database relations one-to-one can be introduced. The concept of ***equivalent*** rewriting introduced in Definition 19.22 is appropriate if the goal of the rewriting is query optimisation or providing physical data independence. However, in the context of data integration on data warehouses equivalent rewritings cannot be sought, since all necessary data may not be available. Thus, the concept of maximally contained rewriting is introduced that depends on the query language used, in contrast to equivalent rewritings.

Definition 19.29 *Let Q be a query, $\mathcal{V}$ be a set of views, $\mathcal{L}$ be a query language. Q' is a **maximally contained rewriting** of Q with respect to $\mathcal{L}$, if*

1. *Q' is a query of language $\mathcal{L}$ using only views from $\mathcal{V}$,*
2. *Q contains Q',*
3. *if query $Q_1 \in \mathcal{L}$ satisfies $Q' \sqsubseteq Q_1 \sqsubseteq Q$, then $Q' \equiv Q_1$.*

Query optimisation using materialised views

Before discussing how can a traditional optimiser be modified in order to use materialised views instead of database relations, it is necessary to survey when can view be used to answer a given query. Essentially, view V can be used to answer query Q, if the intersection of the sets of database relations in the body of V and in the body of Q is non-empty, furthermore some of the attributes are selected by V are also selected by Q. Besides this, in case of equivalent rewriting, if V contains comparison atoms for such attributes that are also occurring in Q, then the view must apply logically equivalent, or weaker condition, than the query. If logically stronger condition is applied in the view, then it can be part of a (maximally) contained rewriting. This can be shown best via an example. Consider the query Q over schema **University** that list those professor, student, semester triplets, where the advisor of the student is the professor and in the given semester the student registered for

some course taught by the professor.

$$\begin{aligned} Q\colon q(\textit{Pname,Student,Semester}) \leftarrow & \textit{Registered}(\textit{Student,C-number,Semester}), \\ & \textit{Advisor}(\textit{Pname,Student}), \\ & \textit{Teaches}(\textit{Pname, C-number,Semester}, x_E), \\ & \textit{Semester} \geq \text{“Fall2000”} \ . \end{aligned} \tag{19.61}$$

View V_1 below can be used to answer Q, since it uses the same join condition for relations *Registered* and *Teaches* as Q, as it is shown by variables of the same name. Furthermore, V_1 selects attributes *Student, PName, Semester*, that are necessary in order to properly join with relation *Advisor*, and for select clause of the query. Finally, the predicate *Semester* > “Fall1999” is weaker than the predicate *Semester* ≥ “Fall2000” of the query.

$$\begin{aligned} V_1\colon v_1(\textit{Student,PName,Semester}) \leftarrow & \textit{Teaches}(\textit{PName,C-number,Semester}, x_E), \\ & \textit{Registered}(\textit{Student,C-number,Semester}), \\ & \textit{Semester} > \text{“Fall1999”} \ . \end{aligned} \tag{19.62}$$

The following four views illustrate how minor modifications to V_1 change the usability in answering the query.

$$\begin{aligned} V_2\colon v_2(\textit{Student,Semester}) \leftarrow & \textit{Teaches}(x_P, \textit{C-number,Semester}, x_E), \\ & \textit{Registered}(\textit{Student,C-number,Semester}), \\ & \textit{Semester} > \text{“Fall1999”} \ . \end{aligned} \tag{19.63}$$

$$\begin{aligned} V_3\colon v_3(\textit{Student,PName,Semester}) \leftarrow & \textit{Teaches}(\textit{PName, C-number}, x_S, x_E), \\ & \textit{Registered}(\textit{Student,C-number,Semester}), \\ & \textit{Semester} > \text{“Fall1999”} \ . \end{aligned} \tag{19.64}$$

$$\begin{aligned} V_4\colon v_4(\textit{Student,PName,Semester}) \leftarrow & \textit{Teaches}(\textit{PName, C-number,Semester}, x_E), \\ & \textit{Adviser}(\textit{PName}, x_{St}), \textit{Professor}(\textit{PName}, x_A), \\ & \textit{Registered}(\textit{Student,C-number,Semester}), \\ & \textit{Semester} > \text{“Fall1999”} \ . \end{aligned} \tag{19.65}$$

$$\begin{aligned} V_5\colon v_5(\textit{Student,PName,Semester}) \leftarrow & \textit{Teaches}(\textit{PName, C-number,Semester}, x_E), \\ & \textit{Registered}(\textit{Student,C-number,Semester}), \\ & \textit{Semester} > \text{“Fall2001”} \ . \end{aligned} \tag{19.66}$$

View V_2 is similar to V_1, except that it does not select the attribute *PName* from relation *Teaches*, which is needed for the join with the relation *Adviser* and for the selection of the query. Hence, to use V_2 in the rewriting, it has to be joined with relation *Teaches* again. Still, if the join of relations *Registered* and *Teaches* is very selective, then employing V_2 may actually result in a more efficient query execution plan.

In view V_3 the join of relations *Registered* and *Teaches* is over only attribute *C-number*, the equality of variables *Semester* and x_S is not required. Since attribute x_S is not selected by V_3, the join predicate cannot be applied in the rewriting, and therefore there is little gain by using V_3.

View V_4 considers only the professors who have at least one area of research. Hence, the view applies an additional condition that does not exists in the query, and cannot be used in an equivalent rewriting unless union and negation are allowed in the rewriting language. However, if there is an integrity constraint stating that every professor has at least one area of research, then an optimiser should be able to realise that V_4 is usable.

Finally, view V_5 applies a stronger predicate than the query, and is therefore usable for a contained rewriting, but not for an equivalent rewriting of the query.

System-R style optimisation

Before discussing the changes to traditional optimisation, first the principles underlying the ***System-R style optimiser*** is recalled briefly. System-R takes a bottom-up approach to building query execution plans. In the first phase, it concentrates of plans of size 1, i.e., chooses the best access paths to every table mentioned in the query. In phase n, the algorithm considers plans of size n, by combining plans obtained in the previous phases (sizes of k and $n-k$). The algorithm terminates after constructing plans that cover all the relations in the query. The efficiency of System-R stems from the fact that it partitions query execution plans into ***equivalence classes,*** and only considers a single execution plan for every equivalence class. Two plans are in the same equivalence class if they

- cover the same set of relations in the query (and therefore are also of the same size), and
- produce the answer in the same interesting order.

In our context, the query optimiser builds query execution plans by accessing a set of views, rather than a set of database relations. Hence, in addition to the meta-data that the query optimiser has about the materialised views (e.g., statistics, indexes) the optimiser is also given as input the query expressions defining the views. Th additional issues that the optimiser needs to consider in the presence of materialised views are as follows.

A. In the first iteration the algorithm needs to decide which views are ***relevant*** to the query according to the conditions illustrated above. The corresponding step is trivial in a traditional optimiser: a relation is relevant to the query if it is in the body of the query expression.

B. Since the query execution plans involve joins over views, rather than joins over database relations, plans can no longer be neatly partitioned into equivalence classes which can be explored in increasing size. This observation implies several changes to the traditional algorithm:

1. ***Termination testing:*** the algorithm needs to distinguish ***partial query execution plans*** of the query from ***complete query execution plans.*** The enumeration of the possible join orders terminates when there are no more unexplored partial plans. In contrast, in the traditional setting the algorithm terminates after considering the equivalence classes that include all the relations of the query.
2. ***Pruning of plans:*** a traditional optimiser compares between pairs of plans ***within*** one equivalence class and saves only the cheapest one for each class. I our context, the query optimiser needs to compare between ***any pair*** of plans generated thus far. A plan p is pruned if there is another plan p' that

   (a) is cheaper than p, and
   (b) has greater or equal contribution to the query than p. Informally, a plan p' contributes more to the query than plan p if it covers more of the relations in the query and selects more of the necessary attributes.

3. ***Combining partial plans:*** in the traditional setting, when two partial plans are combined, the join predicates that involve both plans are explicit in the query, and the enumeration algorithm need only consider the most efficient way to apply these predicates. However, in our case, it may not be obvious a priori which join predicate will yield a correct rewriting of the query, since views are joined rather than database relations directly. Hence, the enumeration algorithm needs to consider several alternative join predicates. Fortunately, in practice, the number of join predicates that need to be considered can be significantly pruned using meta-data about the schema. For example, there is no point in trying to join a string attribute with a numeric one. Furthermore, in some cases knowledge of integrity constraints and the structure of the query can be used to reduce the number of join predicates to be considered. Finally, after considering all the possible join predicates, the optimiser also needs to check whether the resulting plan is still a partial solution to the query.

The following table summarises the comparison of the traditional optimiser versus one that exploits materialised views.

Conventional optimiser	**Optimiser using views**
Iteration 1	***Iteration 1***
a) Find all possible access paths.	a1)Find all views that are relevant to the query. a2) Distinguish between partial and complete solutions to the query.
b) Compare their cost and keep the least expensive.	b) Compare all pairs of views. If one has neither greater contribution nor a lower cost than the other, prune it.
c) If the query has one relation, **stop**.	c) If there are no partial solutions, **stop**.
Iteration 2	***Iteration 2***
For each query join:	
a) Consider joining the relevant access paths found in the previous iteration using all possible join methods.	a1) Consider joining all partial solutions found in the previous iteration using all possible equi-join methods and trying all possible subsets of join predicates. a2) Distinguish between partial and complete solutions to the query.
b) Compare the cost of the resulting join plans and keep the least expensive.	b) If any newly generated solution is either not relevant to the query, or dominated by another, prune it.
c) If the query has two relations, **stop**.	c) If there are no partial solutions, **stop**.
Iteration 3	***Iteration 3***
⋮	⋮

Another method of equivalent rewriting is using transformation rules. The common theme in the works of that area is that replacing some part of the query with a view is considered as another transformation available to the optimiser. These methods are not discussed in detail here.

The query optimisers discussed above were designed to handle cases where the number of views is relatively small (i.e., comparable to the size of the database schema), and cases where equivalent rewriting is required. In contrast, the context of data integration requires consideration of large number of views, since each data source is being described by one or more views. In addition, the view definitions may contain many complex predicates, whose goal is to express fine-grained distinction between the contents of different data sources. Furthermore, in the context of data integration it is often assumed that the views are not complete, i.e., they may only contain a subset of the tuples satisfying their definition. In the foregoing, some algorithms for answering queries using views are described that were developed specifically for the context of data integration.

The Bucket Algorithm

The goal of the Bucket Algorithm is to reformulate a user query that is posed on a mediated (virtual) schema into a query that refers directly to the available sources. Both the query and the sources are described by conjunctive queries that may include atoms of arithmetic comparison predicates. The set of comparison atoms of query Q is denoted by $C(Q)$.

Since the number of possible rewritings may be exponential in the size of the query, the main idea underlying the bucket algorithm is that the number of query rewritings that need to be considered can be drastically reduced if we first consider each ***subgoal*** – the relational atoms of the query – is considered in isolation, and determine which views may be relevant to each subgoal.

The algorithm proceeds as follows. First, a ***bucket*** is created for each subgoal in the query Q that is not in $C(Q)$, containing the views that are relevant to answering the particular subgoal. In the second step, all such conjunctive query rewritings are considered that include one conjunct (view) from each bucket. For each rewriting V obtained it is checked that whether it is semantically correct, that is $V \sqsubseteq Q$ holds, or whether it can be made semantically correct by adding comparison atoms. Finally the remaining plans are minimised by pruning redundant subgoals. Algorithm CREATE-BUCKET executes the first step described above. Its input is a set of source descriptions $\mathcal{V}$ and a conjunctive query Q in the form

$$Q\colon Q(\tilde{X}) \leftarrow R_1(\tilde{X}_1), R_2(\tilde{X}_2), \ldots, R_m(\tilde{X}_m), C(Q)\ . \tag{19.67}$$

CREATE-BUCKET(Q,$\mathcal{V}$)

```
1  for i ← 1 to m
2      do Bucket[i] ← ∅
3         for all V ∈ 𝒱
4                                  ▷ V is of form V: V(Ỹ) ← S_1(Ỹ_1), ... S_n(Ỹ_n), C(V).
5             do for j ← 1 to n
6                if R_i = S_j
7                   then let φ be a mapping defined on the variables
                         of V as follows:
8                        if y is the k^th variable of Ỹ_j and y ∈ Ỹ
9                           then φ(y) = x_k, where x_k is the k^th variable of X̃_i
10                          else φ(y) is a new variable that
                                does not appear in Q or V.
11                       Q'() ← R_1(X̃_1), R_m(X̃_m), C(Q), S_1(φ(Ỹ_1)), ...,
                         S_n(φ(Ỹ_n)), φ(C(V))
12                       if SATISFIABLE^≥(Q')
13                          then add φ(V) to Bucket[i].
14 return Bucket
```

Procedure SATISFIABLE$^{\geq}$ is the extension of SATISFIABLE described in section 19.1.2 to the case when comparison atoms may occur besides equality atoms. The necessary change is only that for all variable y occurring in comparison atoms it

must be checked whether all predicates involving y are satisfiable simultaneously.

CREATE-BUCKET running time is polynomial function of the sizes of Q and $\mathcal{V}$. Indeed, the kernel of the nested loops of lines 3 and 5 runs $n \sum_{V \in \mathcal{V}} |V|$ times. The commands of lines 6–13 require constant time, except for line 12. The condition of of command **if** in line 12 can be checked in polynomial time.

In order to prove the correctness of procedure CREATE-BUCKET, one should check under what condition is a view V put in $Bucket[i]$. In line 6 it is checked whether relation R_i appears as a subgoal in V. If not, then obviously V cannot give usable information for subgoal R_i of Q. If R_i is a subgoal of V, then in lines 9–10 a mapping is constructed that applied to the variables allows the correspondence between subgoals S_j and R_i, in accordance with relations occurring in the heads of Q and V, respectively. Finally, in line 12 it is checked whether the comparison atoms contradict with the correspondence constructed.

In the second step, having constructed the buckets using CREATE-BUCKET, the bucket algorithm finds a set of ***conjunctive query rewritings,*** each of them being a conjunctive query that includes one conjunct from every bucket. Each of these conjunctive query rewritings represents one way of obtaining part of the answer to Q from the views. The result of the bucket algorithm is defined to be the union of the conjunctive query rewritings (since each of the rewritings may contribute different tuples). A given conjunctive query Q' is a ***conjunctive query rewriting,*** if

1. $Q' \sqsubseteq Q$, or
2. Q' can be extended with comparison atoms, so that the previous property holds.

Example 19.10 *Bucket algorithm.* Consider the following query Q that lists those articles x that there exists another article y of the same area such that x and y mutually cites each other. There are three views (sources) available, V_1, V_2, V_3.

$$\begin{array}{lcl} Q(x) & \leftarrow & cite(x,y), cite(y,x), sameArea(x,y) \\ V_1(a) & \leftarrow & cite(a,b), cite(b,a) \\ V_2(c,d) & \leftarrow & sameArea(c,d) \\ V_3(f,h) & \leftarrow & cite(f,g), cite(g,h), sameArea(f,g)\ . \end{array} \tag{19.68}$$

In the first step, applying CREATE-BUCKET, the following buckets are constructed.

$cite(x,y)$	$cite(y,x)$	$sameArea(x,y)$
$V_1(x)$	$V_1(x)$	$V_2(x)$
$V_3(x)$	$V_3(x)$	$V_3(x)$

(19.69)

In the second step the algorithm constructs a conjunctive query Q' from each element of the Cartesian product of the buckets, and checks whether Q' is contained in Q. If yes, it is given to the answer.

In our case, it tries to match V_1 with the other views, however no correct answer is obtained so. The reason is that b does not appear in the head of V_1, hence the join condition of Q – variables x and y occur in relation *sameArea*, as well – cannot be applied. Then rewritings containing V_3 are considered, recognising that equating the variables in the head of V_3 a contained rewriting is obtained. Finally, the algorithm finds that combining V_3 and V_2 rewriting is obtained, as well. This latter is redundant, as it is obtained by simple checking, that is V_2 can be pruned. Thus, the result of the bucket algorithm for query

(19.68) is the following (actually equivalent) rewriting

$$Q'(x) \leftarrow V_3(x,x)\ . \tag{19.70}$$

The strength of the bucket algorithm is that it exploits the predicates in the query to prune significantly the number of candidate conjunctive rewritings that need to be considered. Checking whether a view should belong to a bucket can be done in time polynomial in the size of the query and view definition when the predicates involved are arithmetic comparisons. Hence, if the data sources (i.e., the views) are indeed distinguished by having different comparison predicates, then the resulting buckets will be relatively small.

The main disadvantage of the bucket algorithm is that the Cartesian product of the buckets may still be large. Furthermore, the second step of the algorithm needs to perform a query containment test for every candidate rewriting, which is NP-complete even when no comparison predicates are involved.

Inverse-rules algorithm

The Inverse-rules algorithm is a procedure that can be applied more generally than the Bucket algorithm. It finds a maximally contained rewriting for any query given by arbitrary recursive datalog program that does not contain negation, in polynomial time.

The first question is that for given datalog program $\mathcal{P}$ and set of conjunctive queries $\mathcal{V}$, whether there exists a datalog program $\mathcal{P}_v$ equivalent with $\mathcal{P}$, whose *edb* relations are relations $v_1, v_2, \ldots, v_n$ of $\mathcal{V}$. Unfortunately, this is algorithmically undecidable. Surprisingly, the best, maximally contained rewriting can be constructed. In the case, when there exists a datalog program $\mathcal{P}_v$ equivalent with $\mathcal{P}$, the algorithm finds that, since a maximally contained rewriting contains $\mathcal{P}_v$, as well. This seemingly contradicts to the fact that the existence of equivalent rewriting is algorithmically undecidable, however it is undecidable about the result of the inverse-rules algorithm, whether it is really equivalent to the original query.

Example 19.11 *Equivalent rewriting.* Consider the following datalog program $\mathcal{P}$, where *edb* relations *edge* and *black* contain the edges and vertices coloured black of a graph G.

$$\begin{array}{lllll} \mathcal{P}\colon & q(X,Y) & \leftarrow & edge(X,Z), edge(Z,Y), black(Z) \\ & q(X,Y) & \leftarrow & edge(X,Z), black(Z), q(Z,Y)\ . \end{array} \tag{19.71}$$

It is easy to check that $\mathcal{P}$ lists the endpoints of such paths (more precisely walks) of graph G whose inner points are all black. Assume that only the following two views can be accessed.

$$\begin{array}{lll} v_1(X,Y) & \leftarrow & edge(X,Y), black(X) \\ v_2(X,Y) & \leftarrow & edge(X,Y), black(Y) \end{array} \tag{19.72}$$

v_1 stores edges whose tail is black, while v_2 stores those, whose head is black. There exists an equivalent rewriting $\mathcal{P}_v$ of datalog program $\mathcal{P}$ that uses only views v_1 and v_2 as *edb* relations:

$$\begin{array}{lllll} \mathcal{P}_v\colon & q(X,Y) & \leftarrow & v_2(X,Z), v_1(Z,Y) \\ & q(X,Y) & \leftarrow & v_2(X,Z), q(Z,Y) \end{array} \tag{19.73}$$

However, if only v_1, or v_2 is accessible alone, then equivalent rewriting is not possible, since

only such paths are obtainable whose starting, or respectively, ending vertex is black.

In order to describe the Inverse-rules Algorithm, it is necessary to introduce the ***Horn rule***, which is a generalisation of ***datalog program,*** and ***datalog rule***. If ***function symbols*** are also allowed in the free tuple u_i of rule (19.27) in Definition 19.11, besides variables and constants, then ***Horn rule*** is obtained. A ***logic program*** is a collection of Horn rules. In this sense a logic program without function symbols is a datalog program. The concepts of *edb* and *idb* can be defined for logic programs in the same way as for datalog programs.

The Inverse-rules Algorithm consists of two steps. First, a logic program is constructed that may contain function symbols. However, these will not occur in recursive rules, thus in the second step they can be eliminated and the logic program can be transformed into a datalog program.

Definition 19.30 *The* ***inverse*** v^{-1} *of view* v *given by*

$$v(X_1, \ldots, X_m) \leftarrow v_1(\tilde{Y}_1), \ldots, v_n(\tilde{Y}_n) \tag{19.74}$$

is the following collection of Horn rules. A rule corresponds to every subgoal $v_i(\tilde{Y}_i)$*, whose body is the literal* $v(X_1, \ldots, X_m)$*. The head of the rule is* $v_i(\tilde{Z}_i)$*, where* $\tilde{Z}_i$ *is obtained from* $\tilde{Y}_i$ *by preserving variables appearing in the head of rule (19.74), while function symbol* $f_Y(X_1, \ldots, X_m)$ *is written in place of every variable* Y *not appearing the head. Distinct function symbols correspond to distinct variables. The inverse of a set* $\mathcal{V}$ *of views is the set* $\{v^{-1} \colon v \in \mathcal{V}\}$*, where distinct function symbols occur in the inverses of distinct rules.*

The idea of the definition of inverses is that if a tuple $(x_1, \ldots x_m)$ appears in a view v, for some constants $x_1, \ldots x_m$, then there is a valuation of every variable y appearing in the head that makes the body of the rule true. This "unknown" valuation is denoted by the function symbol $f_Y(X_1, \ldots, X_m)$.

Example 19.12 *Inverse of views.* Let $\mathcal{V}$ be the following collection of views.

$$\begin{array}{lcl} v_1(X,Y) & \leftarrow & edge(X,Z), edge(Z,W), edge(W,Y) \\ v_2(X)) & \leftarrow & edge(X,Z)\ . \end{array} \tag{19.75}$$

Then $\mathcal{V}^{-1}$ consists of the following rules.

$$\begin{array}{lcl} edge(X, f_{1,Z}(X,Y)) & \leftarrow & v_1(X,Y) \\ edge(f_{1,Z}(X,Y), f_{1,W}(X,Y)) & \leftarrow & v_1(X,Y) \\ edge(f_{1,W}(X,Y), Y) & \leftarrow & v_1(X,Y) \\ edge(X, f_{2,Z}(X)) & \leftarrow & v_2(X)\ . \end{array} \tag{19.76}$$

Now, the maximally contained rewriting of datalog program $\mathcal{P}$ using views $\mathcal{V}$ can easily be constructed for given $\mathcal{P}$ and $\mathcal{V}$.

First, those rules are deleted from $\mathcal{P}$ that contain such *edb* relation that do not appear in the definition any view from $\mathcal{V}$. The rules of $\mathcal{V}^{-1}$ are added the datalog program $\mathcal{P}^-$ obtained so, thus forming logic program $(\mathcal{P}^-, \mathcal{V}^{-1})$. Note, that the remaining *edb* relations of $\mathcal{P}$ are *idb* relations in logic program $(\mathcal{P}^-, \mathcal{V}^{-1})$, since they

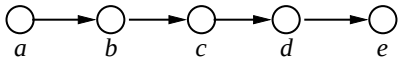


Figure 19.4. The graph G.

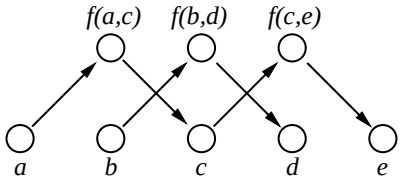


Figure 19.5. The graph G'.

appear in the heads of the rules of $\mathcal{V}^{-1}$. The names of *idb* relations are arbitrary, so they can be renamed so that their names do not coincide with the names of *edb* relations of $\mathcal{P}$. However, this is not done in the following example, for the sake of better understanding.

Example 19.13 *Logic program.* Consider the following datalog program that calculates the transitive closure of relation *edge*.

$$\begin{array}{lll} \mathcal{P}\colon & q(X,Y) & \leftarrow \quad edge(X,Y) \\ & q(X,Y) & \leftarrow \quad edge(X,Z), q(Z,Y) \end{array} \tag{19.77}$$

Assume that only the following materialised view is accessible, that stores the endpoints of paths of length two. If only this view is usable, then the most that can be expected is listing the endpoints of paths of even length. Since the unique *edb* relation of datalog program $\mathcal{P}$ is *edge*, that also appears in the definition of v, the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})$ is obtained by adding the rules of $\mathcal{V}^{-1}$ to $\mathcal{P}$.

$$\begin{array}{lll} (\mathcal{P}^-, \mathcal{V}^{-1})\colon & q(X,Y) & \leftarrow \quad edge(X,Y) \\ & q(X,Y) & \leftarrow \quad edge(X,Z), q(Z,Y) \\ & edge(X, f(X,Y)) & \leftarrow \quad v(X,Y) \\ & edge(f(X,Y),Y) & \leftarrow \quad v(X,Y)\ . \end{array} \tag{19.78}$$

Let the instance of the *edb* relation *edge* of datalog program $\mathcal{P}$ be the graph G shown on Figure 19.4. Then $(\mathcal{P}^-, \mathcal{V}^{-1})$ introduces three new constants, $f(a,c), f(b,d)$ és $f(c,e)$. The *idb* relation *edge* of logic program $\mathcal{V}^{-1}$ is graph G' shown on Figure 19.5. $\mathcal{P}^-$ computes the transitive closure of graph G'. Note that those pairs in th transitive closure that do not contain any of the new constants are exactly the endpoints of even paths of G.

The result of logic program $(\mathcal{P}^-, \mathcal{V}^{-1})$ in Example 19.13 can be calculated by procedure NAÏV-DATALOG, for example. However, it is not true for logic programs in general, that the algorithm terminates. Indeed, consider the logic program

$$\begin{array}{lll} q(X) & \leftarrow & p(X) \\ q(f(X)) & \leftarrow & q(X)\ . \end{array} \tag{19.79}$$

If the *edb* relation p contains the constant a, then the output of the program is the

infinite sequence $a, f(a), f(f(a)), f(f(f(a))), \ldots$. In contrary to this, the output of the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})$ given by the Inverse-rules Algorithm is guaranteed to be finite, thus the computation terminates in finite time.

Theorem 19.31 *For arbitrary datalog program $\mathcal{P}$ and set of conjunctive views $\mathcal{V}$, and for finite instances of the views, there exist a unique minimal fixpoint of the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})$, furthermore procedures* Naiv-Datalog *and* Semi-Naiv-Datalog *give this minimal fixpoint as output.*

The essence of the proof of Theorem 19.31 is that function symbols are only introduced by inverse rules, that are in turn not recursive, thus terms containing nested functions symbols are not produced. The details of the proof are left for the Reader (Exercise 19.3-3).

Even if started from the *edb* relations of a database, the output of a logic program may contain tuples that have function symbols. Thus, a filter is introduced that eliminates the unnecessary tuples. Let database D be the instance of the *edb*relations of datalog program $\mathcal{P}$. $\mathcal{P}(D)\downarrow$ denotes the set of those tuples from $\mathcal{P}(D)$ that do not contain function symbols. Let $\mathcal{P}\downarrow$ denote that program, which computes $\mathcal{P}(D)\downarrow$ for a given instance D. The proof of the following theorem, exceeds the limitations of the present chapter.

Theorem 19.32 *For arbitrary datalog program $\mathcal{P}$ and set of conjunctive views $\mathcal{V}$, the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$ is a maximally contained rewriting of $\mathcal{P}$ using $\mathcal{V}$. Furthermore, $(\mathcal{P}^-, \mathcal{V}^{-1})$ can be constructed in polynomial time of the sizes of $\mathcal{P}$ and $\mathcal{V}$.*

The meaning of Theorem 19.32 is that the simple procedure of adding the inverses of view definitions to a datalog program results in a logic program that uses the views as much as possible. It is easy to see that $(\mathcal{P}^-, \mathcal{V}^{-1})$ can be constructed in polynomial time of the sizes of $\mathcal{P}$ and $\mathcal{V}$, since for every subgoal $v_i \in \mathcal{V}$ a unique inverse rule must be constructed.

In order to completely solve the rewriting problem however, a datalog program needs to be produced that is equivalent with the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$. The key to this is the observation that $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$ contains only finitely many function symbols, furthermore during a bottom-up evaluation like Naiv-Datalog and its versions, nested function symbols are not produced. With proper book keeping the appearance of function symbols can be kept track, without actually producing those tuples that contain them.

The transformation is done bottom-up like in procedure Naiv-Datalog. The function symbol $f(X_1, \ldots, X_k)$ appearing in the *idb* relation of $\mathcal{V}^{-1}$ is replaced by the list of variables $X_1, \ldots, X_k$. At same time the name of the *idb* relation needs to be marked, to remember that the list $X_1, \ldots, X_k$ belongs to function symbol $f(X_1, \ldots, X_k)$. Thus, new "temporary" relation names are introduced. Consider the the rule

$$edge(X, f(X, Y)) \leftarrow v(X, Y) \tag{19.80}$$

of the logic program (19.78) in Example 19.13. It is replaced by rule

$$edge^{\langle 1, f(2,3)\rangle}(X, X, Y) \leftarrow v(X, Y) \tag{19.81}$$

Notation $\langle 1, f(2,3)\rangle$ means that the first argument of $edge^{\langle 1,f(2,3)\rangle}$ is the same as the first argument of *edge*, while the second and third arguments of $edge^{\langle 1,f(2,3)\rangle}$ together with function symbol f give the second argument of *edge*. If a function symbol would become an argument of an *idb* relation of $\mathcal{P}^-$ during the bottom-up evaluation of $(\mathcal{P}^-, \mathcal{V}^{-1})$, then a new rule is added to the program with appropriately marked relation names.

Example 19.14 *Transformation of logic program into datalog program.* The logic program Example 19.13 is transformed to the following datalog program by the procedure sketched above. The different phases of the bottom-up execution of NAIV-DATALOG are separated by lines.

$$
\begin{array}{lcl}
edge^{\langle 1,f(2,3)\rangle}(X,X,Y) & \leftarrow & v(X,Y) \\
edge^{\langle f(1,2),3\rangle}(X,Y,Y) & \leftarrow & v(X,Y) \\
\hline
q^{\langle 1,f(2,3)\rangle}(X,Y_1,Y_2) & \leftarrow & edge^{\langle 1,f(2,3)\rangle}(X,Y_1,Y_2) \\
q^{\langle f(1,2),3\rangle}(X_1,X_2,Y) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1,X_2,Y) \\
\hline
q(X,Y) & \leftarrow & edge^{\langle 1,f(2,3)\rangle}(X,Z_1,Z_2), q^{\langle f(1,2),3\rangle}(Z_1,Z_2,Y) \\
q^{\langle f(1,2),f(3,4)\rangle}(X_1,X_2,Y_1,Y_2) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1,X_2,Z), q^{\langle 1,f(2,3)\rangle}(Z,Y_1,Y_2) \\
\hline
q^{\langle f(1,2),3\rangle}(X_1,X_2,Y) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1,X_2,Z), q(Z,Y) \\
q^{\langle 1,f(2,3)\rangle}(X,Y_1,Y_2) & \leftarrow & edge^{\langle 1,f(2,3)\rangle}(X,Z_1,Z_2), q^{\langle f(1,2),f(3,4)\rangle}(Z_1,Z_2,Y_1,Y_2)
\end{array}
\tag{19.82}
$$

The datalog program obtained shows clearly that which arguments could involve function symbols in the original logic program. However, some rows containing function symbols never give tuples not containing function symbols during the evaluation of the output of the program.

Relation p is called ***significant,*** if in the precedence graph of Definition 19.16[3] there exists oriented path from p to the output relation of q. If p is not significant, then the tuples of p are not needed to compute the output of the program, thus p can be eliminated from the program.

Example 19.15 *Eliminating non-significant relations.* There exists no directed path in the precedence graph of the datalog program obtained in Example 19.14, from relations $q^{\langle 1,f(2,3)\rangle}$ and $q^{\langle f(1,2),f(3,4)\rangle}$ to the output relation q of the program, thus they are not significant, i.e., they can be eliminated together with the rules that involve them. The following datalog program is obtained:

$$
\begin{array}{lcl}
edge^{\langle 1,f(2,3)\rangle}(X,X,Y) & \leftarrow & v(X,Y) \\
edge^{\langle f(1,2),3\rangle}(X,Y,Y) & \leftarrow & v(X,Y) \\
q^{\langle f(1,2),3\rangle}(X_1,X_2,Y) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1,X_2,Y) \\
q^{\langle f(1,2),3\rangle}(X_1,X_2,Y) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1,X_2,Z), q(Z,Y) \\
q(X,Y) & \leftarrow & edge^{\langle 1,f(2,3)\rangle}(X,Z_1,Z_2), q^{\langle f(1,2),3\rangle}(Z_1,Z_2,Y)\ .
\end{array}
\tag{19.83}
$$

One more simplification step can be performed, which does not decrease the number of necessary derivations during computation of the output, however avoids redundant data copying. If p is such a relation in the datalog program that is defined

[3] Here the definition of precedence graph needs to be extended for the *edb* relations of the datalog program, as well.

by a single rule, which in turn contains a single relation in its body, then p can be removed from the program and replaced by the relation of the body of the rule defining p, having equated the variables accordingly.

Example 19.16 *Avoiding unnecessary data copying.* In Example 19.14 the relations $edge^{\langle 1,f(2,3)\rangle}$ and $edge^{\langle f(1,2),3\rangle}$ are defined by a single rule, respectively, furthermore these two rules both have a single relation in their bodies. Hence, program (19.83) can be simplified further.

$$\begin{array}{lcl} q^{\langle f(1,2),3\rangle}(X,Y,Y) & \leftarrow & v(X,Y) \\ q^{\langle f(1,2),3\rangle}(X,Z,Y) & \leftarrow & v(X,Z), q(Z,Y) \\ q(X,Y) & \leftarrow & v(X,Z), q^{\langle f(1,2),3\rangle}(X,Z,Y)\ . \end{array} \tag{19.84}$$

The datalog program obtained in the two simplification steps above is denoted by $(\mathcal{P}^-, \mathcal{V}^{-1})^{datalog}$. It is clear that there exists a one-to-one correspondence between the bottom-up evaluations of $(\mathcal{P}^-, \mathcal{V}^{-1})$ and $(\mathcal{P}^-, \mathcal{V}^{-1})^{datalog}$. Since the function symbols in $(\mathcal{P}^-, \mathcal{V}^{-1})^{datalog}$ are kept track, it is sure that the output instance obtained is in fact the subset of tuples of the output of $(\mathcal{P}^-, \mathcal{V}^{-1})$ that do not contain function symbols.

Theorem 19.33 *For arbitrary datalog program $\mathcal{P}$ that does not contain negations, and set of conjunctive views $\mathcal{V}$, the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$ is equivalent with the datalog program $(\mathcal{P}^-, \mathcal{V}^{-1})^{datalog}$.*

MiniCon

The main disadvantage of the Bucket Algorithm is that it considers each of the subgoals in isolation, therefore does not observe the most of the interactions between the subgoals of the views. Thus, the buckets may contain many unusable views, and the second phase of the algorithm may become very expensive.

The advantage of the Inverse-rules Algorithm is its conceptual simplicity and modularity. The inverses of the views must be computed only once, then they can be applied to arbitrary queries given by datalog programs. On the other hand, much of the computational advantage of exploiting the materialised views can be lost. Using the resulting rewriting produced by the algorithm for actually evaluating queries from the views has significant drawback, since it insists on recomputing the extensions of the database relations.

The MiniCon algorithm addresses the limitations of the previous two algorithms. The key idea underlying the algorithm is a change of perspective: instead of building rewritings for each of the query ***subgoals,*** it is considered how each of the ***variables*** in the query can interact with the available views. The result is that the second phase of MiniCon needs to consider drastically fewer combinations of views. In the following we return to conjunctive queries, and for the sake of easier understanding only such views are considered that do not contain constants.

The MiniCon algorithm starts out like the Bucket Algorithm, considering which views contain subgoals that correspond to subgoals in the query. However, once the algorithm finds a partial variable mapping from a subgoal g in the query to a subgoal g_1 in a view V, it changes perspective and looks at the variables in the query. The

algorithm considers the join predicates in the query – which are specified by multiple occurrences of the same variable – and finds the minimal additional set of subgoals that ***must*** be mapped to subgoals in V, given that g will be mapped to g_1. This set of subgoals and mapping information is called a ***MiniCon Description (MCD)***. In the second phase the MCDs are combined to produce query rewritings. The construction of the MCDs makes the most expensive part of the Bucket Algorithm obsolete, that is the checking of containment between the rewritings and the query, because the generating rule of MCDs makes it sure that their join gives correct result.

For a given mapping $\tau\colon Var(Q) \longrightarrow Var(V)$ subgoal g_1 of view V is said to ***cover*** a subgoal g of query Q, if $\tau(g) = g_1$. $Var(Q)$, and respectively $Var(V)$ denotes the set of variables of the query, respectively of that of the view. In order to prove that a rewriting gives only tuples that belong to the output of the query, a homomorphism must be exhibited from the query onto the rewriting. An MCD can be considered as a part of such a homomorphism, hence, these parts will be put together easily.

The rewriting of query Q is a union of conjunctive queries using the views. Some of the variables may be equated in the heads of some of the views as in the equivalent rewriting (19.70) of Example 19.10. Thus, it is useful to introduce the concept of ***head homomorphism***. The mapping $h\colon Var(V) \longrightarrow Var(V)$ is a ***head homomorphism,*** if it is an identity on variables that do not occur in the head of V, but it can equate variables of the head. For every variable x of the head of V, $h(x)$ also appear in the head of V, furthermore $h(x) = h(h(x))$. Now, the exact definition of MCD can be given.

Definition 19.34 *The quadruple $C = (h_C, V(\tilde{Y})_C, \varphi_C, G_C)$ is a **MiniCon Description (MCD)** for query Q over view V, where*

- *h_C is a head homomorphism over V,*
- *$V(\tilde{Y})_C$ is obtained from V by applying h_C, that is $\tilde{Y} = h_C(\tilde{A})$, where $\tilde{A}$ is the set of variables appearing in the head of V,*
- *φ_C is a partial mapping from $Var(Q)$ to $h_C(Var(V))$,*
- *G_C is a set of subgoals of Q that are covered by some subgoal of $H_C(V)$ using the mapping φ_C (note: not all such subgoals are necessarily included in G_C).*

The procedure constructing MCDs is based on the following proposition.

Proposition 19.35 *Let C be a MiniCon Description over view V for query Q. C can be used for a non-redundant rewriting of Q if the following conditions hold*

C1. *for every variable x that is in the head of Q and is in the domain of φ_C, as well, $\varphi_C(x)$ appears in the head of $h_C(V)$, furthermore*

C2. *if $\varphi_C(y)$ does not appear in the head of $h_C(V)$, then for all such subgoals of Q that contain y holds that*

1. *every variable of g appears in the domain of φ_C and*
2. *$\varphi_C(g) \in h_C(V)$.*

Clause **C1** is the same as in the Bucket Algorithm. Clause **C2** means that if a variable x is part of a join predicate which is not enforced by the view, then x must be in the head of the view so the join predicate can be applied by another subgoal in the rewriting. The procedure FORM-MCDS gives the usable MiniCon Descriptions for a conjunctive query Q and set of conjunctive views $\mathcal{V}$.

FORM-MCDS$(Q, \mathcal{V})$

```
1  C ← ∅
2  for each subgoal g of Q
3      do for V ∈ V
4          do for every subgoal v ∈ V
5              do Let h be the least restrictive head homomorphism on V,
                  such that there exists a mapping φ with φ(g) = h(v).
6                 if φ and h exist
7                    then Add to C any new MCD C,
                          that can be constructed where:
8                         (a) φ_C (respectively, h_C) is
                              an extension of φ (respectively, h),
9                         (b) G_C is the minimal subset of subgoals of Q such that
                              G_C, φ_C and h_C satisfy Proposition 19.35, and
10                        (c) It is not possible to extend φ and h to φ'_C
                              and h'_C such that (b) is satisfied,
                              and G'_C as defined in (b), is a subset of G_C.
11 return C
```

Consider again query (19.68) and the views of Example 19.10. Procedure FORM-MCDS considers subgoal $cite(x, y)$ of the query first. It does not create an MCD for view V_1, because clause **C2** of Proposition 19.35 would be violated. Indeed, the condition would require that subgoal $sameArea(x, y)$ be also covered by V_1 using the mapping $\varphi(x) = a$, $\varphi(y) = b$, since is not in the head of V_1.[4] For the same reason, no MCD will be created for V_1 even when the other subgoals of the query are considered. In a sense, the MiniCon Algorithm shifts some of the work done by the combination step of the Bucket Algorithm to the phase of creating the MCDs by using FORM-MCDS. The following table shows the output of procedure FORM-MCDS.

$V(\tilde{Y})$	h	φ	G
$V_2(c, d)$	$c \to c, d \to d$	$x \to c, y \to d$	3
$V_3(f, f)$	$f \to f, h \to f$	$x \to f, y \to f$	$1, 2, 3$

(19.85)

Procedure FORM-MCDS includes in G_C only the ***minimal*** set of subgoals that are necessary in order to satisfy Proposition 19.35. This makes it possible that in the second phase of the MiniCon Algorithm needs only to consider combinations of MCDs that cover ***pairwise disjoint subsets*** of subgoals of the query.

[4] The case of $\varphi(x) = b$, $\varphi(y) = a$ is similar.

Proposition 19.36 *Given a query Q, a set of views $\mathcal{V}$, and the set of MCDs $\mathcal{C}$ for Q over the views $\mathcal{V}$, the only combinations of MCDs that can result in non-redundant rewritings of Q are of the form $C_1, \ldots C_l$, where*

C3. *$G_{C_1} \cup \cdots \cup G_{C_l}$ contains all the subgoals of Q, and*

C4. *for every $i \neq j$ $G_{C_i} \cap G_{C_j} = \emptyset$.*

The fact that only such sets of MCDs need to be considered that provide partitions of the subgoals in the query reduces the search space of the algorithm drastically. In order to formulate procedure COMBINE-MCDS, another notation needs to be introduced. The φ_C mapping of MCD C may map a set of variables of Q onto the same variable of $h_C(V)$. One arbitrarily chosen representative of this set is chosen, with the only restriction that if there exists variables in this set from the head of Q, then one of those is the chosen one. Let $EC_{\varphi_C}(x)$ denote the representative variable of the set containing x. The MiniCon Description C is considered extended with $EC_{\varphi_C}(x)$ in he following as a quintet $(h_C, V(\tilde{Y}), \varphi_C, G_C, EC_{\varphi_C})$. If the MCDs $C_1, \ldots, C_k$ are to be combined, and for some $i \neq j$ $EC_{\varphi_{C_i}}(x) = EC_{\varphi_{C_i}}(y)$ and $EC_{\varphi_{C_j}}(y) = EC_{\varphi_{C_j}}(z)$ holds, then in the conjunctive rewriting obtained by the join x, y and z will be mapped to the same variable. Let S_C denote the equivalence relation determined on the variables of Q by two variables being equivalent if they are mapped onto the same variable by φ_C, that is, $xS_Cy \iff EC_{\varphi_C}(x) = EC_{\varphi_C}(y)$. Let $\mathcal{C}$ be the set of MCDs obtained as the output of FORM-MCDS.

COMBINE-MCDS($\mathcal{C}$)

```
1  Answer ← ∅
2  for {C_1, ..., C_n} ⊆ C such that G_{C_1}, ..., G_{C_n} is a partition of the subgoals of Q
3      do Define a mapping Ψ_i on Ỹ_i as follows:
4          if there exists a variable x in Q such that φ_i(x) = y
5             then Ψ_i(y) = x
6             else  Ψ_i(y) is a fresh copy of y
7      Let S be the transitive closure of S_{C_1} ∪ ··· ∪ S_{C_n}
8                                          ▷ S is an equivalence relation of variables of Q.
9      Choose a representative for each equivalence class of S.
10     Define mapping EC as follows:
11     if x ∈ Var(Q)
12        then EC(x) is the representative of the equivalence class of x under S
13        else  EC(x) = x
14     Let Q' be given as Q'(EC(X̃)) ← V_{C_1}(EC(Ψ_1(Ỹ_1))), ..., V_{C_n}(EC(Ψ_n(Ỹ_n)))
15     Answer ← Answer ∪ {Q'}
16 return Answer
```

The following theorem summarises the properties of the MiniCon Algorithm.

Theorem 19.37 *Given a conjunctive query Q and conjunctive views $\mathcal{V}$, both without comparison predicates and constants, the MiniCon Algorithm produces the union of conjunctive queries that is a maximally contained rewriting of Q using $\mathcal{V}$.*

The complete proof of Theorem 19.37 exceeds the limitations of the present chapter. However, in Problem *19-1* the reader is asked to prove that union of the conjunctive queries obtained as output of COMBINE-MCDS is contained in Q.

It must be noted that the running times of the Bucket Algorithm, the Inverse-rules Algorithm and the MiniCon Algorithm are the same in the worst case: $O(nmM^n)$, where n is the number of subgoals in the query, m is the maximal number of subgoals in a view, and M is the number of views. However, practical test runs show that in case of large number of views (3–400 views) the MiniCon Algorithm is significantly faster than the other two.

Exercises

19.3-1 Prove Theorem 19.25 using Proposition 19.24 and Theorem 19.20.

19.3-2 Prove the two statements of Lemma 19.26. *Hint.* For the first statement, write in their definitions in place of views $v_i(\tilde{Y}_i)$ into Q'. Minimise the obtained query Q'' using Theorem 19.19. For the second statement use Proposition 19.24 to prove that there exists a homomorphism h_i from the body of the conjunctive query defining view $v_i(\tilde{Y}_i)$ to the body of Q. Show that $\tilde{Y'}_i = h_i(\tilde{Y}_i)$ is a good choice.

19.3-3 Prove Theorem 19.31 using that datalog programs have unique minimal fixpoint.

Problems

19-1 MiniCon is correct

Prove that the output of the MiniCon Algorithm is correct. *Hint.* It is enough to show that for each conjunctive query Q' given in line 14 of COMBINE-MCDS $Q' \sqsubseteq Q$ holds. For the latter, construct a homomorphism from Q to Q'.

19-2 $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$ is correct

Prove that each tuple produced by logic program $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$ is contained in the output of $\mathcal{P}$ (part of the proof of Theorem 19.32). *Hint.* Let t be a tuple in the output of $(\mathcal{P}^-, \mathcal{V}^{-1})$ that does not contain function symbols. Consider the derivation tree of t. Its leaves are literals, since they are extensional relations of program $(\mathcal{P}^-, \mathcal{V}^{-1})$. If these leaves are removed from the tree, then the leaves of the remaining tree are *edb* relations of $\mathcal{P}$. Prove that the tree obtained is the derivation tree of t in datalog program $\mathcal{P}$.

19-3 Datalog views

This problem tries to justify why only conjunctive views were considered. Let $\mathcal{V}$ be a set of views, Q be a query. For a given instance $\mathcal{I}$ of the views the tuple t is a ***certain answer*** of query Q, if for any database instance $\mathcal{D}$ such that $\mathcal{I} \subseteq \mathcal{V}(\mathcal{D})$, $t \in Q(\mathcal{D})$ holds, as well.

a. Prove that if the views of $\mathcal{V}$ are given by datalog programs, query Q is conjunctive and may contain non-equality ($\neq$) predicates, then the question whether for a given instance $\mathcal{I}$ of the views tuple t is a certain answer of Q is algorithmically undecidable. *Hint.* Reduce to this question the ***Post Correspondence Problem,*** which is the following: Given two sets of words $\{w_1, w_2, \ldots, w_n\}$ and

$\{w'_1, w'_2, \ldots, w'_n\}$ over the alphabet $\{a, b\}$. The question is whether there exists a sequence of indices $i_1, i_2, \ldots, i_k$ (repetition allowed) such that

$$w_{i_1} w_{i_2} \cdots w_{i_k} = w'_{i_1} w'_{i_2} \cdots w'_{i_k} \ . \tag{19.86}$$

The Post Correspondence Problem is well known algorithmically undecidable problem. Let the view V be given by the following datalog program:

$$\begin{array}{lcl} V(0,0) & \leftarrow & S(e,e,e) \\ V(X,Y) & \leftarrow & V(X_0,Y_0), S(X_0,X_1,\alpha_1), \ldots, S(X_{g-1},Y,\alpha_g), \\ & & S(Y_0,Y_1,\beta_1), \ldots, S(Y_{h-1},Y,\beta_h) \\ & & \text{where } w_i = \alpha_1 \ldots \alpha_g \text{ and } \ w'_i = \beta_1 \ldots \beta_h \\ & & \text{is a rule for all } i \in \{1,2,\ldots,n\} \\ S(X,Y,Z) & \leftarrow & P(X,X,Y), P(X,Y,Z) \ . \end{array} \tag{19.87}$$

Furthermore, let Q be the following conjunctive query.

$$Q(c) \ \leftarrow \ P(X,Y,Z), P(X,Y,Z'), Z \neq Z' \ . \tag{19.88}$$

Show that for the instance $\mathcal{I}$ of V that is given by $\mathcal{I}(V) = \{\langle e,e\rangle\}$ and $\mathcal{I}(S) = \{\}$, the tuple $\langle c\rangle$ is a certain answer of query Q if and only if the Post Correspondence Problem with sets $\{w_1, w_2, \ldots, w_n\}$ and $\{w'_1, w'_2, \ldots, w'_n\}$ has *no* solution.

b. In contrast to the undecidability result of *a.*, if $\mathcal{V}$ is a set of conjunctive views and query Q is given by datalog program $\mathcal{P}$, then it is easy to decide about an arbitrary tuple t whether it is a certain answer of Q for a given view instance $\mathcal{I}$. Prove that the datalog program $(\mathcal{P}^-, \mathcal{V}^{-1})^{datalog}$ gives exactly the tuples of the certain answer of Q as output.

Chapter notes

There are several dimensions along which the treatments of the problem "answering queries using views" can be classified. Figure 19.6 shows the taxonomy of the work.

The most significant distinction between the different work s is whether their goal is data integration or whether it is query optimisation and maintenance of physical data independence. The key difference between these two classes of works is the output of the the algorithm for answering queries using views. In the former case, given a query Q and a set of views $\mathcal{V}$, the goal of the algorithm is to produce an expression Q' that references the views and is either equivalent to or contained in Q. In the latter case, the algorithm must go further and produce a (hopefully optimal) query execution plan for answering Q using the views (and possibly the database relations). Here the rewriting must be an equivalent to Q in order to ensure the correctness of the plan.

The similarity between these two bodies of work is that they are concerned with the core issue of whether a rewriting of a query is equivalent or contained in the query. However, while logical correctness suffices for the data integration context, it does not in the query optimisation context where we also need to find the *cheapest* plan

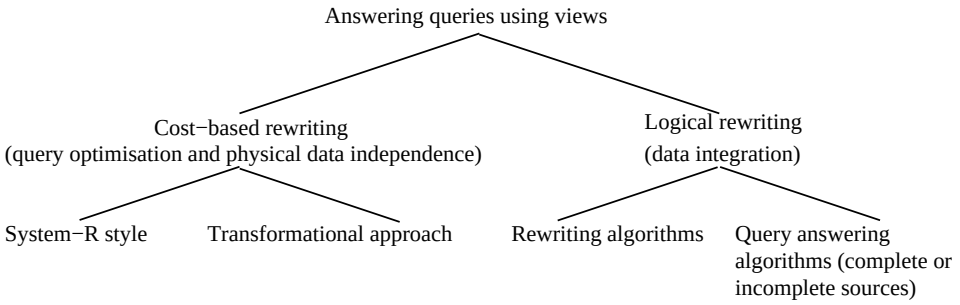


Figure 19.6. A taxonomy of work on answering queries using views.

using the views. The complication arises because the optimisation algorithms need to consider views that do not contribute to the *logical* correctness of the rewriting, but do reduce the cost of the resulting plan. Hence, while the reasoning underlying the algorithms in the data integration context is mostly logical, in the query optimisation case it is both logical and cost-based. On the other hand, an aspect stressed in data integration context is the importance of dealing with a large number of views, which correspond to data sources. In the context of query optimisation it is generally assumed (not always!) that the number of views is roughly comparable to the size of the schema.

The works on query optimisation can be classified further into System-R style optimisers and transformational optimisers. Examples of the former are works of Chaudhuri, Krishnamurty, Potomianos and Shim [61]; Tsatalos, Solomon, and Ioannidis [326]. Papers of Florescu, Raschid, and Valduriez [111]; Bello et. al. [36]; Deutsch, Popa and Tannen [89], Zaharioudakis et. al. [353], furthermore Goldstein és Larson[135] belong to the latter.

Rewriting algorithms in the data integration context are studied in works of Yang and Larson [349]; Levy, Mendelzon, Sagiv and Srivastava [155]; Qian [279]; furthermore Lambrecht, Kambhampati and Gnanaprakasam [207]. The Bucket Algorithm was introduced by Levy, Rajaraman and Ordille [152]. The Inverse-rules Algorithm is invented by Duschka and Genesereth [94, 93]. The MiniCon Algorithm was developed by Pottinger and Halevy [276, 275].

Query answering algorithms and the complexity of the problem is studied in papers of Abiteboul and Duschka [2]; Grahne and Mendelzon [139]; furthermore Calvanese, De Giacomo, Lenzerini and Vardi [55].

The STORED system was developed by Deutsch, Fernandez and Suciu [88]. Semantic caching is discussed in the paper of Yang, Karlapalem and Li [350]. Extensions of the rewriting problem are studied in [50, 112, 120, 205, 350].

Surveys of the area can be found in works of Abiteboul [1], Florescu, Levy and Mendelzon [110], Halevy [153, 154], furthermore Ullman[333].

Research of the authors was (partially) supported by Hungarian National Research Fund (OTKA) grants Nos. T034702, T037846T and T042706.

20. Semi-structured Databases

The use of the internet and the development of the theory of databases mutually affect each other. The contents of web sites are usually stored by databases, while the web sites and the references between them can also be considered a database which has no fixed schema in the usual sense. The contents of the sites and the references between sites are described by the sites themselves, therefore we can only speak of semi-structured data, which can be best characterized by directed labeled graphs. In case of semi-structured data, recursive methods are used more often for giving data structures and queries than in case of classical relational databases. Different problems of databases, e.g. restrictions, dependencies, queries, distributed storage, authorities, uncertainty handling, must all be generalized according to this. Semi-structuredness also raises new questions. Since queries not always form a closed system like they do in case of classical databases, that is, the applicability of queries one after another depends on the type of the result obtained, therefore the problem of checking types becomes more emphasized.

The theoretical establishment of relational databases is closely related to finite modelling theory, while in case of semi-structured databases, automata, especially tree automata are most important.

20.1. Semi-structured data and XML

By semi-structured data we mean a directed rooted labeled graph. The root is a special node of the graph with no entering edges. The nodes of the graph are objects distinguished from each other using labels. The objects are either atomic or complex. Complex objects are connected to one or more objects by directed edges. Values are assigned to atomic objects. Two different models are used: either the vertices or the edges are labeled. The latter one is more general, since an edge-labeled graph can be assigned to all vertex-labeled graphs in such a way that the label assigned to the edge is the label assigned to its endpoint. This way we obtain a directed labeled graph for which all inward directed edges from a vertex have the same label. Using this transformation, all concepts, definitions and statements concerning edge-labeled graphs can be rewritten for vertex-labeled graphs.

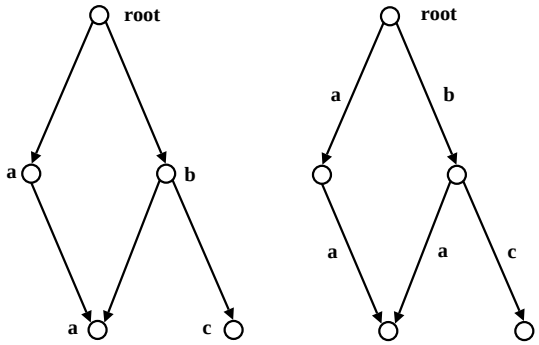


Figure 20.1. Edge-labeled graph assigned to a vertex-labeled graph.

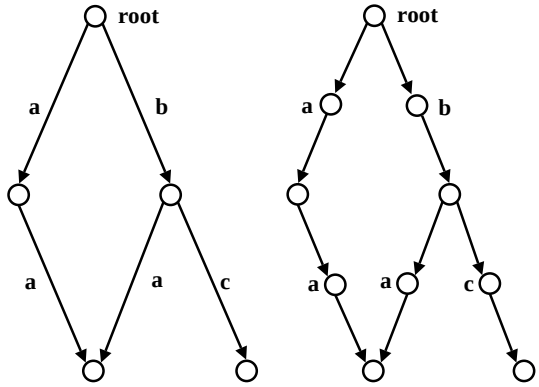


Figure 20.2. An edge-labeled graph and the corresponding vertex-labeled graph.

The following method is used to gain a vertex-labeled graph from an edge-labeled graph. If edge (u, v) has label c, then remove this edge, and introduce a new vertex w with label c, then add edges (u, w) and (w, v). This way we can obtain a vertex-labeled graph of $m+n$ nodes and $2m$ edges from an edge-labeled graph of n vertices and m edges. Therefore all algorithms and cost bounds concerning vertex-labeled graphs can be rewritten for edge-labeled graphs.

Since most books used in practice use vertex-labeled graphs, we will also use vertex-labeled graphs in this chapter.

The ***XML*** (e**X**tensible **M**arkup **L**anguage) language was originally designed to describe embedded ordered labeled elements, therefore it can be used to represent trees of semi-structured data. In a wider sense of the XML language, references between the elements can also be given, thus arbitrary semi-structured data can be described using the XML language.

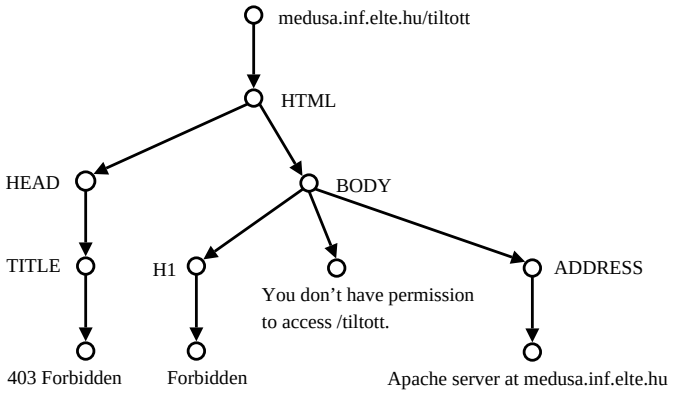


Figure 20.3. The graph corresponding to the XML file "forbidden".

The medusa.inf.elte.hu/forbidden site written in XML language is as follows. We can obtain the vertex-labeled graph of Figure 20.3 naturally from the structural characteristics of the code.

```
<HTML>
        <HEAD>
                <TITLE>403 Forbidden</TITLE>
        </HEAD>
        <BODY>
                <H1>Forbidden</H1>
                You don't have permission to access /forbidden.
                <ADDRESS>Apache Server at medusa.inf.elte.hu </ADDRESS>
        </BODY>
</HTML>
```

Exercises

20.1-1 Give a vertex-labeled graph that represents the structure and formatting of this chapter.

20.1-2 How many different directed vertex-labeled graphs exist with n vertices, m edges and k possible labels? How many of these graphs are not isomorphic? What values can be obtained for $n = 5$, $m = 7$ and $k = 2$?

20.1-3 Consider a tree in which all children of a given node are labeled with different numbers. Prove that the nodes can be labeled with pairs (a_v, b_v), where a_v and b_v are natural numbers, in such a way that

a. $a_v < b_v$ for every node v.

b. If u is a descendant of v, then $a_v < a_u < b_u < b_v$.

c. If u and v are siblings and $number(u) < number(v)$, then $b_u < a_v$.

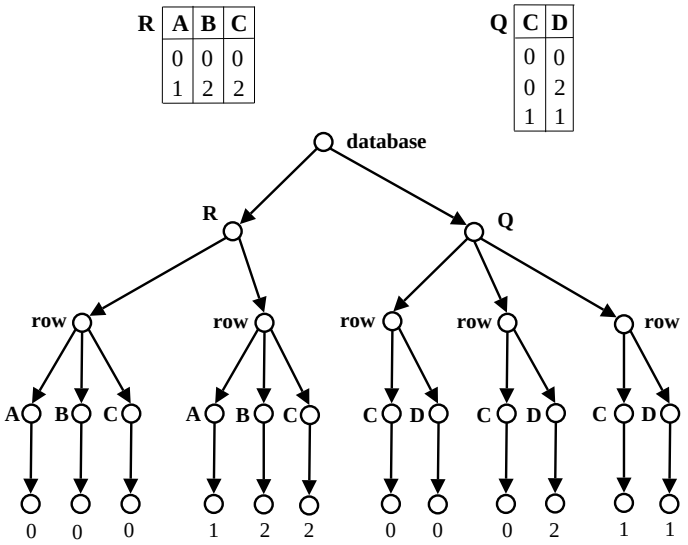


Figure 20.4. A relational database in the semi-structured model.

20.2. Schemas and simulations

In case of relational databases, schemas play an important role in coding and querying data, query optimization and storing methods that increase efficiency. When working with semi-structured databases, the schema must be obtained from the graph. The schema restricts the possible label strings belonging to the paths of the graph.

Figure 20.4 shows the relational schemas with relations $R(A, B, C)$ and $Q(C, D)$, respectively, and the corresponding semi-structured description. The labels of the leaves of the tree are the components of the tuples. The directed paths leading from the root to the values contain the label strings *database.R.tuple.A, database.R.tuple.B, database.R.tuple.C, database.Q.tuple.C, database.Q.tuple.D.* This can be considered the schema of the semi-structured database. Note that the schema is also a graph, as it can be seen on Figure 20.5. The disjoint union of the two graphs is also a graph, on which a simulation mapping can be defined as follows. This way we create a connection between the original graph and the graph corresponding to the schema.

Definition 20.1 *Let $G = (V, E, A, label())$ be a vertex-labeled directed graph, where V denotes the set of nodes, E the set of edges, A the set of labels, and $label(v)$ is the label belonging to node v. Denote by $E^{-1}(v) = \{u \mid (u, v) \in E\}$ the set of the start nodes of the edges leading to node v. A binary relation s ($s \subseteq V \times V$) is a* ***simulation****, if, for $s(u, v)$,*
i) $label(u) = label(v)$ and
ii) for all $u' \in E^{-1}(u)$ there exists a $v' \in E^{-1}(v)$ such that $s(u', v')$
Node v simulates node u*, if there exists a simulation s such that $s(u, v)$.* ***Node u and node v are similar****, $u \approx v$, if u simulates v and v simulates u.*

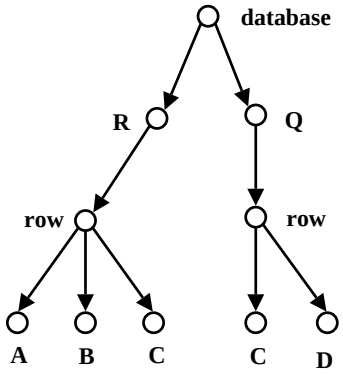


Figure 20.5. The schema of the semi-structured database given in Figure 20.4.

It is easy to see that the empty relation is a simulation, that the union of simulations is a simulation, that there always exists a maximal simulation and that similarity is an equivalence relation. We can write E instead of E^{-1} in the above definition, since that only means that the direction of the edges of the graph is reversed.

We say that graph D simulates graph S if there exists a mapping $f : V_S \mapsto V_D$ such that the relation $(v, f(v))$ is a simulation on the set $V_S \times V_D$.

Two different schemas are used, a lower bound and an upper bound. If the data graph D simulates the schema graph S, then ***S is a lower bound of D***. Note that this means that all label strings belonging to the directed paths in S appear in D at some directed path. If S simulates D, then ***S is an upper bound of D***. In this case, the label strings of D also appear in S.

In case of semi-structured databases, the schemas which are greatest lower bounds or lowest upper bounds play an important role.

A map between graphs S and D that preserves edges is called a morphism. Note that f is a morphism if and only if D simulates S. To determine whether a morphism from D to S exists is an NP-complete problem. We will see below, however, that the calculation of a maximal simulation is a PTIME problem.

Denote by $sim(v)$ the nodes that simulate v. The calculation of the maximal simulation is equivalent to the determination of all sets $sim(v)$ for $v \in V$. First, our naive calculation will be based on the definition.

NAIVE-MAXIMAL-SIMULATION(G)

1 **for** all $v \in V$
2 **do** $sim(v) \leftarrow \{u \in V \mid label(u) = label(v)\}$
3 **while** $\exists\, u, v, w \in V :\ v \in E^{-1}(u)\ \wedge\ w \in sim(u)\ \wedge\ E^{-1}(w) \cap sim(v) = \emptyset$
4 **do** $sim(u) \leftarrow sim(u) \setminus \{w\}$
5 **return** $\{sim(u) \mid u \in V\}$

Proposition 20.2 *The algorithm* NAIVE-MAXIMAL-SIMULATION *computes the maximal simulation in* $O(m^2n^3)$ *time if* $m \geq n$.

Proof Let us start with the elements of $sim(u)$. If an element w of $sim(u)$ does not simulate u by definition according to edge (v, u), then we remove w from set $sim(u)$. In this case, we say that we improved set $sim(u)$ according to edge (v, u). If set $sim(u)$ cannot be improved according to any of the edges, then all elements of $sim(u)$ simulate u. To complete the proof, notice that the **while** cycle consists of at most n^2 iterations. ■

The efficiency of the algorithm can be improved using special data structures. First, introduce a set *sim-candidate(u)*, which contains $sim(u)$, and of the elements of whom we want to find out whether they simulate u.

IMPROVED-MAXIMAL-SIMULATION(G)

```
1  for all v ∈ V
2      do sim-candidate(u) ← V
3         if E^{-1}(v) = ∅
4            then sim(v) ← {u ∈ V | label(u) = label(v)}
5            else sim(v) ← {u ∈ V | label(u) = label(v) ∧ E^{-1}(u) ≠ ∅}
6  while ∃ v ∈ V : sim(v) ≠ sim-candidate(v)
7        do removal-candidate ← E(sim-candidate(v)) \ E(sim(v))
8           for all u ∈ E(v)
9               do sim(u) ← sim(u) \ removal-candidate
10          sim-candidate(v) ← sim(v)
11 return {sim(u) | u ∈ V}
```

The **while** cycle of the improved algorithm possesses the following invariant characteristics.

I_1: $\forall\, v \in V :\ sim(v) \subseteq$ *sim-candidate*(v).
I_2: $\forall\, u, v, w \in V : (v \in E^{-1}(u) \wedge w \in sim(u)) \ \Rightarrow\ (E^{-1}(w) \cap$ *sim-candidate*$(v) \neq \emptyset)$.

When improving the set $sim(u)$ according to edge (v, u), we check whether an element $w \in sim(u)$ has parents in $sim(v)$. It is sufficient to check that for the elements of *sim-candidate*(v) instead of $sim(v)$ because of I_2. Once an element $w' \in$ *sim-candidate*$(v) \setminus sim(v)$ was chosen, it is removed from set *sim-candidate*(v).

We can further improve the algorithm if we do not compute the set *removal-candidate* in the iterations of the **while** cycle but refresh the set dynamically.

EFFICIENT-MAXIMAL-SIMULATION(G)

```
 1 for all v ∈ V
 2     do sim-candidate(v) ← V
 3        if E^{-1}(v) = ∅
 4           then sim(v) ← {u ∈ V | label(u) = label(v)}
 5           else  sim(v) ← {u ∈ V | label(u) = label(v)  ∧  E^{-1}(u) ≠ ∅}
 6     removal-candidate(v) ← E(V) \ E(sim(v))
 7  while ∃ v ∈ V  :  removal-candidate(v) ≠ ∅
 8        do for all u ∈ E(v)
 9           do for all w ∈ removal-candidate(v)
10              do if w ∈ sim(u)
11                    then sim(u) ← sim(u) \ {w}
12                         for all w'' ∈ E(w)
13                             do if E^{-1}(w'') ∩ sim(u) = ∅
14                                   then removal-candidate(u)
                                          ← removal-candidate(u) ∪ {w''}
15           sim-candidate(v) ← sim(v)
16           removal-candidate(v) ← ∅
17 return {sim(u) | u ∈ V}
```

The above algorithm possesses the following invariant characteristic with respect to the **while** cycle.

I_3: $\forall\ v \in V$: *removal-candidate*$(v) = E($*sim-candidate*$(v)) \setminus E($*sim*$(v))$.

Use an $n \times n$ array as a *counter* for the realization of the algorithm. Let the value *counter*$[w'', u]$ be the nonnegative integer $|E^{-1}(w'') \cap sim(u)|$. The initial values of the counter are set in $O(mn)$ time. When element w is removed from set $sim(u)$, the values *counter*$[w'', u]$ must be decreased for all children w'' of w. By this we ensure that the innermost **if** condition can be checked in constant time. At the beginning of the algorithm, the initial values of the sets $sim(v)$ are set in $O(n^2)$ time if $m \geq n$. The setting of sets *removal-candidate*(v) takes altogether $O(mn)$ time. For arbitrary nodes v and w, if $w \in$ *removal-candidate*(v) is true in the i-th iteration of the **while** cycle, then it will be false in the j-th iteration for $j > i$. Since $w \in$ *removal-candidate*(v) implies $w \notin E(sim(v))$, the value of *sim-candidate*(v) in the j-th iteration is a subset of the value of $sim(v)$ in the i-th iteration, and we know that invariant I_3 holds. Therefore $w \in sim(u)$ can be checked in $\sum_v \sum_w |E(v)| = O(mn)$ time. $w \in sim(u)$ is true at most once for all nodes w and u, since once the condition holds, we remove w from set $sim(u)$. This implies that the computation of the outer **if** condition of the **while** cycle takes $\sum_v \sum_w (1 + |E(v)|) = O(mn)$ time.

Thus we have proved the following proposition.

Proposition 20.3 *The algorithm* EFFECTIVE-MAXIMAL-SIMULATION *computes the maximal simulation in $O(mn)$ time if $m \geq n$.*

If the inverse of a simulation is also a simulation, then it is called a bisimulation. The empty relation is a bisimulation, and there always exist a maximal bisimulation.

The maximal bisimulation can be computed more efficiently than the simulation. The maximal bisimulation can be computed in $O(m \lg n)$ time using the PT algorithm. In case of edge-labeled graphs, the cost is $O(m \lg(m+n))$.

We will see that bisimulations play an important role in indexing semi-structured databases, since the quotient graph of a graph with respect to a bisimulation contains the same label strings as the original graph. Note that in practice, instead of simulations, the so-called ***DTD*** descriptions are also used as schemas. DTD consists of data type definitions formulated in regular language.

Exercises

20.2-1 Show that simulation does not imply bisimulation.

20.2-2 Define the operation *turn-tree* for a directed, not necessarily acyclic, vertex-labeled graph G the following way. The result of the operation is a not necessarily finite graph G', the vertices of which are the directed paths of G starting from the root, and the labels of the paths are the corresponding label strings. Connect node p_1 with node p_2 by an edge if p_1 can be obtained from p_2 by deletion of its endpoint. Prove that G and *turn-tree*(G) are similar with respect to the bisimulation.

20.3. Queries and indexes

The information stored in semi-structured databases can be retrieved using queries. For this, we have to fix the form of the questions, so we give a ***query language***, and then define the meaning of questions, that is, the ***query evaluation*** with respect to a semi-structured database. For efficient evaluation we usually use indexes. The main idea of ***indexing*** is that we reduce the data stored in the database according to some similarity principle, that is, we create an index that reflects the structure of the original data. The original query is executed in the index, then using the result we find the data corresponding to the index values in the original database. The size of the index is usually much smaller than that of the original database, therefore queries can be executed faster. Note that the inverted list type index used in case of classical databases can be integrated with the schema type indexes introduced below. This is especially advantageous when searching XML documents using keywords.

First we will get acquainted with the query language consisting of regular expressions and the index types used with it.

Definition 20.4 *Given a directed vertex-labeled graph* $G = (V, E, root, \Sigma, label, id, value)$*, where* V *denotes the set of vertices,* $E \subseteq V \times V$ *the set of edges and* Σ *the set of labels.* Σ *contains two special labels, ROOT and VALUE. The label of vertex* v *is* $label(v)$*, and the identifier of vertex* v *is* $id(v)$*. The root is a node with label ROOT, and from which all nodes can be reached via directed paths. If* v *is a leaf, that is, if it has no outgoing edges, then its label is VALUE, and* $value(v)$ *is the value corresponding to leaf* v*. Under the term path we always mean a directed path, that is, a sequence of nodes* $n_0, \ldots, n_p$ *such that there is an edge from* n_i *to* n_{i+1} *if* $0 \leq i \leq p-1$*. A sequence of labels* $l_0, \ldots, l_p$ *is called a* ***label sequence*** *or* ***simple expression****.* ***Path*** $\boldsymbol{n_0, \ldots, n_p}$ ***fits to the label sequence*** $\boldsymbol{l_0, \ldots, l_p}$ *if* $label(n_i) = l_i$ *for all* $0 \leq i \leq p$*.*

We define regular expressions recursively.

Definition 20.5 *Let $R ::= \varepsilon \mid \Sigma \mid _ \mid R.R \mid R|R \mid (R) \mid R? \mid R^*$, where **R** is a **regular expression**, and ε is the empty expression, _ denotes an arbitrary label, . denotes succession, | is the logical OR operation, ? is the optional choice, and * means finite repetition. Denote by **L(R)** the regular language consisting of the label sequences determined by R. **Node n fits to a label sequence** if there exists a path from the root to node n such that fits to the label sequence. **Node n fits to the regular expression** R if there exists a label sequence in the language L(R), to which node n fits. The **result of the query on graph G** determined by the regular expression R is the set R(G) of nodes that fit to expression R.*

Since we are always looking for paths starting from the root when evaluating regular expressions, the first element of the label sequence is always ROOT, which can therefore be omitted.

Note that the set of languages $L(R)$ corresponding to regular expressions is closed under intersection, and the problem whether $L(R) = \emptyset$ is decidable.

The result of the queries can be computed using the nondeterministic automaton A_R corresponding to the regular expression R. The algorithm given recursively is as follows.

NAIVE-EVALUATION(G, A_R)

1 *Visited* $\leftarrow \emptyset$ ▷ If we were in node u in state s, then (u, s) was put in set *Visited*.
2 TRAVERSE (*root*(G), *starting-state*(A_R))

TRAVERSE(u, s)

1 **if** $(u, s) \in$ *Visited*
2 **then return** *result*$[u, s]$
3 *Visited* $\leftarrow$ *Visited* $\cup \{(u, s)\}$
4 *result*$[u, s] \leftarrow \emptyset$
5 **for** all $s \xrightarrow{\epsilon} s'$ ▷ If we get to state s' from state s by reading sign ϵ.
6 **do if** $s' \in$ *final-state*(A_R)
7 **then** *result*$[u, s] \leftarrow \{u\} \cup$ *result*$[u, s]$
8 *result*$[u, s] \leftarrow$ *result*$[u, s] \cup$ TRAVERSE(u, s')
9 **for** all $s \xrightarrow{label(u)} s'$ ▷ If we get to state s' from state s by reading sign *label*(u).
10 **do if** $s' \in$ *final-state*(A_R)
11 **then** *result*$[u, s] \leftarrow \{u\} \cup$ *result*$[u, s]$
12 **for** all v, where $(u, v) \in E(G)$ ▷ Continue the traversal for the children of node u recursively.
13 **do** *result*$[u, s] \leftarrow$ *result*$[u, s] \cup$ TRAVERSE(v, s')
14 **return** *result*$[u, s]$

Proposition 20.6 *Given a regular query R and a graph G, the calculation cost of R(G) is a polynomial of the number of edges of G and the number of different states of the finite nondeterministic automaton corresponding to R.*

Proof The sketch of the proof is the following. Let A_R be the finite nondeterministic automaton corresponding to R. Denote by $|A_R|$ the number of states of A_R. Consider the breadth-first traversal corresponding to the algorithm TRAVERSE of graph G with m edges, starting from the root. During the traversal we get to a new state of the automaton according to the label of the node, and we store the state reached at the node for each node. If the final state of the automaton is acceptance, then the node is a result. During the traversal, we sometimes have to step back on an edge to ensure we continue to places we have not seen yet. It can be proved that during a traversal every edge is used at most once in every state, so this is the number of steps performed by that automaton. This means $O(|A_R|m)$ steps altogether, which completes the proof. ■

Two nodes of graph G are ***indistinguishable with regular expressions*** if there is no regular R for which one of the nodes is among the results and the other node is not. Of course, if two nodes cannot be distinguished, then their labels are the same. Let us categorize the nodes in such a way that nodes with the same label are in the same class. This way we produce a partition P of the set of nodes, which is called the ***basic partition***. It can also be seen easily that if two nodes are indistinguishable, then it is also true for the parents. This implies that the set of label sequences corresponding to paths from the root to the indistinguishable nodes is the same. Let $L(n) = \{l_0, \ldots, l_p \mid n$ fits to the label sequence $l_0, \ldots, l_p\}$ for all nodes n. Nodes n_1 and n_2 are indistinguishable if and only if $L(n_1) = L(n_2)$. If the nodes are assigned to classes in such a way that the nodes having the same value $L(n)$ are arranged to the same class, then we get a refinement P' of partition P. For this new partition, if a node n is among the results of a regular query R, then all nodes from the equivalence class of n are also among the results of the query.

Definition 20.7 *Given a graph $G = (V, E, root, \Sigma, label, id, value)$ and a partition P of V that is a refinement of the basic partition, that is, for which the nodes belonging to the same equivalence class have the same label. Then the graph $I(G) = (P, E', root', \Sigma, label', id', value')$ is called an* ***index****. The nodes of the index graph are the equivalence classes of partition P, and $(I, J) \in E'$ if and only if there exist $i \in I$ and $j \in J$ such that $(i, j) \in E$. If $I \in P$, then $id'(I)$ is the identifier of index node I, $label'(I) = label(n)$, where $n \in I$, and root' is the equivalence class of partition P that contains the root of G. If $label(I) = VALUE$, then $label'(I) = \{value(n) \mid n \in I\}$.*

Given a partition P of set V, denote by ***class(n)*** the equivalence class of P that contains n for $n \in V$. In case of indexes, the notation $I(n)$ can also be used instead of $class(n)$.

Note that basically the indexes can be identified with the different partitions of the nodes, so partitions can also be called indexes without causing confusion. Those indexes will be good that are of small size and for which the result of queries is the same on the graph and on the index. Indexes are usually given by an equivalence relation on the nodes, and the partition corresponding to the index consists of the equivalence classes.

Definition 20.8 *Let P be the partition for which $n, m \in I$ for a class I if and only if $L(n) = L(m)$. Then the index $I(G)$ corresponding to P is called a* ***naive index****.*

In case of naive indexes, the same language $L(n)$ is assigned to all elements n of class I in partition P, which will be denoted by $L(I)$.

Proposition 20.9 *Let I be a node of the naive index and R a regular expression. Then $I \cap R(G) = \emptyset$ or $I \subseteq R(G)$.*

Proof Let $n \in I \cap R(G)$ and $m \in I$. Then there exists a label sequence $l_0, \ldots, l_p$ in $L(R)$ to which n fits, that is, $l_0, \ldots, l_p \in L(n)$. Since $L(n) = L(m)$, m also fits to this label sequence, so $m \in I \cap R(G)$. ■

NAIVE-INDEX-EVALUATION(G, R)

```
1 let I_G be the naive index of G
2 Q ← ∅
3 for all I ∈ NAIVE-EVALUATION(I_G, A_R)
4     do Q ← Q ∪ I
5 return Q
```

Proposition 20.10 *Set Q produced by the algorithm* NAIVE-INDEX-EVALUATION *is equal to $R(G)$.*

Proof Because of the previous proposition either all elements of a class I are among the results of a query or none of them. ■

Using naive indexes we can evaluate queries, but, according to the following proposition, not efficiently enough. The proposition was proved by Stockmeyer and Meyer in 1973.

Proposition 20.11 *The creation of the naive index I_G needed in the algorithm* NAIVE-INDEX-EVALUATION *is PSPACE-complete.*

The other problem with using naive indexes is that the sets $L(I)$ are not necessary disjoint for different I, which might cause redundancy in storing.

Because of the above we will try to find a refinement of the partition corresponding to the naive index, which can be created efficiently and can still be used to produce $R(G)$.

Definition 20.12 *Index $I(G)$ is **safe** if for any $n \in V$ and label sequence $l_0, \ldots, l_p$ such that n fits to the label sequence $l_0, \ldots, l_p$ in graph G, class(n) fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$. Index $I(G)$ is **exact** if for any class I of the index and label sequence $l_0, \ldots, l_p$ such that I fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$, arbitrary node $n \in I$ fits to the label sequence $l_0, \ldots, l_p$ in graph G.*

Safety means that the nodes belonging to the result we obtain by evaluation using the index contain the result of the regular query, that is, $R(G) \subseteq R(I(G))$, while exactness means that the evaluation using the index does not provide false results, that is, $R(I(G)) \subseteq R(G)$. Using the definitions of exactness and of the edges of the index the following proposition follows.

Proposition 20.13 *1. Every index is safe.*
2. The naive index is safe and exact.

If I is a set of nodes of G, then the language $L(I)$, to the label strings of which the elements of I fit, was defined using graph G. If we wish to indicate this, we use the notation $L(I, G)$. However, $L(I)$ can also be defined using graph $I(G)$, in which I is a node. In this case, we can use the notation $L(I, I(G))$ instead of $L(I)$, which denotes all label sequences to which node I fits in graph $I(G)$. $L(I, G) = L(I, I(G))$ for safe and exact indexes, so in this case we can write $L(I)$ for simplicity. Then $L(I)$ can be computed using $I(G)$, since the size of $I(G)$ is usually smaller than that of G.

Arbitrary index graph can be queried using the algorithm NAIVE-EVALUATION. After that join the index nodes obtained. If we use an exact index, then the result will be the same as the result we would have obtained by querying the original graph.

INDEX-EVALUATION$(G, I(G), A_R)$

```
1 let I(G) be the index of G
2 Q ← ∅
3 for all I ∈ NAIVE-EVALUATION(I(G), A_R)
4     do Q ← Q ∪ I
5 return Q
```

First, we will define a safe and exact index that can be created efficiently, and is based on the similarity of nodes. We obtain the 1-index this way. Its size can be decreased if we only require similarity locally. The A(k)-index obtained this way lacks exactness, therefore using the algorithm INDEX-EVALUATION we can get results that do not belong to the result of the regular query R, so we have to test our results to ensure exactness.

Definition 20.14 *Let $\approx$ be an equivalence relation on set V such that, for $u \approx v$,*
i) label(u) = label(v),
ii) if there is an edge from node u' to node u, then there exists a node v' for which there is an edge from node v' to node v and $u' \approx v'$.
iii) if there is an edge from node v' to node v, then there exists a node u' for which there is an edge from node u' to node u and $u' \approx v'$.
The above equivalence relation is called a ***bisimulation****. Nodes u and v of a graph are* ***bisimilar*** *if and only if there exists a bisimulation $\approx$ such that $u \approx v$.*

Definition 20.15 *Let P be the partition consisting of the equivalence classes of a bisimulation. The index defined using partition P is called* ***1-index****.*

Proposition 20.16 *The 1-index is a refinement of the naive index. If the labels of the ingoing edges of the nodes in graph G are different, that is, label$(x) \neq$ label(x') for $x \neq x'$ and $(x, y), (x', y) \in E$, then $L(u) = L(v)$ if and only if u and v are bisimilar.*

Proof $label(u) = label(v)$ if $u \approx v$. Let node u fit to the label sequence $l_0, \ldots, l_p$, and let u' be the node corresponding to label l_{p-1}. Then there exists a v' such that $u' \approx v'$ and $(u', u), (v', v) \in E$. u' fits to the label sequence $l_0, \ldots, l_{p-1}$, so, by induction, v' also fits to the label sequence $l_0, \ldots, l_{p-1}$, therefore v fits to the label sequence $l_0, \ldots, l_p$. So, if two nodes are in the same class according to the 1-index, then they are in the same class according to the naive index as well.

To prove the second statement of the proposition, it is enough to show that the naive index corresponds to a bisimulation. Let u and v be in the same class according to the naive index. Then $label(u) = label(v)$. If $(u', u) \in E$, then there exists a label sequence $l_0, \ldots, l_p$ such that the last two nodes corresponding to the labels are u' and u. Since we assumed that the labels of the parents are different, $L(u) = L' \cup L''$, where L' and L'' are disjoint, and $L' = \{l_0, \ldots, l_p \mid u' \text{ fits to the sequence } l_0, \ldots, l_{p-1}, \text{ and } l_p = label(u)\}$, while $L'' = L(u) \setminus L'$. Since $L(u) = L(v)$, there exists a v' such that $(v', v) \in E$ and $label(u') = label(v')$. $L' = \{l_0, \ldots, l_p \mid v'$ fits to the sequence $l_0, \ldots, l_{p-1}$, and $l_p = label(v)\}$ because of the different labels of the parents, so $L(u') = L(v')$, and $u' \approx v'$ by induction, therefore $u \approx v$. ■

Proposition 20.17 *The 1-index is safe and exact.*

Proof If x_p fits to the label sequence $l_0, \ldots, l_p$ in graph G because of nodes $x_0, \ldots, x_p$, then, by the definition of the index graph, there exists an edge from $class(x_i)$ to $class(x_{i+1})$, $0 \leq i \leq p-1$, that is, $class(x_p)$ fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$. To prove exactness, assume that I_p fits to the label sequence $l_0, \ldots, l_p$ in graph $I(G)$ because of $I_0, \ldots, I_p$. Then there are $u' \in I_{p-1}$, $u \in I_p$ such that $u' \approx v'$ and $(v', v) \in E$, that is, $v' \in I_{p-1}$. We can see by induction that v' fits to the label sequence $l_0, \ldots, l_{p-1}$ because of nodes $x_0, \ldots, x_{p-2}, v'$, but then v fits to the label sequence $l_0, \ldots, l_p$ because of nodes $x_0, \ldots, x_{p-2}, v', v$ in graph G. ■

If we consider the bisimulation in case of which all nodes are assigned to different partitions, then the graph $I(G)$ corresponding to this 1-index is the same as graph G. Therefore the size of $I(G)$ is at most the size of G, and we also have to store the elements of I for the nodes I of $I(G)$, which means we have to store all nodes of G. For faster evaluation of queries we need to find the smallest 1-index, that is, the coarsest 1-index. It can be checked that x and y are in the same class according to the coarsest 1-index if and only if x and y are bisimilar.

1-INDEX-EVALUATION(G, R)

1 let I_1 be the coarsest 1-index of G
2 **return** INDEX-EVALUATION(G, I_1, A_R)

In the first step of the algorithm, the coarsest 1-index has to be given. This can be reduced to finding the coarsest stable partition, what we will discuss in the next section of this chapter. Thus using the efficient version of the PT-algorithm, the coarsest 1-index can be found with computation cost $O(m \lg n)$ and space requirement $O(m+n)$, where n and m denote the number of nodes and edges of graph G, respectively.

Since graph I_1 is safe and exact, it is sufficient to evaluate the query in graph I_1, that is, to find the index nodes that fit to the regular expression R. Using Proposition 20.6, the cost of this is a polynomial of the size of graph I_1.

The size of I_1 can be estimated using the following parameters. Let p be the number of different labels in graph G, and k ***the diameter of graph*** G, that is, the length of the longest directed path. (No node can appear twice in the directed path.) If the graph is a tree, then the diameter is the depth of the tree. We often create websites that form a tree of depth d, then we add a navigation bar consisting of q elements to each page, that is, we connect each node of the graph to q chosen pages. It can be proved that in this case the diameter of the graph is at most $d + q(d-1)$. In practice, d and q are usually very small compared to the size of the graph. The proof of the following proposition can be found in the paper of Milo and Suciu.

Proposition 20.18 *Let the number of different labels in graph G be at most p, and let the diameter of G be less than k. Then the size of the 1-index I_1 defined by an arbitrary bisimulation can be bounded from above with a bound that only depends on k and p but does not depend on the size of G.*

Exercises

20.3-1 Show that the index corresponding to the maximal simulation is between the 1-index and the naive index with respect to refinement. Give an example that shows that both inclusions are proper.

20.3-2 Denote by $I_s(G)$ the index corresponding to the maximal simulation. Does $I_s(I_s(G)) = I_s(G)$ hold?

20.3-3 Represent graph G and the state transition graph of the automaton corresponding to the regular expression R with relational databases. Give an algorithm in a relational query language, for example in PL/SQL, that computes $R(G)$.

20.4. Stable partitions and the PT-algorithm

Most index structures used for efficient evaluation of queries of semi-structured databases are based on a partition of the nodes of a graph. The problem of creating indexes can often be reduced to finding the coarsest stable partition.

Definition 20.19 *Let E be a binary relation on the finite set V, that is, $E \subseteq V \times V$. Then V is the set of* ***nodes****, and E is the set of* ***edges****. For arbitrary $S \subseteq V$, let $E(S) = \{y \mid \exists x \in S, (x,y) \in E\}$ and $E^{-1}(S) = \{x \mid \exists\ y \in S, (x,y) \in E\}$. We say that B* ***is stable with respect to*** *S for arbitrary $S \subseteq V$ and $B \subseteq V$, if $B \subseteq E^{-1}(S)$ or $B \cap E^{-1}(S) = \emptyset$. Let P be a partition of V, that is, a decomposition of V into disjoint sets, or in other words, blocks. Then P* ***is stable with respect to*** *S, if all blocks of P are stable with respect to S. P* ***is stable with respect to partition*** *P', if all blocks of P are stable with respect to all blocks of P'. If P is stable with respect to all of its blocks, then partition P is* ***stable****. Let P and Q be two partitions of V. Q* ***is a refinement of*** *P, or P* ***is coarser than*** *Q, if every block of P is the union of some blocks of Q. Given V, E and P,* ***the coarsest stable partition*** *is the coarsest stable refinement of P, that is, the stable refinement of P that is coarser than any other stable refinement of P.*

Note that stability is sometimes defined the following way. B is stable with respect to S if $B \subseteq E(S)$ or $B \cap E(S) = \emptyset$. This is not a major difference, only the direction of the edges is reversed. So in this case stability is defined with respect to the binary relation E^{-1} instead of E, where $(x, y) \in E^{-1}$ if and only if $(y, x) \in E$, since $(E^{-1})^{-1}(S) = \{x \mid \exists\, y \in S, (x, y) \in E^{-1})\} = \{x \mid \exists\, y \in S, (y, x) \in E\} = E(S)$.

Let $|V| = n$ and $|E| = m$. We will prove that there always exists a unique solution of the problem of finding the coarsest stable partition, and there is an algorithm that finds the solution in $O(m \lg n)$ time with space requirement $O(m+n)$. This algorithm was published by R. Paige and R. E. Tarjan in 1987, therefore it will be called the ***PT-algorithm***.

The main idea of the algorithm is that if a block is not stable, then it can be split into two in such a way that the two parts obtained are stable. First we will show a naive method. Then, using the properties of the split operation, we will increase its efficiency by continuing the procedure with the smallest part.

Definition 20.20 *Let E be a binary relation on V, $S \subseteq V$ and Q a partition of V. Furthermore, let **split**(S, Q) be the refinement of Q which is obtained by splitting all blocks B of Q that are not disjoint from $E^{-1}(S)$, that is, $B \cap E^{-1}(S) \neq \emptyset$ and $B \setminus E^{-1}(S) \neq \emptyset$. In this case, add blocks $B \cap E^{-1}(S)$ and $B \setminus E^{-1}(S)$ to the partition instead of B. S is a **splitter** of Q if split$(S, Q) \neq Q$.*

Note that Q is not stable with respect to S if and only if S is a splitter of Q.

Stability and splitting have the following properties, the proofs are left to the Reader.

Proposition 20.21 *Let S and T be two subsets of V, while P and Q two partitions of V. Then*
1. Stability is preserved under refinement, that is, if Q is a refinement of P, and P is stable with respect to S, then Q is also stable with respect to S.
2. Stability is preserved under unification, that is, if P is stable with respect to both S and T, then P is stable with respect to $S \cup T$.
3. The split operation is monotonic in its second argument, that is, if P is a refinement of Q, then split(S, P) is a refinement of split(S, Q).
4. The split operation is commutative in the following sense. For arbitrary S, T and P, split$(S,$ split$(T, P)) =$ split$(T,$ split$(S, P))$, and the coarsest partition of P that is stable with respect to both S and T is split$(S,$ split$(T, P))$.

In the naive algorithm, we refine partition Q starting from partition P, until Q is stable with respect to all of its blocks. In the refining step, we seek a splitter S of Q that is a union of some blocks of Q. Note that finding a splitter among the blocks of Q would be sufficient, but this more general way will help us in improving the algorithm.

NAIVE-PT(V, E, P)

```
1 Q ← P
2 while Q is not stable
3       do let S be a splitter of Q that is the union of some blocks of Q
4          Q ← split(S, Q)
5 return Q
```

Note that the same set S cannot be used twice during the execution of the algorithm, since stability is preserved under refinement, and the refined partition obtained in step 4 is stable with respect to S. The union of the sets S used can neither be used later, since stability is also preserved under unification. It is also obvious that a stable partition is stable with respect to any S that is a union of some blocks of the partition. The following propositions can be proved easily using these properties.

Proposition 20.22 *In any step of the algorithm* NAIVE-PT*, the coarsest stable refinement of P is a refinement of the actual partition stored in Q.*

Proof The proof is by induction on the number of times the cycle is executed. The case $Q = P$ is trivial. Suppose that the statement holds for Q before using the splitter S. Let R be the coarsest stable refinement of P. Since S consists of blocks of Q, and, by induction, R is a refinement of Q, therefore S is the union of some blocks of R. R is stable with respect to all of its blocks and the union of any of its blocks, thus R is stable with respect to S, that is, $R = \mathit{split}(S, R)$. On the other hand, using that the split operation is monotonic, $\mathit{split}(S, R)$ is a refinement of $\mathit{split}(S, Q)$, which is the actual value of Q. ■

Proposition 20.23 *The algorithm* NAIVE-PT *determines the unique coarsest stable refinement of P, while executing the cycle at most $n - 1$ times.*

Proof The number of blocks of Q is obviously at least 1 and at most n. Using the split operation, at least one block of Q is divided into two, so the number of blocks increases. This implies that the cycle is executed at most $n - 1$ times. Q is a stable refinement of P when the algorithm terminates, and, using the previous proposition, the coarsest stable refinement of P is a refinement of Q. This can only happen if Q is the coarsest stable refinement of P. ■

Proposition 20.24 *If we store the set $E^{-1}(\{x\})$ for all elements x of V, then the cost of the algorithm* NAIVE-PT *is at most $O(mn)$.*

Proof We can assume, without restricting the validity of the proof, that there are no sinks in the graph, that is, every node has outgoing edges. Then $1 \leq |E(\{x\})|$ for arbitrary x in V. Consider a partition P, and split all blocks B of P. Let B' be the set of the nodes of B that have at least one outgoing edge. Then $B' = B \cap E^{-1}(V)$. Now let $B'' = B \setminus E^{-1}(V)$, that is, the set of sinks of B. Set B'' is stable with respect to arbitrary S, since $B'' \cap E^{-1}(S) = \emptyset$, so B'' does not have to be split during the algorithm. Therefore, it is enough to examine partition P' consisting of blocks B'

instead of P, that is, a partition of set $V' = E^{-1}(V)$. By adding blocks B'' to the coarsest stable refinement of P' we obviously get the coarsest stable refinement of P. This means that there is a preparation phase before the algorithm in which P' is obtained, and a processing phase after the algorithm in which blocks B'' are added to the coarsest stable refinement obtained by the algorithm. The cost of preparation and processing can be estimated the following way. V' has at most m elements. If, for all x in V we have $E^{-1}(\{x\})$, then the preparation and processing requires $O(m+n)$ time.

From now on we will assume that $1 \leq |E(\{x\})|$ holds for arbitrary x in V, which implies that $n \leq m$. Since we store sets $E^{-1}(\{x\})$, we can find a splitter among the blocks of partition Q in $O(m)$ time. This, combined with the previous proposition, means that the algorithm can be performed in $O(mn)$ time. ■

The algorithm can be executed more efficiently using a better way of finding splitter sets. The main idea of the improved algorithm is that we work with two partitions besides P, Q and a partition X that is a refinement of Q in every step such that Q is stable with respect to all blocks of X. At the start, let $Q = P$ and let X be the partition consisting only one block, set V. The refining step of the algorithm is repeated until $Q = X$.

PT(V, E, P)

```
1  Q ← P
2  X ← {V}
3  while X ≠ Q
4        do let S be a block of X that is not a block of Q,
             and B a block of Q in S for which |B| ≤ |S|/2
5           X ← (X \ {S}) ∪ {B, S \ B}
6           Q ← split(S \ B, split(B, Q))
7  return Q
```

Proposition 20.25 *The result of the* PT*-algorithm is the same as that of algorithm* Naive-PT.

Proof At the start, Q is a stable refinement of P with respect to the blocks of X. In step 5, a block of X is split, thus we obtain a refinement of X. In step 6, by refining Q using splits we ensure that Q is stable with respect to two new blocks of X. The properties of stability mentioned in Proposition 20.21 and the correctness of algorithm Naive-PT imply that the PT-algorithm also determines the unique coarsest stable refinement of P. ■

In some cases one of the two splits of step 6 can be omitted. A sufficient condition is that E is a function of x.

Proposition 20.26 *If $|E(\{x\})| = 1$ for all x in V, then step 6 of the* PT*-algorithm can be exchanged with $Q \leftarrow split(B, Q)$.*

Proof Suppose that Q is stable with respect to a set S which is the union of some blocks of Q. Let B be a block of Q that is a subset of S. It is enough to prove that

split(B, Q) is stable with respect to $(S \setminus B)$. Let B_1 be a block of *split*(B, Q). Since the result of a split according to B is a stable partition with respect to B, either $B_1 \subseteq E^{-1}(B)$ or $B_1 \subseteq E^{-1}(S) \setminus E^{-1}(B)$. Using $|E(\{x\})| = 1$, we get $B_1 \cap E^{-1}(S \setminus B) = \emptyset$ in the first case, and $B_1 \subseteq E^{-1}(S \setminus B)$ in the second case, which means that we obtained a stable partition with respect to $(S \setminus B)$. ■

Note that the stability of a partition with respect to S and B generally does not imply that it is also stable with respect to $(S \setminus B)$. If this is true, then the execution cost of the algorithm can be reduced, since the only splits needed are the ones according to B because of the reduced sizes.

The two splits of step 6 can cut a block into four parts in the general case. According to the following proposition, one of the two parts gained by the first split of a block remains unchanged at the second split, so the two splits can result in at most three parts. Using this, the efficiency of the algorithm can be improved even in the general case.

Proposition 20.27 *Let Q be a stable partition with respect to S, where S is the union of some blocks of Q, and let B be a block of Q that is a subset of S. Furthermore, let D be a block of Q that is cut into two (proper) parts D_1 and D_2 by the operation split(B, Q) in such a way that none of these is the empty set. Suppose that block D_1 is further divided into the nonempty sets D_{11} and D_{12} by split$(S \setminus B,$ split$(B, Q))$. Then*
1. $D_1 = D \cap E^{-1}(B)$ and $D_2 = D \setminus D_1$ if and only if $D \cap E^{-1}(B) \neq \emptyset$ and $D \setminus E^{-1}(B) \neq \emptyset$.
2. $D_{11} = D_1 \cap E^{-1}(S \setminus B)$ and $D_{12} = D_1 \setminus D_{11}$ if and only if $D_1 \cap E^{-1}(S \setminus B) \neq \emptyset$ and $D_1 \setminus E^{-1}(S \setminus B) \neq \emptyset$.
3. The operation split$(S \setminus B,$ split$(B, Q))$ leaves block D_2 unchanged.
4. $D_{12} = D_1 \cap (E^{-1}(B) \setminus E^{-1}(S \setminus B))$.

Proof The first two statements follow using the definition of the split operation. To prove the third statement, suppose that D_2 was obtained from D by a proper decomposition. Then $D \cap E^{-1}(B) \neq \emptyset$, and since $B \subseteq S$, $D \cap E^{-1}(S) \neq \emptyset$. All blocks of partition Q, including D, are stable with respect to S, which implies $D \subseteq E^{-1}(S)$. Since $D_2 \subseteq D$, $D_2 \subseteq E^{-1}(S) \setminus E^{-1}(B) = E^{-1}(S \setminus B)$ using the first statement, so D_2 is stable with respect to the set $S \setminus B$, therefore a split according to $S \setminus B$ does not divide block D_2. Finally, the fourth statement follows from $D_1 \subseteq E^{-1}(B)$ and $D_{12} = D_1 \setminus E^{-1}(S \setminus B)$. ■

Denote by *counter*(x, S) the number of nodes in S that can be reached from x, that is, *counter*$(x, S) = |S \cap E(\{x\})|$. Note that if $B \subseteq S$, then $E^{-1}(B) \setminus E^{-1}(S \setminus B) = \{x \in E^{-1}(B) \mid$ *counter*$(x, B) =$ *counter*$(x, S)\}$.

Since sizes are always halved, an arbitrary x in V can appear in at most $\lg n + 1$ different sets B that were used for refinement in the PT-algorithm. In the following, we will give an execution of the PT algorithm in which the determination of the refinement according to block B in steps 5 and 6 of the algorithm costs $O(|B| + \sum_{y \in B} |E^{-1}(\{y\})|)$. Summing this for all blocks B used in the algorithm and for all elements of these blocks, we get that the complexity of the algorithm EFFICIENT-PT is at most $O(m \lg n)$. To give such a realization of the algorithm, we have to choose good data structures for the representation of our data.

- Attach node x to all edges (x, y) of set E, and attach the list $\{(x, y) \mid (x, y) \in E\}$ to all nodes y. Then the cost of reading set $E^{-1}(\{y\})$ is proportional to the size of $E^{-1}(\{y\})$.
- Let partition Q be a refinement of partition X. Represent the blocks of the two partitions by records. A block S of partition X is ***simple*** if it consists of one block of Q, otherwise it is ***compound***.
- Let C be the list of all compound blocks in partition X. At start, let $C = \{V\}$, since V is the union of the blocks of P. If P consists of only one block, then P is its own coarsest stable refinement, so no further computation is needed.
- For any block S of partition P, let $Q\text{-}blocks(S)$ be the double-chained list of the blocks of partition Q the union of which is set S. Furthermore, store the values $counter(x, S)$ for all x in set $E^{-1}(S)$ to which one pointer points from all edges (x, y) such that y is an element of S. At start, the value assigned to all nodes x is $counter(x, V) = |E(\{x\})|$, and make a pointer to all nodes (x, y) that points to the value $counter(x, V)$.
- For any block B of partition Q, let $X\text{-}block(B)$ be the block of partition X in which B appears. Furthermore, let $size(B)$ be the cardinality of B, and $elements(B)$ the double-chained list of the elements of B. Attach a pointer to all elements that points to the block of Q in which this element appears. Using double chaining any element can be deleted in $O(1)$ time.

Using the proof of Proposition 20.24, we can suppose that $n \leq m$ without restricting the validity. It can be proved that in this case the space requirement for the construction of such data structures is $O(m)$.

EFFICIENT-PT(V, E, P)

```
1  if |P| = 1
2     then return P
3  Q ← P
4  X ← {V}
5  C ← {V}                          ▷ C is the list of the compound blocks of X.
6  while C ≠ ∅
7        do let S be an element of C
8           let B be the smaller of the first two elements of S
9           C ← C \ {S}
10          X ← (X \ {S}) ∪ {{B}, S \ {B}}
11          S ← S \ {B}
12          if |S| > 1
13             then C ← C ∪ {S}
```

14 Generate set $E^{-1}(B)$ by reading the edges (x, y) of set E for which y is an element of B, and for all elements x of this set, compute the value $\mathit{counter}(x, B)$.
15 Find blocks $D_1 = D \cap E^{-1}(B)$ and $D_2 = D \setminus D_1$ for all blocks D of Q by reading set $E^{-1}(B)$
16 By reading all edges (x, y) of set E for which y is an element of B, create set $E^{-1}(B) \setminus E^{-1}(S \setminus B)$ checking the condition $\mathit{counter}(x, B) = \mathit{counter}(x, S)$
17 Reading set $E^{-1}(B) \setminus E^{-1}(S \setminus B)$, for all blocks D of Q, determine the sets $D_{12} = D_1 \cap (E^{-1}(B) \setminus E^{-1}(S \setminus B))$ and $D_{11} = D_1 \setminus D_{12}$
18 **for** all blocks D of Q for which $D_{11} \neq \emptyset$, $D_{12} \neq \emptyset$ and $D_2 \neq \emptyset$
19 **do if** D is a simple block of X
20 **then** $C \leftarrow C \cup \{D\}$
21 $Q \leftarrow (Q \setminus \{D\}) \cup \{D_{11}, D_{12}, D_2\}$
22 Compute the value $\mathit{counter}(x, S)$ by reading the edges (x, y) of E for which y is an element of B.
23 **return** Q

Proposition 20.28 *The algorithm* Efficient-PT *determines the coarsest stable refinement of P. The computation cost of the algorithm is $O(m \lg n)$, and its space requirement is $O(m + n)$.*

Proof The correctness of algorithm follows from the correctness of the PT-algorithm and Proposition 20.27. Because of the data structures used, the computation cost of the steps of the cycle is proportional to the number of edges examined and the number of elements of block B, which is $O(|B| + \sum_{y \in B} |E^{-1}(\{y\})|)$ altogether. Sum this for all blocks B used during the refinement and all elements of these blocks. Since the size of B is at most half the size of S, arbitrary x in set V can be in at most $\lg n + 1$ different sets B. Therefore, the total computation cost of the algorithm is $O(m \lg n)$. It can be proved easily that a space of $O(m + n)$ size is enough for the storage of the data structures used in the algorithm and their maintenance. ■

Note that the algorithm could be further improved by contracting some of its steps but that would only decrease computation cost by a constant factor.

Let $G^{-1} = (V, E^{-1})$ be the graph that can be obtained from G by changing the direction of all edges of G. Consider a 1-index in graph G determined by the bisimulation $\approx$. Let I and J be two classes of the bisimulation, that is, two nodes of $I(G)$. Using the definition of bisimulation, $J \subseteq E(I)$ or $E(I) \cap J = \emptyset$. Since $E(I) = (E^{-1})^{-1}(I)$, this means that J is stable with respect to I in graph G^{-1}. So the coarsest 1-index of G is the coarsest stable refinement of the basic partition of graph G^{-1}.

Corollary 20.29 *The coarsest 1-index can be determined using the algorithm* Efficient-PT. *The computation cost of the algorithm is at most $O(m \lg n)$, and its space requirement is at most $O(m + n)$.*

Exercises

20.4-1 Prove Proposition 29.21.

20.4-2 Partition P is size-stable with respect to set S if $|E(\{x\}) \cap S| = |E(\{y\}) \cap S|$ for arbitrary elements x, y of a block B of P. A partition is size-stable if it is size-stable with respect to all its blocks. Prove that the coarsest size-stable refinement of an arbitrary partition can be computed in $O(m \lg n)$ time.

20.4-3 The 1-index is minimal if no two nodes I and J with the same label can be contracted, since there exists a node K for which $I \cup J$ is not stable with respect to K. Give an example that shows that the minimal 1-index is not unique, therefore it is not the same as the coarsest 1-index.

20.4-4 Prove that in case of an acyclic graph, the minimal 1-index is unique and it is the same as the coarsest 1-index.

20.5. A(k)-indexes

In case of 1-indexes, nodes of the same class fit to the same label sequences starting from the root. This means that the nodes of a class cannot be distinguished by their ancestors. Modifying this condition in such a way that indistinguishability is required only locally, that is, nodes of the same class cannot be distinguished by at most k generations of ancestors, we obtain an index that is coarser and consists of less classes than the 1-index. So the size of the index decreases, which also decreases the cost of the evaluation of queries. The 1-index was safe and exact, which we would like to preserve, since these guarantee that the result we get when evaluating the queries according to the index is the result we would have obtained by evaluating the query according to the original graph. The A(k)-index is also safe, but it is not exact, so this has to be ensured by modification of the evaluation algorithm.

Definition 20.30 *The* ***k-bisimulation*** $\approx^k$ *is anequivalence relation on the nodes V of a graph defined recursively as*
i) $u \approx^0 v$ *if and only if label(u) = label(v),*
ii) $u \approx^k v$ *if and only if* $u \approx^{k-1} v$ *and if there is an edge from node u' to node u, then there is a node v' from which there is an edge to node v and* $u' \approx^{k-1} v'$*, also, if there is an edge from node v' to node v, then there is a node u' from which there is an edge to node u and* $u' \approx^{k-1} v'$*.*
In case $u \approx^k v$ *u and v are* ***k-bisimilar****. The classes of the partition according to the* ***A(k)-index*** *are the equivalence classes of the k-bisimulation.*

The "A" in the notation refers to the word "approximative".

Note that the partition belonging to $k = 0$ is the basic partition, and by increasing k we refine this, until the coarsest 1-index is reached.

Denote by $L(u, k, G)$ the label sequences of length at most k to which u fits in graph G. The following properties of the A(k)-index can be easily checked.

Proposition 20.31
1. If u and v are k-bisimilar, then $L(u,k,G) = L(v,k,G)$.
2. If I is a node of the $A(k)$-index and $u \in I$, then $L(I,k,I(G)) = L(u,k,G)$.
3. The $A(k)$-index is exact in case of simple expressions of length at most k.
4. The $A(k)$-index is safe.
5. The $(k+1)$-bisimulation is a (not necessarily proper) refinement of the k-bisimulation.

The A(k)-index compares the k-distance half-neighbourhoods of the nodes which contain the root, so the equivalence of the nodes is not affected by modifications outside this neighbourhood, as the following proposition shows.

Proposition 20.32 *Suppose that the shortest paths from node v to nodes x and y contain more than k edges. Then adding or deleting an edge from u to v does not change the k-bisimilarity of x and y.*

We use a modified version of the PT-algorithm for creating the A(k)-index. Generally, we can examine the problem of approximation of the coarsest stable refinement.

Definition 20.33 *Let P be a partition of V in the directed graph $G = (V,E)$, and let $P_0, P_1, \ldots, P_k$ be a sequence of partitions such that $P_0 = P$ and P_{i+1} is the coarsest refinement of P_i that is stable with respect to P_i. In this case, partition P_k is the **k-step approximation of the coarsest stable refinement of** P.*

Note that every term of sequence P_i is a refinement of P, and if $P_k = P_{k-1}$, then P_k is the coarsest stable refinement of P. It can be checked easily that an arbitrary approximation of the coarsest stable refinement of P can be computed greedily, similarly to the PT-algorithm. That is, if a block B of P_i is not stable with respect to a block S of P_{i-1}, then split B according to S, and consider the partition $split(S, P_i)$ instead of P_i.

NAIVE-APPROXIMATION(V, E, P, k)

```
1  P_0 ← P
2  for i ← 1 to k
3       do P_i ← P_{i-1}
4          for all S ∈ P_{i-1} such that split(S, P_i) ≠ P_i
5               do P_i ← split(S, P_i)
6  return P_k
```

Note that the algorithm NAIVE-APPROXIMATION could also be improved similarly to the PT-algorithm.

Algorithm NAIVE-APPROXIMATION can be used to compute the A(k)-index, all we have to notice is that the partition belonging to the A(k)-index is stable with respect to the partition belonging to the A($k-1$)-index in graph G^{-1}. It can be shown that the computation cost of the A(k)-index obtained this way is $O(km)$, where m is the number of edges in graph G.

A(k)-INDEX-EVALUATION(G, A_R, k)

```
1 let I_k be the A(k)-index of G
2 Q ← INDEX-EVALUATION(G, I_k, A_R)
3 for all u ∈ Q
4     do if L(u) ∩ L(A_R) = ∅
5           then Q ← Q \ {u}
6 return Q
```

The A(k)-index is safe, but it is only exact for simple expressions of length at most k, so in step 4, we have to check for all elements u of set Q whether it satisfies query R, and we have to delete those from the result that do not fit to query R. We can determine using a finite nondeterministic automaton whether a given node satisfies expression R as in Proposition 20.6, but the automaton has to run in the other way. The number of these checks can be reduced according to the following proposition, the proof of which is left to the Reader.

Proposition 20.34 *Suppose that in the graph I_k belonging to the $A(k)$-index, index node I fits to a label sequence that ends with $s = l_0, \ldots, l_p$, $p \leq k-1$. If all label sequences of the form s'.s that start from the root satisfy expression R in graph G, then all elements of I satisfy expression R.*

Exercises

20.5-1 Denote by $A_k(G)$ the A(k)-index of G. Determine whether $A_k(A_l(G)) = A_{k+l}(G)$.

20.5-2 Prove Proposition 20.31.

20.5-3 Prove Proposition 20.32.

20.5-4 Prove Proposition 20.34.

20.5-5 Prove that the algorithm NAIVE-APPROXIMATION generates the coarsest k-step stable approximation.

20.5-6 Let $A = \{A_0, A_1, \ldots, A_k\}$ be a set of indexes, the elements of which are A(0)-, A(1)-, ..., A(k)-indexes, respectively. A is *minimal*, if by uniting any two elements of A_i, A_i is not stable with respect to A_{i-1}, $1 \leq i \leq k$. Prove that for arbitrary graph, there exists a unique minimal A the elements of which are coarsest A(i)-indexes, $0 \leq i \leq k$.

20.6. D(k)- and M(k)-indexes

When using A(k)-indexes, the value of k must be chosen appropriately. If k is too large, the size of the index will be too big, and if k is too small, the result obtained has to be checked too many times in order to preserve exactness. Nodes of the same class are similar locally, that is, they cannot be distinguished by their k distance neighbourhoods, or, more precisely, by the paths of length at most k leading to them. The same k is used for all nodes, even though there are less important nodes. For instance, some nodes appear very rarely in results of queries in practice, and only the label sequences of the paths passing through them are examined. There is no

reason for using a better refinement on the less important nodes. This suggests the idea of using the dynamic D(k)-index, which assigns different values k to the nodes according to queries. Suppose that a set of queries is given. If there is an $R.a.b$ and an $R.a.b.c$ query among them, where R and R' are regular queries, then a partition according to at least 1-bisimulation in case of nodes with label b, and according to at least 2-bisimulation in case of nodes with label c is needed.

Definition 20.35 *Let $I(G)$ be the index graph belonging to graph G, and to all index node I assign a nonnegative integer $k(I)$. Suppose that the nodes of block I are $k(I)$-bisimilar. Let the values $k(I)$ satisfy the following condition: if there is an edge from I to J in graph $I(G)$, then $k(I) \geq k(J) - 1$. The index $I(G)$ having this property is called a* ***D*(k)*-index**.

The "D" in the notation refers to the word "dynamic". Note that the A(k)-index is a special case of the D(k)-index, since in case of A(k)-indexes, the elements belonging to any index node are exactly k-bisimilar.

Since classification according to labels, that is, the basic partition is an A(0)-index, and in case of finite graphs, the 1-index is the same as an A(k)-index for some k, these are also special cases of the D(k)-index. The D(k)-index, just like any other index, is safe, so it is sufficient to evaluate the queries on them. Results must be checked to ensure exactness. The following proposition states that exactness is guaranteed for some queries, therefore checking can be omitted in case of such queries.

Proposition 20.36 *Let I_1, I_2, ..., I_s be a directed path in the D(k)-index, and suppose that $k(I_j) \geq j - 1$ if $1 \leq j \leq s$. Then all elements of I_s fit to the label sequence label(I_1), label(I_2), ..., label(I_s).*

Proof The proof is by induction on s. The case $s = 1$ is trivial. By the inductive assumption, all elements of I_{s-1} fit to the label sequence label(I_1), label(I_2), ..., label(I_{s-1}). Since there is an edge from node I_{s-1} to node I_s in graph $I(G)$, there exist $u \in I_s$ and $v \in I_{s-1}$ such that there is an edge from v to u in graph G. This means that u fits to the label sequence label(I_1), label(I_2), ..., label(I_s) of length $s - 1$. The elements of I_s are at least $(s-1)$-bisimilar, therefore all elements of I_s fit to this label sequence. ■

Corollary 20.37 *The D(k)-index is exact with respect to label sequence l_0, ..., l_m if $k(I) \geq m$ for all nodes I of the index graph that fit to this label sequence.*

When creating the D(k)-index, we will refine the basic partition, that is, the A(0)-index. We will assign initial values to the classes consisting of nodes with the same label. Suppose we use t different values. Let K_0 be the set of these values, and denote the elements of K_0 by $k_1 > k_2 > \ldots > k_t$. If the elements of K_0 do not satisfy the condition given in the D(k)-index, then we increase them using the algorithm WEIGHT-CHANGER, starting with the greatest value, in such a way that they satisfy the condition. Thus, the classes consisting of nodes with the same label will have good k values. After this, we refine the classes by splitting them, until all elements of a class are k-bisimilar, and assign this k to all terms of the split. During this process

the edges of the index graph must be refreshed according to the partition obtained by refinement.

WEIGHT-CHANGER(G, K_0)

```
1  K ← ∅
2  K_1 ← K_0
3  while K_1 ≠ ∅
4        do for all I, where I is a node of the A(0)-index and k(I) = max(K_1)
5              do for all J, where J is a node of the A(0)-index
                     and there is an edge from J to I
6                    k(J) ← max(k(J), max(K_1) − 1)
7        K ← K ∪ {max(K_1)}
8        K_1 ← {k(A) | A is a node of the A(0)-index } \ K
9  return K
```

It can be checked easily that the computation cost of the algorithm WEIGHT-CHANGER is $O(m)$, where m is the number of edges of the A(0)-index.

D(k)-INDEX-CREATOR(G, K_0)

```
1  let I(G) be the A(0)-index belonging to graph G, let V_I be the set
   of nodes of I(G), let E_I be the set of edges of I(G)
2  K ← WEIGHT-CHANGER(G, K_0)              ▷ Changing the initial weights
                                             according to the condition of the D(k)-index.
3  for k ← 1 to max(K)
4     do for all I ∈ V_I
5           do if k(I) ≥ k
6                 then for all J, where (J, I) ∈ E_I
7                         do V_I ← (V_I \ {I}) ∪ {I ∩ E(J), I \ E(J)}
8                            k(I ∩ E(J)) ← k(I)
9                            k(I \ E(J)) ← k(I)
10                           E_I ← {(A, B) | A, B ∈ V_I, ∃ a ∈ A,
                                    ∃ b ∈ B, (a, b) ∈ E}
11                           I(G) ← (V_I, E_I)
12 return I(G)
```

In step 7, a split operation is performed. This ensures that the classes consisting of $(k-1)$-bisimilar elements are split into equivalence classes according to k-bisimilarity. It can be proved that the computation cost of the algorithm D(k)-INDEX-CREATOR is at most $O(km)$, where m is the number of edges of graph G, and $k = max(K_0)$.

In some cases, the D(k)-index results in a partition that is too fine, and it is not efficient enough for use because of its huge size. Over-refinement can originate in the following. The algorithm D(k)-INDEX-CREATOR assigns the same value k to the nodes with the same label, although some of these nodes might be less important with respect to queries, or appear more often in results of queries of length much less than k, so less fineness would be enough for these nodes. Based on the value

k assigned to a node, the algorithm WEIGHT-CHANGER will not decrease the value assigned to the parent node if it is greater than $k-1$. Thus, if these parents are not very significant nodes considering frequent queries, then this can cause over-refinement. In order to avoid over-refinement, we introduce the M(k)-index and the M*(k)-index, where the "M" refers to the word "mixed", and the "*" shows that not one index is given but a finite hierarchy of gradually refined indexes. The M(k)-index is a D(k)-index the creation algorithm of which not necessarily assigns nodes with the same label to the same k-bisimilarity classes.

Let us first examine how a D(k)-index $I(G) = (V_I, E_I)$ must be modified if the initial weight k_I of index node I is increased. If $k(I) \geq k_I$, then $I(G)$ does not change. Otherwise, to ensure that the conditions of the D(k)-index on weights are satisfied, the weights on the ancestors of I must be increased recursively until the weight assigned to the parents is at least $k_I - 1$. Then, by splitting according to the parents, the fineness of the index nodes obtained will be at least k_I, that is, the elements belonging to them will be at least k_I-bisimilar. This will be achieved using the algorithm WEIGHT-INCREASER.

WEIGHT-INCREASER$(I, k_I, I(G))$

```
1  if k(I) ≥ k_I
2     then return I(G)
3  for all (J, I) ∈ E_I
4        do I(G) ← WEIGHT-INCREASER(J, k_I − 1, I(G))
5  for all (J, I) ∈ E_I
6        do V_I ← (V_I \ {I}) ∪ {I ∩ E(J), I \ E(J)}
7           E_I ← {(A, B) | A, B ∈ V_I, ∃ a ∈ A, ∃ b ∈ B, (a, b) ∈ E}
8           I(G) ← (V_I, E_I)
9  return I(G)
```

The following proposition can be easily proved, and with the help of this we will be able to achieve the appropriate fineness in one step, so we will not have to increase step by step anymore.

Proposition 20.38 *$u \approx^k v$ if and only if $u \approx^0 v$, and if there is an edge from node u' to node u, then there is a node v', from which there is an edge to node v and $u' \approx^{k-1} v'$, and, conversely, if there is an edge from node v' to node v, then there is a node u', from which there is an edge to node u and $u' \approx^{k-1} v'$.*

Denote by *FRE* the set of simple expressions, that is, the label sequences determined by the frequent regular queries. We want to achieve a fineness of the index that ensures that it is exact on the queries belonging to *FRE*. For this, we have to determine the significant nodes, and modify the algorithm D(k)-INDEX-CREATOR in such a way that the not significant nodes and their ancestors are always deleted at the refining split.

Let $R \in FRE$ be a frequent simple query. Denote by S and T the set of nodes that fit to R in the index graph and data graph, respectively, that is $S = R(I(G))$ and $T = R(G)$. Denote by $k(I)$ the fineness of index node I in the index graph $I(G)$, then the nodes belonging to I are at most $k(I)$-bisimilar.

REFINE(R, S, T)

```
1  for all I ∈ S
2      do I(G) ← REFINE-INDEX-NODE(I, length(R), I ∩ T)
3  while ∃ I ∈ V_I such that k(I) < length(R) and I fits to R
4        do I(G) ← WEIGHT-INCREASER(I, length(R), I(G))
5  return I(G)
```

The refinement of the index nodes will be done using the following algorithm. First, we refine the significant parents of index node I recursively. Then we split I according to its significant parents in such a way that the fineness of the new parts is k. The split parts of I are kept in set H. Lastly, we unite those that do not contain significant nodes, and keep the original fineness of I for this united set.

REFINE-INDEX-NODE$(I,\ k,\ \textit{significant-nodes})$

```
1  if k(I) ≥ k
2    then return I(G)
3  for all (J, I) ∈ E_I
4      do significant-parents ← E^{-1}(significant-nodes) ∩ J
5         if significant-parents ≠ ∅
6           then REFINE-INDEX-NODE(J, k − 1, significant-parents)
7  k-previous ← k(I)
8  H ← {I}
9  for all (J, I) ∈ E_I
10     do if E^{-1}(significant-parents) ∩ J ≠ ∅
11           then for all F ∈ H
12                    do V_I ← (V_I \ {F}) ∪ {F ∩ E(J), F \ E(J)}
13                       E_I ← {(A, B) | A, B ∈ V_I, ∃ a ∈ A, ∃ b ∈ B, (a, b) ∈ E}
14                       k(F ∩ E(J)) ← k
15                       k(F \ E(J)) ← k
16                       I(G) ← (V_I, E_I)
17                       H ← (H \ {F}) ∪ {F ∩ E(J), F \ E(J)}
18 remaining ← ∅
19 for all F ∈ H
20     do if significant-nodes ∩ F = ∅
21           then remaining ← remaining ∪ F
22                V_I ← (V_I \ {F})
23 V_I ← V_I ∪ {remaining}
24 E_I ← {(A, B) | A, B ∈ V_I, ∃ a ∈ A, ∃ b ∈ B, (a, b) ∈ E}
25 k(remaining) ← k-previous
26 I(G) ← (V_I, E_I)
27 return I(G)
```

The algorithm REFINE refines the index graph $I(G)$ according to a frequent simple expression in such a way that it splits an index node into not necessarily

equally fine parts, and thus avoids over-refinement. If we start from the A(0)-index, and create the refinement for all frequent queries, then we get an index graph that is exact with respect to frequent queries. This is called the M(k)-index. The set *FRE* of frequent queries might change during the process, so the index must be modified dynamically.

Definition 20.39 *The* ***M*(k)*-index** is a $D(k)$-index created using the following* M(k)-INDEX-CREATOR *algorithm.*

M(k)-INDEX-CREATOR(G, *FRE*)

```
1  I(G) ← the A(0) index belonging to graph G
2  V_I ← the nodes of I(G)
3  for all I ∈ V_I
4      do k(I) ← 0
5  E_I ← the set of edges of I(G)
6  for all R ∈ FRE
7      do I(G) ← REFINE(R, R(I(G)), R(G))
8  return I(G)
```

The M(k)-index is exact with respect to frequent queries. In case of a not frequent query, we can do the following. The M(k)-index is also a D(k)-index, therefore if an index node fits to a simple expression R in the index graph $I(G)$, and the fineness of the index node is at least the length of R, then all elements of the index node fit to the query R in graph G. If the fineness of the index node is less, then for all of its elements, we have to check according to NAIVE-EVALUATION whether it is a solution in graph G.

When using the M(k)-index, over-refinement is the least if the lengths of the frequent simple queries are the same. If there are big differences between the lengths of frequent queries, then the index we get might be too fine for the short queries. Create the sequence of gradually finer indexes with which we can get from the A(0)-index to the M(k)-index in such a way that, in every step, the fineness of parts obtained by splitting an index node is greater by at most one than that of the original index node. If the whole sequence of indexes is known, then we do not have to use the finest and therefore largest index for the evaluation of a simple query, but one whose fineness corresponds to the length of the query.

Definition 20.40 *The* ***M***(k)*-index** is a sequence of indexes I_0, I_1, ..., I_k such that*

1. *Index I_i is an M(i)-index, where $i = 0, 1, \ldots, k$.*
2. *The fineness of all index nodes in I_i is at most i, where $i = 0, 1, \ldots, k$.*
3. *I_{i+1} is a refinement of I_i, where $i = 0, 1, \ldots, k-1$.*
4. *If node J of index I_i is split in index I_{i+1}, and J' is a set obtained by this split, that is, $J' \subseteq J$, then $k(J) \leq k(J') \leq k(J) + 1$.*

5. *Let J be a node of index I_i, and $k(J) < i$. Then $k(J) = k(J')$ for $i < i'$ and for all J' index nodes of $I_{i'}$ such that $J' \subseteq J$.*

It follows from the definition that in case of M*(k)-indexes I_0 is the A(0)-index. The last property says that if the refinement of an index node stops, then its fineness will not change anymore. The M*(k)-index possesses the good characteristics of the M(k)-index, and its structure is also similar: according to frequent queries the index is further refined if it is necessary to make it exact on frequent queries, but now we store and refresh the coarser indexes as well, not only the finest.

When representing the M*(k)-index, we can make use of the fact that if an index node is not split anymore, then we do not need to store this node in the new indexes, it is enough to refer to it. Similarly, edges between such nodes do not have to be stored in the sequence of indexes repeatedly, it is enough to refer to them. Creation of the M*(k)-index can be done similarly to the M(k)-INDEX-CREATOR algorithm. The detailed description of the algorithm can be found in the paper of He and Yang.

With the help of the M*(k)-index, we can use several strategies for the evaluation of queries. Let R be a frequent simple query.

The simplest strategy is to use the index the fineness of which is the same as the length of the query.

M*(k)-INDEX-NAIVE-EVALUATION(G, *FRE*, R)

```
1  {I_0, I_1 ,... , I_k} ← the M*(k)-index corresponding to graph G
2  h ← length(R)
3  return INDEX-EVALUATION(G, I_h, A_R)
```

The evaluation can also be done by gradually evaluating the longer prefixes of the query according to the index the fineness of which is the same as the length of the prefix. For the evaluation of a prefix, consider the partitions of the nodes found during the evaluation of the previous prefix in the next index and from these, seek edges labeled with the following symbol. Let $R = l_0, l_1, \ldots, l_h$ be a simple frequent query, that is, $length(R) = h$.

M*(k)-INDEX-EVALUATION-TOP-TO-BOTTOM(G, *FRE*, R)

```
1  {I_0, I_1, ..., I_k} ← the M*(k)-index corresponding to graph G
2  R_0 ← l_0
3  H_0 ← ∅
4  for all C ∈ E_{I_0}(root(I_0))          ▷ The children of the root in graph I_0.
5      do if label(C) = l_0
6           then H_0 ← H_0 ∪ {C}
```

```
7  for j ← 1 to length(R)
8      do H_j ← ∅
9          R_j ← R_{j−1}.l_j
10         H_{j−1} ← M*(k)-Index-Evaluation-Top-to-Bottom(G,FRE, R_{j−1})
11         for all A ∈ H_{j−1}                    ▷ Node A is a node of graph I_{j−1}.
12             do if A = ∪B_m, where B_m ∈ V_{I_j}   ▷ The partition of node A
                                                      in graph I_j.
13                    then for minden B_m
14                             do for all C ∈ E_{I_j}(B_m)   ▷ For all children of
                                                               B_m in graph I_j.
15                                    do if label(C) = l_j
16                                          then H_j ← H_j ∪ {C}
17 return H_h
```

Our strategy could also be that we first find a subsequence of the label sequence corresponding to the simple query that contains few nodes, that is, its selectivity is large. Then find the fitting nodes in the index corresponding to the length of the subsequence, and using the sequence of indexes see how these nodes are split into new nodes in the finer index corresponding to the length of the query. Finally, starting from these nodes, find the nodes that fit to the remaining part of the original query. The detailed form of the algorithm M*(k)-Index-Prefiltered-Evaluation is left to the Reader.

Exercises

20.6-1 Find the detailed form of the algorithm M*(k)-Index-Prefiltered-Evaluation . What is the cost of the algorithm?

20.6-2 Prove Proposition 20.38.

20.6-3 Prove that the computation cost of the algorithm Weight-Changer is $O(m)$, where m is the number of edges of the A(0)-index.

20.7. Branching queries

With the help of regular queries we can select the nodes of a graph that are reached from the root by a path the labels of which fit to a given regular pattern. A natural generalization is to add more conditions that the nodes of the path leading to the node have to satisfy. For example, we might require that the node can be reached by a label sequence from a node with a given label. Or, that a node with a given label can be reached from another node by a path with a given label sequence. We can take more of these conditions, or use their negation or composition. To check whether the required conditions hold, we have to step not only forward according to the direction of the edges, but sometimes also backward. In the following, we will give the description of the language of branching queries, and introduce the forward-backward indexes. The forward-backward index which is safe and exact with respect

to all branching queries is called FB-index. Just like the 1-index, this is also usually too large, therefore we often use an FB(f, b, d)-index instead, which is exact if the length of successive forward steps is at most f, the length of successive backward steps is at most b, and the depth of the composition of conditions is at most d. In practice, values f, b and d are usually small. In case of queries for which the value of one of these parameters is greater than the corresponding value of the index, a checking step must be added, that is, we evaluate the query on the index, and only keep those nodes of the resulted index nodes that satisfy the query.

If there is a directed edge from node n to node m, then this can be denoted by n/m or $m\backslash n$. If node m can be reached from node n by a directed path, then we can denote that by $n//m$ or $m\backslash\backslash n$. (Until now we used . instead of /, so // represents the regular expression _* or * in short.)

From now on, a label sequence is a sequence in which separators are the forward signs (/, //) and the backward signs (\, \\). A sequence of nodes fit to a label sequence if the relation of successive nodes is determined by the corresponding separator, and the labels of the nodes come according to the label sequence.

There are only forward signs in ***forward label sequences***, and only backward signs in ***backward label sequences***.

Branching queries are defined by the following grammar .

```
branching_query          ::= forward_label sequence [ or_expression ]
                               forward_sign branching_expression
                           | forward_label_sequence [ or_expression ]
                           | forward_label_sequence
or_expression            ::= and_expression or or_expression
                           | and_expressnion
and_expression           ::= branching_condition and and_expression
                           | not_branching_condition and and_expression
                           | branching_condition
                           | not_branching_condition
not_branching_condition  ::= not branching_condition
branching_condition      ::= condition_label_sequence [ or_expression ]
                               branching_condition
                           | condition_label_sequence [ or_expression ]
                           | condition_label_sequence
condition_label_sequence ::= forward_sign label_sequence
                           | backward_sign label_sequence
```

In branching queries, a condition on a node with a given label holds if there exists a label sequence that fits to the condition. For example, the $root//a/b[\backslash c//d$ and not $\backslash e/f]/g$ query seeks nodes with label g such that the node can be reached from the root in such a way that the labels of the last two nodes are a and b, furthermore, there exists a parent of the node with label b whose label is c, and among the descendants of the node with label c there is one with label d, but it has no children with label e that has a parent with label f.

If we omit all conditions written between signs [ ] from a branching query, then we get the ***main query*** corresponding to the branching query. In our previous

example, this is the query $root//a/b/g$. The main query always corresponds to a forward label sequence.

A directed graph can be assigned naturally to branching queries. Assign nodes with the same label to the label sequence of the query, in case of separators / and \, connect the successive nodes with a directed edge according to the separator, and in case of separators // and \\, draw the directed edge and label it with label // or \\. Finally, the logic connectives are assigned to the starting edge of the corresponding condition as a label. Thus, it might happen that an edge has two labels, for example // and "and". Note that the graph obtained cannot contain a directed cycle because of the definition of the grammar.

A simple degree of complexity of the query can be defined using the tree obtained. Assign 0 to the nodes of the main query and to the nodes from which there is a directed path to a node of the main query. Then assign 1 to the nodes that can be reached from the nodes with sign 0 on a directed path and have no sign yet. Assign 2 to the nodes from which a node with sign 1 can be reached and have no sign yet. Assign 3 to the nodes that can be reached from nodes with sign 2 and have no sign yet, etc. Assign $2k+1$ to the nodes that can be reached from nodes with sign $2k$ and have no sign yet, then assign $2k+2$ to the nodes from which nodes with sign $2k+1$ can be reached and have no sign yet. The value of the greatest sign in the query is called the ***depth of the tree***. The depth of the tree shows how many times the direction changes during the evaluation of the query, that is, we have to seek children or parents according to the direction of the edges. The same query could have been given in different ways by composing the conditions differently, but it can be proved that the value defined above does not depend on that, that is why the complexity of a query was not defined as the number of conditions composed.

The 1-index assigns the nodes into classes according to incoming paths, using bisimulations. The concept of stability used for computations was ***descendant-stability***. A set A of the nodes of a graph is ***descendant-stable*** with respect to a set B of nodes if $A \subseteq E(B)$ or $A \cap E(B) = \emptyset$, where $E(B)$ is the set of nodes that can be reached by edges from B. A partition is stable if any two elements of the partition are descendant-stable with respect to each other. The 1-index is the coarsest descendant-stable partition that assigns nodes with the same label to same classes, which can be computed using the PT-algorithm. In case of branching queries, we also have to go backwards on directed edges, so we will need the concept of ***ancestor-stability*** as well. A set A of nodes of a graph is ***ancestor-stable*** with respect to a set B of the nodes if $A \subseteq E^{-1}(B)$ or $A \cap E^{-1}(B) = \emptyset$, where $E^{-1}(B)$ denotes the nodes from which a node of B can be reached.

Definition 20.41 *The **FB-index** is the coarsest refinement of the basic partition that is ancestor-stable and descendant-stable.*

Note that if the direction of the edges of the graph is reversed, then an ancestor-stable partition becomes a descendant-stable partition and vice versa, therefore the PT-algorithm and its improvements can be used to compute the coarsest ancestor-stable partition. We will use this in the following algorithm. We start with classes of nodes with the same label, compute the 1-index corresponding to this partition, then reverse the direction of the edges, and refine this by computing the 1-index

corresponding to this. When the algorithm stops, we get a refinement of the initial partition that is ancestor-stable and descendant-stable at the same time. This way we obtain the coarsest such partition. The proof of this is left to the Reader.

FB-INDEX-CREATOR(V, E)

```
1  P ← A(0)                          ▷ Start with classes of nodes with the same label.
2  while P changes
3        do P ← PT(V, E^{-1}, P)                          ▷ Compute the 1-index.
4           P ← PT(V, E, P)                               ▷ Reverse the direction
                                                          ▷ of edges, and compute the 1-index.
5  return P
```

The following corollary follows simply from the two stabilities.

Corollary 20.42 *The FB-index is safe and exact with respect to branching queries.*

The complexity of the algorithm can be computed from the complexity of the PT-algorithm. Since P is always the refinement of the previous partition, in the worst case refinement is done one by one, that is, we always take one element of a class and create a new class consisting of that element. So in the worst case, the cycle is repeated $O(n)$ times. Therefore, the cost of the algorithm is at most $O(mn \lg n)$.

The partition gained by executing the cycle only once is called the ***F+B-index***, the partition obtained by repeating the cycle twice is the ***F+B+F+B-index***, etc.

The following proposition can be proved easily.

Proposition 20.43 *The F+B+F+B+$\cdots$+F+B-index, where F+B appears d times, is safe and exact with respect to the branching queries of depth at most d.*

Nodes of the same class according to the FB-index cannot be distinguished by branching queries. This restriction is usually too strong, therefore the size of the FB-index is usually much smaller than the size of the original graph. Very long branching queries are seldom used in practice, so we only require local equivalence, similarly to the A(k)-index, but now we will describe it with two parameters depending on what we want to restrict: the length of the directed paths or the length of the paths with reversed direction. We can also restrict the depth of the query. We can introduce the FB(f, b, d)-index, with which such restricted branching queries can be evaluated exactly. We can also evaluate branching queries that do not satisfy the restrictions, but then the result must be checked.

FB(f,b,d)-INDEX-CREATOR(V,E,f,b,d)

```
1  P ← A(0)                          ▷ start with classes of nodes with the same label.
2  for i ← 1 to d
3      do P ← NAIVE-APPROXIMATION(V, E^{-1}, P, f) ▷ Compute the A(f)-index.
4         P ← NAIVE-APPROXIMATION(V, E, P, b)      ▷ Reverse the direction
                                                   ▷ of the edges, and compute the A(b)-index.
5  return P
```

The cost of the algorithm, based on the computation cost of the A(k)-index, is at most $O(dm\max(f,b))$, which is much better than the computation cost of the FB-index, and the index graph obtained is also usually much smaller.

The following proposition obviously holds for the index obtained.

Proposition 20.44 *The FB(f,b,d)-index is safe and exact for the branching queries in which the length of forward-sequences is at most f, the length of backward-sequences is at most b, and the depth of the tree corresponding to the query is at most d.*

As a special case we get that the FB(∞,∞,∞)-index is the FB-index, the FB(∞,∞,d)-index is the F+B+$\cdots$+F+B-index, where F+B appears d times, the FB$(\infty,0,1)$-index is the 1-index, and the FB$(k,0,1)$-index is the A(k)-index.

Exercises

20.7-1 Prove that the algorithm FB-INDEX-CREATOR produces the coarsest ancestor-stable and descendant-stable refinement of the basic partition.

20.7-2 Prove Proposition 20.44.

20.8. Index refresh

In database management we usually have three important aspects in mind. We want space requirement to be as small as possible, queries to be as fast as possible, and insertion, deletion and modification of the database to be as quick as possible. Generally, a result that is good with respect to one of these aspects is worse with respect to another aspect. By adding indexes of typical queries to the database, space requirement increases, but in return we can evaluate queries on indexes which makes them faster. In case of dynamic databases that are often modified we have to keep in mind that not only the original data but also the index has to be modified accordingly. The most costly method which is trivially exact is that we create the index again after every modification to the database. It is worth seeking procedures to get the modified indexes by smaller modifications to those indexes we already have.

Sometimes we index the index or its modification as well. The index of an index is also an index of the original graph, although formally it consists of classes of index nodes, but we can unite the elements of the index nodes belonging to the same class. It is easy to see that by that we get a partition of the nodes of the graph, that is, an index.

In the following, we will discuss those modifications of semi-structured databases when a new graph is attached to the root and when a new edges is added to the graph, since these are the ones we need when creating a new website or a new reference.

Suppose that $I(G)$ is the 1-index of graph G. Let H be a graph that has no common node with G. Denote by $I(H)$ the 1-index of H. Let $F = G + H$ be the graph obtained by uniting the roots of G and H. We want to create $I(G+H)$ using $I(G)$ and $I(H)$. The following proposition will help us.

Proposition 20.45 *Let $I(G)$ be the 1-index of graph G, and let J be an arbitrary refinement of $I(G)$. Then $I(J) = I(G)$.*

Proof Let u and v be two nodes of G. We have to show that u and v are bisimilar in G with respect to the 1-index if and only if $J(u)$ and $J(v)$ are bisimilar in the index graph $I(G)$ with respect to the 1-index of $I(G)$. Let u and v be bisimilar in G with respect to the 1-index. We will prove that there is a bisimulation according to which $J(u)$ and $J(v)$ are bisimilar in $I(G)$. Since the 1-index is the partition corresponding to the coarsest bisimulation, the given bisimulation is a refinement of the bisimulation corresponding to the 1-index, so $J(u)$ and $J(v)$ are also bisimilar with respect to the bisimulation corresponding to the 1-index of $I(G)$. Let $J(a) \approx' J(b)$ if and only if a and b are bisimilar in G with respect to the 1-index. Note that since J is a refinement of $I(G)$, all elements of $J(a)$ and $J(b)$ are bisimilar in G if $J(a) \approx' J(b)$. To show that the relation $\approx'$ is a bisimulation, let $J(u')$ be a parent of $J(u)$, where u' is a parent of u_1, and u_1 is an element of $J(u)$. Then u_1, u and v are bisimilar in G, so there is a parent v' of v for which u' and v' are bisimilar in G. Therefore $J(v')$ is a parent of $J(v)$, and $J(u') \approx' J(v')$. Since bisimulation is symmetric, relation $\approx'$ is also symmetric. We have proved the first part of the proposition.

Let $J(u)$ and $J(v)$ be bisimilar in $I(G)$ with respect to the 1-index of $I(G)$. It is sufficient to show that there is a bisimulation on the nodes of G according to which u and v are bisimilar. Let $a \approx' b$ if and only if $J(a) \approx J(b)$ with respect to the 1-index of $I(G)$. To prove bisimilarity, let u' be a parent of U. Then $J(u')$ is also a parent of $J(u)$. Since $J(u)$ and $J(v)$ are bisimilar if $u \approx' v$, there is a parent $J(v'')$ of $J(v)$ for which $J(u')$ and $J(v'')$ are bisimilar with respect to the 1-index of $I(G)$, and v'' is a parent of an element v_1 of $J(v)$. Since v and v_1 are bisimilar, there is a parent v' of v such that v' and v'' are bisimilar. Using the first part of the proof, it follows that $J(v')$ and $J(v'')$ are bisimilar with respect to the 1-index of $I(G)$. Since bisimilarity is transitive, $J(u')$ and $J(v')$ are bisimilar with respect to the 1-index of $I(G)$, so $u' \approx' v'$. Since relation $\approx'$ is symmetric by definition, we get a bisimulation. ■

As a consequence of this proposition, $I(G+H)$ can be created with the following algorithm for disjoint G and H.

GRAPHADDITION-1-INDEX(G, H)

```
1  P_G ← A_G(0)              ▷ P_G is the basic partition according to labels.
2  P_H ← A_H(0)              ▷ P_H is the basic partition according to labels.
3  I_1 ← PT(V_G, E_G^{-1}, P_G)   ▷ I_1 is the 1-index of G.
4  I_2 ← PT(V_H, E_H^{-1}, P_H)   ▷ I_2 is the 1-index of H.
5  J ← I_1 + I_2             ▷ The 1-indexes are joined at the roots.
6  P_J ← A_J(0)              ▷ P_J is the basic partition according to labels.
7  I ← PT(V_J, E_J^{-1}, PJ)      ▷ I is the 1-index of J.
8  return I
```

To compute the cost of the algorithm, suppose that the 1-index $I(G)$ of G is given. Then the cost of the creation of $I(G+H)$ is $O(m_H \lg n_H + (m_{I(H)} + m_{I(G)}) \lg(n_{I(G)} + n_{I(H)}))$, where n and m denote the number of nodes and edges of the graph, respectively.

To prove that the algorithm works, we only have to notice that $I(G) + I(H)$ is a refinement of $I(G+H)$ if G and H are disjoint. This also implies that index $I(G) + I(H)$ is safe and exact, so we can use this as well if we do not want to find the minimal index. This is especially useful if new graphs are added to our graph many times. In this case we use the ***lazy method***, that is, instead of computing the minimal index for every pair, we simply sum the indexes of the addends and then minimize only once.

Proposition 20.46 *Let $I(G_i)$ be the 1-index of graph G_i, $i = 1, \ldots, k$, and let the graphs be disjoint. Then $I(G_1 + \cdots + G_k) = I(I(G_1) + \cdots + I(G_k))$ for the 1-index $I(G_1 + \cdots + G_k)$ of the union of the graphs joined at the roots.*

In the following we will examine what happens to the index if a new edge is added to the graph. Even an operation like this can have significant effects. It is not difficult to construct a graph that contains two identical subgraphs at a distant of 2 from the root which cannot be contracted because of a missing edge. If we add this critical edge to the graph, then the two subgraphs can be contracted, and therefore the size of the index graph decreases to about the half of its original size.

Suppose we added a new edge to graph G from u to v. Denote the new graph by G', that is, $G' = G + (u, v)$. Let partition $I(G)$ be the 1-index of G. If there was an edge from $I(u)$ to $I(v)$ in $I(G)$, then the index graph does not have to be modified, since there is a parent of the elements of $I(v)$, that is, of all elements bisimilar to v, in $I(u)$ whose elements are bisimilar to u. Therefore $I(G') = I(G)$.

If there was no edge from $I(u)$ to $I(v)$, then we have to add this edge, but this might cause that $I(v)$ will no longer be stable with respect to $I(u)$. Let Q be the partition we get from $I(G)$ by splitting $I(v)$ in such a way that v is in one part and the other elements of $I(v)$ are in the other, and leaving all other classes of the partition unchanged. Q defines its edges the usual way, that is, if there is an edge from an element of a class to an element of another class, then we connect the two classes with an edge directed the same way.

Let partition X be the original $I(G)$. Then Q is a refinement of X, and Q is stable with respect to X according to G'. Note that the same invariant property

appeared in the PT-algorithm for partitions X and Q. Using Proposition 20.45 it is enough to find a refinement of $I(G')$. If we can find an arbitrary stable refinement of the basic partition of G', then, since the 1-index is the coarsest stable partition, this will be a refinement of $I(G')$. X is a refinement of the basic partition, that is, the partition according to labels, and so is Q. So if Q is stable, then we are done. If it is not, then we can stabilize it using the PT-algorithm by starting with the above partitions X and Q. First we have to examine those classes of the partition that contain a children of v, because these might lost their stability with respect to the two new classes gained by the split. The PT-algorithm stabilizes these by splitting them, but because of this we now have to check their children, since they might have lost stability because of the split, etc. We can obtain a stable refinement using this stability-propagator method. Since we only walk through the nodes that can be reached from v, this might not be the coarsest stable refinement. We have shown that the following algorithm computes the 1-index of the graph $G + (u, v)$.

EDGEADDITION-1-INDEX$(G,\ (u,\ v))$

1 $P_G \leftarrow A_G(0)$ ▷ P_G is the basic partition according to labels.
2 $I \leftarrow PT(V_G, E_G^{-1}, P_G)$ ▷ I is the 1-index of G.
3 $G' \leftarrow G + (u, v)$ ▷ Add edge (u, v).
4 **if** $(I(u), I(v)) \in E_I$ ▷ If there was an edge from $I(u)$ to $I(v)$, then no modification is needed.
5 **then return** I
6 $I' \leftarrow \{v\}$ ▷ Split $I(v)$.
7 $I'' \leftarrow I(v) \setminus \{v\}$
8 $X \leftarrow I$ ▷ X is the old partition.
9 $E_I \leftarrow E_I \cup \{(I(u), I(v))\}$ ▷ Add an edge from $I(u)$ to $I(v)$.
10 $Q \leftarrow (I \setminus \{I(v)\}) \cup \{I', I''\}$ ▷ Replace $I(v)$ with I' and I''.
11 $E \leftarrow E_Q$ ▷ Determine the edges of Q.
12 $J \leftarrow PT(V_{G'}, E_{G'}^{-1}, P_{G'}, X, Q)$ ▷ Execute the PT-algorithm ▷ starting with X and Q.
13 $J \leftarrow PT(V_J, E_J^{-1}, P_J)$ ▷ J is the coarsest stable refinement.
14 **return** J

Step 13 can be omitted in practice, since the stable refinement obtained in step 12 is a good enough approximation of the coarsest stable partition, there is only 5% difference between them in size.

In the following we will discuss how FB-indexes and A(k)-indexes can be refreshed. The difference between FB-indexes and 1-indexes is that in the FB-index, two nodes are in the same similarity class if not only the incoming but also the outgoing paths have the same label sequences. We saw that in order to create the FB-index we have to execute the PT-algorithm twice, using it on the graph with the edges reversed at the second time. The FB-index can be refreshed similarly to the 1-index. The following proposition can be proved similarly to Proposition 20.45, therefore we leave it to the Reader.

Proposition 20.47 *Let $I(G)$ be the FB-index of graph G, and let J be an arbitrary refinement of $I(G)$. Denote by $I(J)$ the FB-index of J. Then $I(J) = I(G)$.*

As a consequence of the above proposition, the FB-index of $G+H$ can be created using the following algorithm for disjoint G and H.

GRAPHADDITION-FB-INDEX(G, H)

```
1  I_1 ← FB-INDEX-CREATOR(V_G, E_G)          ▷ I_1 is the FB-index of G.
2  I_2 ← FB-INDEX-CREATOR(V_H, E_H)          ▷ I_2 is the FB-index of H.
3  J ← I_1 + I_2                             ▷ Join the FB-indexes at their roots.
4  I ← FB-INDEX-CREATOR(V_J, E_J)            ▷ I is the FB-index of J.
5  return I
```

When adding edge (u, v), we must keep in mind that stability can be lost in both directions, so not only $I(v)$ but also $I(u)$ has to be split into $\{v\}$, $(I \setminus \{v\})$ and $\{u\}$, $(I(u) \setminus \{u\})$, respectively. Let X be the partition before the modification, and Q the partition obtained after the splits. We start the PT-algorithm with X and Q in step 3 of the algorithm FB-INDEX-CREATOR. When stabilizing, we will now walk through all descendants of v and all ancestors of u.

EDGEADDITION-FB-INDEX(G, (u, v))

```
1  I ← FB-INDEX-CREATOR(V_G, E_G)            ▷ I is the FB-index of G.
2  G' ← G + (u, v)                           ▷ Add edge (u, v).
3  if (I(u), I(v)) ∈ E_I                     ▷ If there was an edge from I(u) to I(v),
                                             then no modification is needed.
4     then return I
5  I_1 ← {v}                                 ▷ Split I(v).
6  I_2 ← I(v) \ {v}
7  I_3 ← {u}                                 ▷ Split I(u).
8  I_4 ← I(u) \ {u}
9  X ← I                                     ▷ X is the old partition.
10 E_I ← E_I ∪ {(I(u), I(v))}                ▷ Add an edge form I(u) to I(v).
11 Q ← (I \ {I(v), I(u)}) ∪ {I_1, I_2, I_3, I_4}   ▷ Replace I(v) with I_1 and I_2,
                                                    I(u) with I_3 and I_4.
12 E ← E_Q                                   ▷ Determine the edges of Q.
13 J ← FB-INDEX-CREATOR(V_G', E_G', X, Q)    ▷ Start the PT-algorithm
                                             ▷ with X and Q in the algorithm FB-INDEX-CREATOR.
14 J ← FB-INDEX-CREATOR(V_J, E_J)            ▷ J is the coarsest ancestor-stable
                                             and descendant-stable refinement.
15 return J
```

Refreshing the A(k)-index after adding an edge is different than what we have seen. There is no problem with adding a graph though, since the following proposition holds, the proof of which is left to the Reader.

Proposition 20.48 *Let $I(G)$ be the $A(k)$-index of graph G, and let J be an arbitrary refinement of $I(G)$. Denote by $I(J)$ the $A(k)$-index of $I(J)$. Then $I(J) = I(G)$.*

As a consequence of the above proposition, the A(k)-index of $G + H$ can be created using the following algorithm for disjoint G and H.

GRAPHADDITION-A(k)-INDEX(G, H)

```
1  P_G ← A_G(0)                                  ▷ P_G is the basic partition according to labels.
2  I_1 ← NAIVE-APPROXIMATION(V_G, E_G^{-1}, P_G, k)   ▷ I_1 is the A(k)-index of G.
3  P_H ← A_H(0)                                  ▷ P_H is the basic partition according to labels.
4  I_2 ← NAIVE-APPROXIMATION(V_H, E_H^{-1}, P_H, k)   ▷ I_1 is the A(k)-index of H.
5  J ← I_1 + I_2                                 ▷ Join the A(k)-indexes.
6  P_J ← A_J(0)                                  ▷ P_J is the basic partition according to labels.
7  I ← NAIVE-APPROXIMATION(V_J, E_J^{-1}, P_J, k)     ▷ I is the A(k)-index of J.
8  return I
```

If we add a new edge (u, v) to the graph, then, as earlier, first we split $I(v)$ into two parts, one of which is $I' = \{v\}$, then we have to repair the lost k-stabilities walking through the descendants of v, but only within a distant of k. What causes the problem is that the A(k)-index contains information only about k-bisimilarity, it tells us nothing about $(k-1)$-bisimilarity. For example, let v_1 be a child of v, and let $k = 1$. When stabilizing according to the 1-index, v_1 has to be detached from its class if there is an element in this class that is not a children of v. This condition is too strong in case of the A(1)-index, and therefore it causes too many unnecessary splits. In this case, v_1 should only be detached if there is an element in its class that has no 0-bisimilar parent, that is, that has the same label as v. Because of this, if we refreshed the A(k)-index the above way when adding a new edge, we would get a very bad approximation of the A(k)-index belonging to the modification, so we use a different method. The main idea is to store all A(i)-indexes not only the A(k)-index, where $i = 1, \ldots, k$. Yi et al. give an algorithm based on this idea, and creates the A(k)-index belonging to the modification. The given algorithms can also be used for the deletion of edges with minor modifications, in case of 1-indexes and A(k)-indexes.

Exercises

20.8-1 Prove Proposition 20.47.

20.8-2 Give an algorithm for the modification of the index when an edge is deleted from the data graph. Examine different indexes. What is the cost of the algorithm?

20.8-3 Give algorithms for the modification of the D(k)-index when the data graph is modified.

Problems

20-1 Implication problem regarding constraints

Let R and Q be regular expressions, x and y two nodes. Let predicate $R(x, y)$ mean that y can be reached from x by a label sequence that fits to R. Denote by $R \subseteq Q$ the constraint $\forall x(R(\text{root}, x) \rightarrow Q(\text{root}, x))$. $R = Q$ if $R \subseteq Q$ and $Q \subseteq R$. Let C be a finite set of constraints, and c a constraint.

a. Prove that the implication problem $C \models c$ is a 2-EXPSPACE problem.

b. Denote by $R \subseteq Q@u$ the constraint $\forall v(R(u, v) \rightarrow Q(u, v))$. Prove that the implication problem is undecidable with respect to this class.

20-2 Transformational distance of trees

Let the *transformational distance* of vertex-labeled trees be the minimal number of basic operations with which a tree can be transformed to the other. We can use three basic operations: addition of a new node, deletion of a node, and renaming of a label.

a. Prove that the transformational distance of trees T and T' can be computed in $O(n_T n_{T'} d_T d_{T'})$ time, with storage cost of $O(n_T n_{T'})$, where n_T is the number of nodes of the tree and d_T is the depth of the tree.

b. Let S and S' be two trees. Give an algorithm that generates all pairs (T, T'), where T and T' simulates graphs S and S', respectively, and the transformational distance of T and T' is less then a given integer n. (This operation is called *approximate join*.)

20-3 Queries of distributed databases

A distributed database is a vertex-labeled directed graph the nodes of which are distributed in m partitions (servers). The edges between different partitions are *cross references*. Communication is by message broadcasting between the servers. An algorithm that evaluates a query is *efficient*, if the number of communication steps is constant, that is, it does not depend on the data and the query, and the size of the data transmitted during communication only depends on the size of the result of the query and the number of cross references. Prove that an efficient algorithm can be given for the regular query of distributed databases in which the number of communication steps is 4, and the size of data transmitted is $O(n^2) + O(k)$, where n is the size of the result of the query, and k is the number of cross references. (*Hint.* Try to modify the algorithm NAIVE-EVALUATION for this purpose.)

Chapter notes

This chapter examined those fields of the world of semi-structured databases where the morphisms of graphs could be used. Thus we discussed the creation of schemas and indexes from the algorithmic point of view. The world of semi-structured databases and XML is much broader than that. A short summary of the development, current issues and the possible future development of semi-structured databases can be found in the paper of Vianu [335].

The paper of M. Henzinger, T. Henzinger and Kopke [162] discusses the computation of the maximal simulation. They extend the concept of simulation to infinite graphs that can be represented efficiently (these are called effective graphs), and prove that for such graphs, it can be determined whether two nodes are similar. In their paper, Corneil and Gotlieb [74] deal with quotient graphs and the determination of isomorphism of graphs. Arenas and Libkin [18] extend normal forms used in the relational model to XML documents. They show that arbitrary DTD can be rewritten without loss as XNF, a normal form they introduced.

Buneman, Fernandez and Suciu [51] introduce a query language, the UnQL, based on structural recursion, where the data model used is defined by bisimulation. Gottlob, Koch and Pichler [138] examine the classes of the query language XPath with respect to complexity and parallelization. For an overview of complexity problems we recommend the classical work of Garey and Johnson [128] and the paper of Stockmeyer and Meyer [307].

The PT-algorithm was first published in the paper of Paige and Tarjan [264]. The 1-index based on bisimulations is discussed in detail by Milo and Suciu [246], where they also introduce the 2-index, and as a generalization of this, the T-index

The A(k)-index was introduced by Kaushik, Shenoy, Bohannon and Gudes [189]. The D(k)-index first appeared in the work of Chen, Lim and Ong [63]. The M(k)-index and the M*(k)-index, based on frequent queries, are the results of He and Yang [160]. FB-indexes of branching queries were first examined by Kaushik, Bohannon, Naughton and Korth [191]. The algorithms of the modifications of 1-indexes, FB-indexes and A(k)-indexes were summarized by Kaushik, Bohannon, Naughton and Shenoy [192]. The methods discussed here are improved and generalized in the work of Yi, He, Stanoi and Yang [351]. Polyzotis and Garafalakis use a probability model for the study of the selectivity of queries [274]. Kaushik, Krishnamurthy, Naughton and Ramakrishnan [190] suggest the combined use of structural indexes and inverted lists.

The book of Tucker [328] and the encyclopedia edited by Khosrow-Pour [196] deal with the use of XML in practice.

VI. APPLICATIONS

21. Bioinformatics

In this chapter at first we present algorithms on sequences, trees and stochastic grammars, then we continue with algorithms of comparison of structures and constructing of evolutionary trees, and finish the chapter with some rarely discussed topics of bioinformatics.

21.1. Algorithms on sequences

In this section, we are going to introduce dynamic programming algorithms working on sequences. Sequences are finite series of characters over a finite alphabet. The basic idea of dynamic programming is that calculations for long sequences can be given via calculations on substrings of the longer sequences.

The algorithms introduced here are the most important ones in bioinformatics, they are the basis of several software packages.

21.1.1. Distances of two sequences using linear gap penalty

DNA contains the information of living cells. Before the duplication of cells, the DNA molecules are doubled, and both daughter cells contain one copy of DNA. The replication of DNA is not perfect, the stored information can be changed by random mutations. Random mutations creates variants in the population, and these variants evolve to new species.

Given two sequences, we can ask the question how much the two species are related, and how many mutations are needed to describe the evolutionary history of the two sequences.

We suppose that mutations are independent from each other, and hence, the probability of a series of mutations is the product of probabilities of the mutations. Each mutation is associated with a weight, mutations with high probability get a smaller weight, mutations with low probability get a greater weight. A reasonable choice might be the logarithm of one over the probability of the mutation. In this case the weight of a series of mutations is the sum of the weights of the individual mutations. We also assume that mutation and its reverse have the same probability,

therefore we study how a sequence can be transfered into another instead of evolving two sequences from a common ancestor. Assuming minimum evolution ***minimum evolution*** , we are seeking for the minimum weight series of mutations that transforms one sequence into another. An important question is how we can quickly find such a minimum weight series. The naive algorithm finds all the possible series of mutations and chooses the minimum weight. Since the possible number of series of mutations grows exponentially – as we are going to show it in this chapter –, the naive algorithm is obviously too slow.

We are going to introduce the Sellers' algorithm [298]. Let Σ be a finite set of symbols, and let Σ^* denote the set of finite long sequences over Σ. The n long prefix of $A \in \Sigma^*$ will be denoted by A_n, and a_n denotes the nth character of A. The following transformations can be applied for a sequence:

- Insertion of symbol a before position i, denoted by $a \leftarrow^i -$.
- Deletion of symbol a at position i, denoted by $- \leftarrow^i a$.
- Substitution of symbol a to symbol b at position i, denoted by $b \leftarrow^i a$.

The concatenation of mutations is denoted by the $\circ$ symbol. τ denotes the set of finite long concatenations of the above mutations, and $T(A) = B$ denotes that $T \in \tau$ transforms a sequence A into sequence B. Let $w : \tau \to R^+ \cup \{0\}$ a weight function such that for any T_1, T_2 and S transformations satisfying

$$T_1 \circ T_2 = S \ , \tag{21.1}$$

the

$$w(T_1) + w(T_2) = w(S) \ , \tag{21.2}$$

equation also holds. Furthermore, let $w(a \leftarrow^i b)$ be independent from i. The transformation distance between two sequences, A and B, is the minimum weight of transformations transforming A into B:

$$\delta(A, B) = \min\{w(T) | (T(A) = B\} \ . \tag{21.3}$$

If we assume that w satisfies

$$w(a \leftarrow b) = w(b \leftarrow a) \ , \tag{21.4}$$

$$w(a \leftarrow a) = 0 \ , \tag{21.5}$$

$$w(b \leftarrow a) + w(c \leftarrow b) \geq w(c \leftarrow a) \tag{21.6}$$

for any a, b, $c \in \Sigma \cup \{-\}$, then the $\delta(,)$ transformation distance is indeed a metric on Σ^*.

Since $w(,)$ is a metric, it is enough to concern with transformations that change each position of a sequence at most once. Series of transformations are depicted with ***sequence alignments*** . By convention, the sequence at the top is the ancestor and the sequence at the bottom is its descendant. For example, the alignment below shows that there were substitutions at positions three and five, there was an insertion in the first position and a deletion in the eighth position.

```
- A U C G U A C A G
U A G C A U A - A G
```

A pair at a position is called aligned pair. The weight of the series of transformations described by the alignment is the sum of the weights of aligned pairs. Each series of mutations can be described by an alignment, and this description is unique up the permutation of mutations in the series. Since the summation is commutative, the weight of the series of mutations does not depend on the order of mutations.

We are going to show that the number of possible alignments also grows exponentially with the length of the sequences. The alignments that do not contain this pattern

```
# -
- #
```

where # an arbitrary character of Σ gives a subset of possible alignments. The size of this subset is $\binom{|A|+|B|}{|A|}$, since there is a bijection between this set of alignments and the set of coloured sequences that contains the characters of A and B in increasing order, and the characters of A is coloured with one colour, and the characters of B is coloured with the other colour. For example, if $|A| = |B| = n$, then $|A| + |B|\binom{|}{A}| = \Theta(2^{2n}/n^{0.5})$.

An alignment whose weight is minimal called an ***optimal alignment*** . Let the set of optimal alignments of A_i and B_j be denoted by $\alpha^*(A_i, B_j)$, and let $w(\alpha^*(A_i, B_j))$ denote the weights of any alignment in $\alpha^*(A_i, B_j)$.

The key of the fast algorithm for finding an optimal alignment is that if we know $w(\alpha^*(A_{i-1}, B_j))$, $w(\alpha^*(A_i, B_{j-1}))$, and $w(\alpha^*(A_{i-1}, B_{j-1}))$, then we can calculate $w(\alpha^*(A_i, j_m))$ in constant time. Indeed, if we delete the last aligned pair of an optimal alignment of A_i and B_j, we get the optimal alignment of A_{i-1} and B_j, or A_i and B_{j-1} or A_{i-1} and B_{j-1}, depending on the last aligned column depicts a deletion, an insertion, substitution or match, respectively. Hence,

$$\begin{aligned} w(\alpha^*(A_i, B_j)) = \quad & \min\{w(\alpha^*(A_{i-1}, B_j)) + w(- \leftarrow a_i); \\ & w(\alpha^*(A_i, B_{j-1})) + w(b_i \leftarrow -); \\ & w(\alpha^*(A_{i-1}, B_{j-1})) + w(b_i \leftarrow a_i)\}. \end{aligned} \tag{21.7}$$

The weights of optimal alignments are calculated in the so-called ***dynamic programming table*** , D. The $d_{i,j}$ element of D contains $w(\alpha^*(A_i, B_j))$. Comparing of an n and an m long sequence requires the fill-in of an $(n+1)x(m+1)$ table, indexing of rows and columns run from 0 till n and m, respectively. The initial conditions for column 0 and row 0 are

$$d_{0,0} = 0, \tag{21.8}$$

$$d_{i,0} = \sum_{k=1}^{i} w(- \leftarrow a_k) \ , \tag{21.9}$$

$$d_{0,j} = \sum_{l=1}^{j} w(b_l \leftarrow -) \ . \tag{21.10}$$

The table can be filled in using Equation (21.7). The time requirement for the fill-in is $\Theta(nm)$. After filling in the dynamic programming table, the set of all optimal alignments can be find in the following way, called ***trace-back*** . We go from the right bottom corner to the left top corner choosing the cell(s) giving the optimal value of the current cell (there might be more than one such cells). Stepping up from position $d_{i,j}$ means a deletion, stepping to the left means an insertion, and the diagonal steps mean either a substitution or a match depending on whether or not $a_i = b_j$. Each step is represented with an oriented edge, in this way, we get an oriented graph, whose vertices are a subset of the cells of the dynamic programming table. The number of optimal alignments might grow exponentially with the length of the sequences, however, the set of optimal alignments can be represented in polynomial time and space. Indeed, each path from $d_{n,m}$ to $d_{0,0}$ on the oriented graph obtained in the trace-back gives an optimal alignment.

21.1.2. Dynamic programming with arbitrary gap function

Since deletions and insertions get the same weight, the common name of them is indel or ***gap,*** and their weights are called ***gap penalty.*** Usually gap penalties do not depend on the deleted or inserted characters. The gap penalties used in the previous section grow linearly with the length of the gap. This means that a long indel is considered as the result of independent insertions or deletions of characters. However, the biological observation is that long indels can be formed in one evolutionary step, and these long indels are penalised too much with the linear gap penalty function. This observation motivated the introduction of more complex gap penalty functions [339]. The algorithm introduced by Waterman *et al.* penalises a k long gap with g_k. For example the weight of this alignment:

```
- - A U C G A C G U A C A G
U A G U C - - - A U A G A G
```

is $g_2 + w(G \leftarrow A) + g_3 + w(A \leftarrow G) + w(G \leftarrow C)$.

We are still seeking for the minimal weight series of transformations transforming one sequence into another or equivalently for an optimal alignment. Since there might be a long indel at the end of the optimal alignment, above knowing $w(\alpha^*(A_{i-1}, B_{j-1)})$, we must know all $w(\alpha^*(A_k, B_j))$, $0 \le k < i$ and $w(\alpha^*(A_i, B_l))$, $0 \le l < j$ to calculate $w(\alpha^*(A_i, B_j))$. The dynamic programming recursion is given by the following equations:

$$\begin{aligned} w(\alpha^*(A_i, B_j)) = \quad & \min\{w(\alpha^*(A_{i-1}, B_{j-1})) + w(b_j \leftarrow a_i) \ ; \\ & \min_{0 \le k < i}\{w(\alpha^*(A_k, B_j)) + g_{i-k}\} \ ; \\ & \min_{0 \le l < j}\{w(\alpha^*(A_i, B_l)) + g_{j-l}\}\} \ . \end{aligned} \tag{21.11}$$

The initial conditions are:

$$d_{0,0} = 0 \ , \tag{21.12}$$

$$d_{i,0} = g_i \ , \tag{21.13}$$

$$d_{0,j} = g_j \ . \tag{21.14}$$

The time requirement for calculating $d_{i,j}$ is $\Theta(i+j)$, hence the running time of the fill-in part to calculate the weight of an optimal alignment is $\Theta(nm(n+m))$. Similarly to the previous algorithm, the set of optimal alignments represented by paths from $d_{n,m}$ to $d_{0,0}$ can be found in the trace-back part.

If $|A| = |B| = n$, then the running time of this algorithm is $\Theta(n^3)$. With restrictions on the gap penalty function, the running time can be decreased. We are going to show two such algorithms in the next two sections.

21.1.3. Gotoh algorithm for affine gap penalty

A gap penalty function is ***affine*** if

$$g_k = u_k + v, \quad u \geq 0, \;\; v \geq 0 \; . \tag{21.15}$$

There exists a $\Theta(nm)$ running time algorithm for affine gap penalty [137]. Recall that in the Waterman algorithm,

$$d_{i,j} = \min\{d_{i-1,j-1} + w(b_j \leftarrow a_i);\; p_{i,j};\; q_{i,j}\} \; , \tag{21.16}$$

where

$$p_{i,j} = \min_{0 \leq k < i} \{d_{i-k,j} + g_k\} \; , \tag{21.17}$$

$$q_{i,j} = \min_{0 \leq l < j} \{d_{i,j-l} + g_l\} \; . \tag{21.18}$$

The key of the Gotoh algorithm is the following reindexing

$$\begin{aligned} p_{i,j} &= \min\{d_{i-1,j} + g_1,\; \min_{1 \leq k < i} \{d_{i-k,j} + g_k\}\} \\ &= \min\{d_{i-1,j} + g_1,\; \min_{0 \leq k < i-1} \{d_{i-1-k,j} + g_{k+1}\}\} \\ &= \min\{d_{i-1,j} + g_1,\; \min_{0 \leq k < i-1} \{d_{i-1-k,j} + g_k\} + u\} \\ &= \min\{d_{i-1,j} + g_1,\; p_{i-1,j} + u\} \; . \end{aligned} \tag{21.19}$$

And similarly

$$q_{i,j} = \min\{d_{i,j-1} + g_1,\; q_{i,j-1} + u\} \; . \tag{21.20}$$

In this way, $p_{i,j}$ és $q_{i,j}$ can be calculated in constant time, hence $d_{i,j}$. Thus, the running time of the algorithm remains $\Theta(nm)$, and the algorithm will be only a constant factor slower than the dynamic programming algorithm for linear gap penalties.

21.1.4. Concave gap penalty

There is no biological justification for the affine gap penalty function [37, 136], its wide-spread use (for example, CLUSTAL-W [322]) is due to its low running time. There is a more realistic gap penalty function for which an algorithm exists whose running time is slightly more than the running time for affine gap penalty, but it is still significantly better than the cubic running time algorithm of Waterman *et al.* [124, 245].

A gap penalty function is concave if for each i, $g_{i+1} - g_i \leq g_i - g_{i-1}$. Namely, the increasement of gap extensions are penalised less and less. It might happen that the function starts decreasing after a given point, to avoid this, it is usually assumed that the function increases monotonously. Based on empirical data [37], if two sequences evolved for d PAM unit [80], the weight of a q long indel is

$$35.03 - 6.88 \log_{10} d + 17.02 \log_{10} q \ , \tag{21.21}$$

which is also a concave function. (One PAM unit is the time span on which 1% of the sequence changed.) There exist an $O(nm(\log n + \log m))$ running time algorithm for concave gap penalty functions. this is a so-called ***forward looking*** algorithm. The FORWARD-LOOKING algorithm calculates the ith row of the dynamic programming table in the following way for an arbitrary gap penalty function:

FORWARD-LOOKING

```
1  for 1 ≤ j ≤ m
2      q1[i, j] ← d[i, 0] + g[j]
3      b[i, j] ← 0
4  for 1 ≤ j ≤ m
5      q[i, j] ← q1[i, j]
6      d[i, j] ← min[q[i, j], p[i, j], d[i − 1, j − 1] + w(b_j ← a_i)]
7  ▷ At this step, we suppose that p[i, j] and d[i − 1, j − 1] are already calculated.
8      for j < j1 ≤ m                                          ▷(inner cycle)
9          if q1[i, j1] < d[i, j] + g[j1 − j] then
10                                  q1[j1] ← d[i, j] + g[j1 − j]
11                                  b[i, j1] ← j
```

where $g[\,]$ is the gap penalty function and b is a pointer whose role will be described later. In row 6, we assume that we already calculated $p[i, j]$ and $d[i - 1, j - 1]$. It is easy to show that the forward looking algorithm makes the same comparisons as the traditional, backward looking algorithm, but in a different order. While the backward looking algorithm calculates $q_{i,j}$ at the jth position of the row looking back to the already calculated entries of the dynamic programming table, the FORWARD-LOOKING algorithm has already calculated $q_{i,j}$ by arriving to the jth position of the row. On the other hand, it sends forward candidate values for $q[i, j_1]$, $j_1 > j$, and by arriving to cell j_1, all the needed comparisons of candidate values have been made. Therefore, the FORWARD-LOOKING algorithm is not faster than the traditional backward looking algorithm, however, the conception helps accelerate the algorithm.

The key idea is the following.

Lemma 21.1 *Let j be the actual cell in the row. If*

$$d_{i,j} + g_{j_1 - j} \geq q_1[i, j_1] \ , \tag{21.22}$$

then for all $j_2 > j_1$

$$d_{i,j} + g_{j_2 - j} \geq q_1[i, j_2] \ . \tag{21.23}$$

Proof From the condition it follows that there is a $k < j < j_1 < j_2$ for which

$$d_{i,j} + g_{j_1-j} \geq d_{i,k} + g_{j_1-k} \ . \tag{21.24}$$

Let us add $g_{j_2-k} - g_{j_1-k}$ to the equation:

$$d_{i,j} + g_{j_1-j} + g_{j_2-k} - g_{j_1-k} \geq d_{i,k} + g_{j_2-k} \ . \tag{21.25}$$

For each concave gap penalty function,

$$g_{j_2-j} - g_{j_1-j} \geq g_{j_2-k} - g_{j_1-k} \ , \tag{21.26}$$

rearranging this and using Equation (21.25)

$$d_{i,j} + g_{j_2-j} \geq d_{i,j} + g_{j_1-j} + g_{j_2-k} - g_{j_1-k} \geq d_{i,k} + g_{j_2-k} \geq q[i, j_2] \tag{21.27}$$

■

The idea of the algorithm is to find the position with a binary search from where the actual cell cannot send forward optimal $q_{i,j}$ values. This is still not enough for the desired acceleration since $O(m)$ number of candidate values should be rewritten in the worst case. However, the corollary of the previous lemma leads to the desired acceleration:

Corollary 21.2 *Before the jth cell sends forward candidate values in the inner cycle of the forward looking algorithm, the cells after cell j form blocks, each block having the same pointer, and the pointer values are decreasing by blocks from left to right.*

The pseudocode of the algorithm is the following:

FORWARD-LOOKING-BINARY-SEARCHING(i, m, q, d, g, w, a, b)

```
1  pn[i] ← 0; p[i,0] ← m; b[i,0] ← 0
2  for j ← 1 to m
3      do q[i,j] ← q[i,b[i,pn[i]]] + g[j − b[i,pn[i]]]
4         d[i,j] ← min[q[i,j], p[i,j], d[i−1,j−1] + w(b_j ← a_i)]
5  ▷ At this step, we suppose that p[i,j] and d[i−1,j−1] are already calculated.
6         if p[i,pn[i]] = j then
7            then pn[i] − −
8         if j + 1 < m and d[i,b[i,0]] + g[m − b[i,0]] > d[i,j] + g[m − j]
9            then pn[i] ← 0; b[i,0] ← j
10           else if j + 1 < m
11                   then Y ← max_{0≤X≤pn[i]}{X | d[i,b[i,X]] + g[p[i,X]
                                 −b[i,X]] ≤ p[i,j] + g[p[i,X] − j]}
12           if d[i,b[i,Y]] + g[p[i,Y] − b[i,Y]] = p[i,j] + g[p[i,X] − j]
13              then pn[i] ← Y; b[i,Y] ← j
14              else E = p[i,Y]
15                   if Y < pn[i]
16                      then B ← p[i,Y + 1] − 1
17                      else B ← j + 1
```

18 $pn[i] + +$
19 $b[i, pn[i]] \leftarrow j$
20 $p[i, pn[i]] \leftarrow \max_{B \leq X \leq E}\{X | d[i, j] + g[X - j] \leq d[i, b[i, Y]] + g[x - b[i, Y]]\}$

The algorithm works in the following way: for each row, we maintain a variable storing the number of blocks, a list of positions of block ends, and a list of pointers for each block. For each cell j, the algorithm finds the last position for which the cell gives an optimal value using binary search. There is first a binary search for the blocks then for the positions inside the choosen block. It is enough to rewrite three values after the binary searches: the number of blocks, the end of the last block and its pointer. Therefore the most time consuming part is the binary search, which takes $O(\lg m)$ time for each cell.

We do the same for columns. If the dynamic programming table is filled in row by row, then for each position j in row i, the algorithms uses the block system of column j. Therefore the running time of the algorithm is $O(nm(\lg n + \lg m))$.

21.1.5. Similarity of two sequences, the Smith-Waterman algorithm

We can measure not only the distance but also the similarity of two sequences. For measuring the similarity of two characters, $S(a, b)$, the most frequently used function is the ***log-odds*** :

$$S(a, b) = \log\left(\frac{\Pr\{a, b\}}{\Pr\{a\}\Pr\{b\}}\right) , \tag{21.28}$$

where $\Pr\{a, b\}$ is the joint probability of the two characters (namely, the probability of observing them together in an alignment column), $\Pr\{a\}$ and $\Pr\{b\}$ are the marginal probabilities. The similarity is positive if $\Pr\{a, b\} > \Pr\{a\}\Pr\{b\}$, otherwise negative. Similarities are obtained from empirical data, for aminoacids, the most commonly used similarities are given by the PAM and BLOSUM matrices.

If we penalise gaps with negative numbers then the above described, global alignment algorithms work with similarities by changing minimalisation to maximalisation.

It is possible to define a special problem that works for similarities and does not work for distances. It is the local similarity problem or the local sequence alignment problem [304]. Given two sequences, a similarity and a gap penalty function, the problem is to give two substrings of the sequences whose similarity is maximal. A ***substring*** of a sequence is a consecutive part of the sequence. The biological motivation of the problem is that some parts of the biological sequences evolve slowly while other parts evolve fast. The local alignment finds the most conservated part of the two sequences. Local alignment is widely used for homology searching in databases. The reason why local alignments works well for homology searching is that the local alignment score can separate homologue and non-homologue sequences better since the statistics is not decreased due to the variable regions of the sequences.

The Smith-Waterman algorithm work in the following way. The initial conditions are:

$$d_{0,0} = d_{i,0} = d_{0,j} = 0 \; . \tag{21.29}$$

Considering linear gap penalty, the dynamic programming table is filled in using the following recursions:

$$d_{i,j} = \max\{0;\, d_{i-1,j-1} + S(a_i, b_j), d_{i-1,j} + g;\, d_{i,j-1} + g\} \; . \tag{21.30}$$

Here g, the gap penalty is a negative number. The best local similarity score of the two sequences is the maximal number in the table. The trace-back starts in the cell having the maximal number, and ends when first reaches a 0.

It is easy to prove that the alignment obtained in the trace-back will be locally optimal: if the alignment could be extended at the end with a sub-alignment whose similarity is positive then there would be a greater number in the dynamic programming table. If the alignment could be extended at the beginning with a subalignment having positive similarity then the value at the end of the traceback would not be 0.

21.1.6. Multiple sequence alignment

The multiple sequence alignment problem was introduced by David Sankoff [293], and by today, the multiple sequence alignment has been the central problem in bioinformatics. Dan Gusfield calls it the Holy Grail of bioinformatics. [148]. Multiple alignments are widespread both in searching databases and inferring evolutionary relationships. Using multiple alignments, it is possible to find the conservated parts of a sequence family, the positions that describe the functional properties of the sequence family. AS Arthur Lesk said: [168]: *What two sequences whisper, a multiple sequence alignment shout out loud.*

The columns of a multiple alignment of k sequences is called aligned k-tuples. The dynamic programming for the optimal multiple alignment is the generalisation of the dynamic programming for optimal pairwise alignment. To align k sequences, we have to fill in a k dimensional dynamic programming table. To calculate an entry in this table using linear gap penalty, we have to look back to a k dimensional hypercube. Therefore, the memory requirement in case of k sequence, n long each is $\Theta(n^k)$, and the running time of the algorithm is $\Theta(2^k n^k)$ if we use linear gap penalty, and $\Theta(n^{2k-1})$ with arbitrary gap penalty.

There are two fundamental problems with the multiple sequence alignment. The first is an algorithmic problem: it is proven that the multiple sequence alignment problem is NP-complete [337]. The other problem is methodical: it is not clear how to score a multiple alignment. An objective scoring function could be given only if the evolutionary relationships were known, in this case an aligned k-tuple could be scored according to an evolutionary tree [263].

A heuristic solution for both problems is the ***iterative sequence alignment*** [105],[75],[322]. This method first construct a ***guide-tree*** using pairwise distances (such tree-building methods are described in section 21.5). The guide-tree is used then to construct a multiple alignment. Each leaf is labelled with a sequence, and first the sequences in "cherry-motives" are aligned into each other, then sequence

alignments are aligned to sequences and sequence alignments according to the guide-tree. The iterative sequence alignment method uses the "once a gap – always gap" rule. This means that gaps already placed into an alignment cannot be modified when aligning the alignment to other alignment or sequence. The only possibility is to insert all-gap columns into an alignment. The aligned sequences are usually described with a ***profile*** . The profile is a $(|\Sigma| + 1) \times l$ table, where l is the length of the alignment. A column of a profile contains the statistics of the corresponding aligned k-tuple, the frequencies of characters and the gap symbol.

The obtained multiple alignment can be used for constructing another guide-tree, that can be used for another iterative sequence alignment, and this procedure can be iterated till convergence. The reason for the iterative alignment heuristic is that the optimal pairwise alignment of closely related sequences will be the same in the optimal multiple alignment. The drawback of the heuristic is that even if the previous assumption is true, there might be several optimal alignments for two sequences, and their number might grow exponentially with the length of the sequences. For example, let us consider the two optimal alignments of the sequences `AUCGGUACAG` and `AUCAUACAG`.

```
  A U C G G U A C A G     A U C G G U A C A G
A U C - A U A C A G     A U C A - U A C A G .
```

We cannot choose between the two alignments, however, in a multiple alignment, only one of them might be optimal. For example, if we align the sequence `AUCGAU` to the two optimal alignments, we get the following locally optimal alignments:

```
A U C G G U A C A G     A U C G G U A C A G
A U C - A U A C A G     A U C A - U A C A G
A U C G A U - - - -     A U C - G - A U - -
```

The left alignment is globally optimal, however, the right alignment is only locally optimal.

Hence, the iterative alignment method yields only a locally optimal alignment. Another problem of this method is that it does not give an upper bound for the goodness of the approximation. In spite of its drawback, the iterative alignment methods are the most widely used ones for multiple sequence alignments in practice, since it is fast and usually gives biologically reasonable alignments. Recently some approximation methods for multiple sequence alignment have been published with known upper bounds for their goodness. [147],[283]. However, the bounds biologically are not reasonable, and in practice, these methods usually give worse results than the heuristic methods.

We must mention a novel greedy method that is not based on dynamic programming. The DiAlign [249],[250],[248] first searches for gap-free homologue substrings by pairwise sequence comparison. The gap-free alignments of the homologous substrings are called diagonals of the dynamic programming name, hence the name of the method: Diagonal Alignment. The diagonals are scored according to their similarity value and diagonals that are not compatible with high-score diagonals get a penalty. Two diagonals are not compatible if they cannot be in the same alignment. After scoring the diagonals, they are aligned together a multiple alignment in a greedy

way. First the best diagonal is selected, then the best diagonal that is comparable with the first one, then the third best alignment that is comparable with the first two ones, etc. The multiple alignment is the union of the selected diagonals that might not cover all the characters in the sequence. Those characters that were not in any of the selected diagonals are marked as "non alignable". The drawback of the method is that sometimes it introduces too many gaps due to not penalising the gaps at all. However, DiAlign has been one of the best heuristic alignment approach and is widely used in the bioinformatics community.

21.1.7. Memory-reduction with the Hirschberg algorithm

If we want to calculate only the distance or similarity between two sequences and we are not interested in an optimal alignment, then in case of linear or affine gap penalties, it is very easy to construct an algorithm that uses only linear memory. Indeed, note that the dynamic programming recursion needs only the previous row (in case of filling in the dynamic table by rows), and the algorithm does not need to store earlier rows. On the other hand, once the dynamic programming table has reached the last row and forgot the earlier rows, it is not possible to trace-back the optimal alignment. If the dynamic programming table is scrolled again and again in linear memory to trace-back the optimal alignment row by row then the running time grows up to $O(n^3)$, where n is the length of the sequences.

However, it is possible to design an algorithm that obtains an optimal alignment in $O(n^2)$ running time and uses only linear memory. This is the ***Hirschberg algorithm*** [165], which we are going to introduce for distance-based alignment with linear gap penalty.

We introduce the suffixes of a sequence, a suffix is a substring ending at the end of the sequence. Let A^k denote the suffix of A starting with character a_{k+1}.

The Hirschberg algorithm first does a dynamic programming algorithm for sequences $A_{[|A|/2]}$ and B using liner memory as described above. Similarly, it does a dynamic programming algorithm for the reverse of the sequences $A^{[|A|/2]}$ and B.

Based on the two dynamic programming procedures, we know what is the score of the optimal alignment of $A_{[|A|/2]}$ and an arbitrary prefix of B, and similarly what is the score of the optimal alignment of $A^{[|A|/2]}$ and an arbitrary suffix of B. >From this we can tell what is the score of the optimal alignment of A and B:

$$\min_j \left\{ w(\alpha^*(A_{[|A|/2]}, B_j)) + w(\alpha^*(A^{[|A|/2]}, B^j)) \right\} , \tag{21.31}$$

and from this calculation it must be clear that in the optimal alignment of A and B, $A_{[|A|/2]}$ is aligned with the prefix B_j for which

$$w(\alpha^*(A_{[|A|/2]}, B_j)) + w(\alpha^*(A^{[|A|/2]}, B^j)) \tag{21.32}$$

is minimal.

Since we know the previous rows of the dynamic tables, we can tell if $a_{[|A|/2]}$ and $a_{[|A|/2]+1}$ is aligned with any characters of B or these characters are deleted in the optimal alignment. Similarly, we can tell if any character of B is inserted between $a_{[|A|/2]}$ and $a_{[|A|/2]+1}$.

In this way, we get at least two columns of the optimal alignment. Then we do the same for $A_{[|A|/2]-1}$ and the remaining part of the prefix of B, and for $A^{[|A|/2]+1}$ and the remaining part of the suffix of B. In this way we get alignment columns at the quarter and the three fourths of sequence A. In the next iteration, we do the same for the for pairs of sequences, etc., and we do the iteration till we get all the alignment columns of the optimal alignment.

Obviously, the memory requirement still only grows linearly with the length of the sequences. We show that the running time is still $\Theta(nm)$, where n and m are the lengths of the sequences. This comes from the fact that the running time of the first iteration is $|A| \times |B|$, the running time of the second iteration is $|A|/2) \times j^* + (|A|/2) \times (|B| - j^*$, where j^* is the position for which we get a minimum distance in Eqn. (21.31). Hence the total running time is:

$$nm \times \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) = \Theta(nm) \ . \tag{21.33}$$

21.1.8. Memory-reduction with corner-cutting

The dynamic programming algorithm reaches the optimal alignment of two sequences with aligning longer and longer prefixes of the two sequences. The algorithm can be accelerated with excluding the bad alignments of prefixes that cannot yield an optimal alignment. Such alignments are given with the ordered paths going from the right top and the left bottom corners to $d_{0,0}$, hence the name of the technique.

Most of the corner-cutting algorithms use a test value. This test value is an upper bound of the evolutionary distance between the two sequences. Corner-cutting algorithms using a test value can obtain the optimal alignment of two sequences only if the test value is indeed smaller then the distance between the two sequences, otherwise the algorithm stops before reaching the right bottom corner or gives a non-optimal alignment. Therefore these algorithms are useful for searching for sequences similar to a given one and we are not interested in sequences that are farther from the query sequence than the test value.

We are going to introduce two algorithms. the Spouge algorithm [305],[306] is a generalisation of the Fickett [107] and the Ukkonnen algorithm [330],[331]. The other algorithm was given by Gusfield, and this algorithm is an example for a corner-cutting algorithm that reaches the right bottom corner even if the distance between the two sequence is greater than the test value, but in this case the calculated score is bigger than the test value, indicating that the obtained alignment is not necessary optimal.

The Spouge algorithm calculates only those $d_{i,j}$ for which

$$d_{i,j} + |(n-i) - (m-j)| \times g \leq t \ , \tag{21.34}$$

where t is the test value, g is the gap penalty, n and m are the length of the sequences. The key observation of the algorithm is that any path going from $d_{i,j}$ to $d_{n,m}$ will increase the score of the alignment at least by $|(n-i) - (m-j)| \times g$. Therefore is t is smaller than the distance between the sequences, the Spouge algorithm obtains the optimal alignments, otherwise will stop before reaching the right bottom corner.

This algorithm is a generalisation of the Fickett algorithm and the Ukkonen algorithm. Those algorithms also use a test value, but the inequality in the Fickett algorithm is:

$$d_{i,j} \leq t \ , \tag{21.35}$$

while the inequality in the Ukkonnen algorithm is:

$$|i-j| \times g + |(n-i)-(m-j)| \times g \leq t \ . \tag{21.36}$$

Since in both cases, the left hand side of the inequalities are not greater than the left end side of the Spouge inequality, the Fickett and the Ukkonnen algorithms will calculate at least as much part of the dynamic programming table than the Spouge algorithm. Empirical results proved that the Spouge algorithm is significantly better [306]. The algorithm can be extended to affine and concave gap penalties, too.

The k-difference global alignment problem [148] asks the following question: Is there an alignment of the sequences whose weight is smaller than k? The algorithm answering the question has $O(kn)$ running time, where n is the length of the longer sequence. The algorithm is based on the observation that any path from $d_{n,m}$ to $d_{0,0}$ having at most score k cannot contain a cell $d_{i,j}$ for which $|i-j| > k/g$. Therefore the algorithm calculates only those $d_{i,j}$ cells for which $(i-j) < k/g$ and disregards the $d_{e,f}$ neighbours of the border elements for which $|e-f| > k/g$. If there exists an alignment with a score smaller or equal than k, then $d_{n,m} < k$ and $d_{n,m}$ is indeed the distance of the two sequences. Otherwise $d_{n,m} > k$, and $d_{n,m} > k$ is not necessary the score of the optimal alignment since there might be an alignment that leaves the band defined by the $|i-j| < k/g$ inequality and still has a smaller score then the best optimal alignment in the defined band.

The corner-cutting technique has been extended to multiple sequence alignments scored by the ***sum-of-pairs*** scoring scheme [58]. The sum-of-pairs score is:

$$SP_l = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} d(c_{i,l}, c_{j,l}) \ , \tag{21.37}$$

where SP_l is the lth aligned k-tuple $d(,)$ is the distance function on $\Sigma \cup \{-\}$, k is the number of sequences, $c_{i,j}$ is the character of the multiple alignment in the ith row and jth column. The l-suffix of sequence S is S^l. Let $w_{i,j}(l,m)$ denote the distance of the optimal alignment of the l-suffix and the m-suffix of the ith and the jth sequences. The Carillo and Lipman algorithm calculates only the positions for which

$$d_{i_1,i_2,\ldots i_n} + \sum_{j=1}^{k-1} \sum_{l=j}^{k} w_{j,l}(i_j, i_l) \leq t \ , \tag{21.38}$$

where t is the test value. The goodness of the algorithm follows from the fact that the sum-of-pairs score of the optimal alignment of the not yet aligned suffixes cannot be smaller than the sum of the scores of the optimal pairwise alignments. This corner cutting method can align at most six moderately long sequences [222].

Exercises

21.1-1 Show that the number of possible alignments of an n and an m long sequences is

$$\sum_{i=0}^{\min(n,m)} \frac{(n+m-i)!}{(n-i)!(m-i)!i!} .$$

21.1-2 Give a series of pairs of sequences and a scoring scheme such that the number of optimal alignments grows exponentially with the length of the sequences.
21.1-3 Give the Hirschberg algorithm for multiple alignments.
21.1-4 Give the Hirschberg algorithm for affine gap penalties.
21.1-5 Give the Smith-Waterman algorithm for affine gap penalties.
21.1-6 Give the Spouge algorithm for affine gap penalties.
21.1-7 Construct an example showing that the optimal multiple alignment of three sequences might contain a pairwise alignment that is only suboptimal.

21.2. Algorithms on trees

Algorithms introduced in this section work on rooted trees. The dynamic programming is based on the reduction to rooted subtrees. As we will see, above obtaining optimal cases, we can calculate algebraic expressions in the same running time.

21.2.1. The small parsimony problem

The (weighted) parsimony principle is to describe the changes of biological sequences with the minimum number (minimum weight) of mutations. We will concern only with substitutions, namely, the input sequences has the same length and the problem is to give the evolutionary relationships of sequences using only substitutions and the parsimony principle. We can define the large and the small parsimony problem. For the large parsimony problem, we do not know the topology of the evolutionary tree showing the evolutionary relationships of the sequences, hence the problem is to find both the best topology and an evolutionary history on the tree. The solution is not only locally but globally optimal. It has been proved that the large parsimony problem is NP-complete [119].

The small parsimony problem is to find the most parsimonious evolutionary history on a given tree topology. The solution for the small parsimony problem is only locally optimal, and there is no guarantee for global optimum.

Each position of the sequences is scored independently, therefore it is enough to find a solution for the case where there is only one character at each leaf of the tree. In this case, the evolutionary history can be described with labelling the internal nodes with characters. If two characters at neighbouring vertices are the same, then no mutation happened at the corresponding edge, otherwise one mutation happened. The naive algorithm investigates all possible labelings and selects the most parsimonious solution. Obviously, it is too slow, since the number of possible

labelings grows exponentially with the internal nodes of the tree.

The dynamic programming is based on the reduction to smaller subtrees [293]. Here the definition of subtrees is the following: there is a natural partial ordering on the nodes in the rooted subtree such that the root is the greatest node and the leaves are minimal. A subtree is defined by a node, and the subtree contains this node and all nodes that are smaller than the given node. The given node is the root of the subtree. We suppose that for any t child of the node r and any character ω we know the minimum number of mutations that are needed on the tree with root t given that there is ω at node t. Let $m_{t,\omega}$ denote this number. Then

$$m_{r,\omega} = \sum_{t \in D(r)} \min_{\sigma \in \Sigma} \{m_{t,\sigma} + \delta_{\omega,\sigma}\} \ , \tag{21.39}$$

where $D(r)$ is the set of children of r, Σ is the alphabet, and $\delta_{\omega,\sigma}$ is 1 if $\omega = \sigma$ and 0 otherwise.

The minimum number of mutations on the entire tree is $\min_{\omega \in \Sigma} m_{R,\omega}$, where R is the root of the tree. A most parsimonious labelling can be obtained with trace-backing the tree from the root to the leaves, writing to each nodes the character that minimises Eqn. 21.39. To do this, we have to store $m_{r,\omega}$ for all r and ω.

The running time of the algorithm is $\Theta(n|\Sigma|^2)$ for one character, where n is the number of nodes of the tree, and $\Theta(nl|\Sigma|^2)$ for entire sequences, where l is the length of the sequences.

21.2.2. The Felsenstein algorithm

The input of the ***Felsenstein algorithm*** [104] is a multiple alignment of DNA (or RNA or protein) sequences, an evolutionary tree topology and edge lengths, and a model that gives for each pair of characters, σ and ω and time t, what is the probability that σ evolves to ω duting time t. Let $f_{\sigma\omega}(t)$ denote this probability. The equilibrium probability distribution of the characters is denoted by π. The question is what is the likelihood of the tree, namely, what is the probability of observing the sequences at the leaves given the evolutionary parameters consisting of the edge lengths and parameters of the substitution model.

We assume that each position evolves independently, hence the probability of an evolutionary process is the product of the evolutionary probabilities for each position. Therefore it is enough to show how to calculate the likelihood for a sequence position. We show this for an example tree that can be seen on Fig. 21.1. s_i will denote the character at node i and v_j is the length of edge j. Since we do not know the characters at the internal nodes, we must sum the probabilities for all possible configurations:

$$\begin{aligned} L \ = \ & \sum_{s_0} \sum_{s_6} \sum_{s_7} \sum_{s_8} \pi_{s_0} \times f_{s_0 s_6}(v_6) \times f_{s_6 s_1}(v_1) \times f_{s_6 s_2}(v_2) \\ & \times f_{s_0 s_8}(v_8) \times f_{s_8 s_3}(v_3) \times f_{s_8 s_7}(v_7) \times f_{s_7 s_4}(v_4) \times f_{s_7 s_5}(v_5). \end{aligned} \tag{21.40}$$

If we consider the four character alphabet of DNA, the summation has 256 members, an in case of n species, it would have 4^{n-1}, namely the computational time grows exponentially with the number of sequences. However, if we move the expressions not

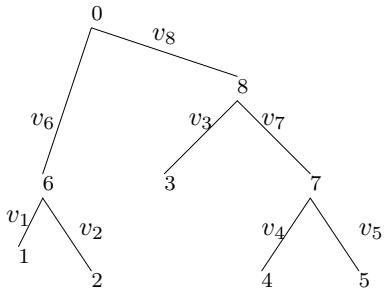


Figure 21.1. The tree on which we introduce the Felsenstein algorithm. Evolutionary times are denoted with vs on the edges of the tree.

depending on the summation index out of the summation, then we get the following product:

$$L = \sum_{s_0} \pi_{s_0} \left\{ \sum_{s_6} f_{s_0 s_6}(v_6)[f_{s_6 s_1}(v_1)][f_{s_6 s_2}(v_2)] \right\} \times \left\{ \sum_{s_8} f_{s_0 s_8}(v_8)[f_{s_8 s_3}(v_3)] \left(\sum_{s_7} f_{s_8 s_7}(v_7)[f_{s_7 s_4}(v_4)][f_{s_7 s_5}(v_5)] \right) \right\} \quad (21.41)$$

which can be calculated in significantly less time. Note that the parenthesis in Eqn. 21.41 gives the topology of the tree. Each summation can be calculated independently then we multiply the results. Hence the running time of calculating the likelihood for one position decreases to $\Theta(|\Sigma|^2 n)$ and the running time of calculating the likelihood for the multiple alignment is $\Theta(|\Sigma|^2 nl)$ where l is the length of the alignment.

Exercises

21.2-1 Give an algorithm for the weighted small parsimony problem where we want to get minimum weight evolutionary labeling given a tree topology and a set of sequences associated to the leaves of the tree.

21.2-2 The gene content changes in species, a gene that can be found in a genome of a species might be abundant in another genome. In the simplest model an existing gene might be deleted from the genome and an abundant gene might appear. Give the small parsimony algorithm for this gene content evolution model.

21.2-3 Give an algorithm that obtains the Maximum Likelihood labelling on a tree.

21.2-4 Rewrite the small parsimony problem in the form of Eqn. 21.40 replacing sums with minimalisation, and show that the Sankoff algorithm is based on the same rearrangement as the Felsenstein algorithm.

21.2-5 The Fitch algorithm [109] works in the following way: Each r node is associated with a set of characters, C_r. The leaves are associated with a set containing

the character associated to the leaves, and each internal character r has the set:

$$\begin{array}{ll} \cap_{t\in D(r)} C_t & \text{if} \quad \cap_{t\in D(r)} C_t \neq \emptyset \\ \cup_{t\in D(r)} C_t & \text{otherwise} \ , \end{array}$$

After reaching the root, we select an arbitrary character from C_R, where R is the root of the tree, and we choose the same character that we chose at the parent node if the set of the child node has this character, otherwise an arbitrary character from the set of the child node. Show that we get a most parsimonious labelling. What is the running time of this algorithm?

21.2-6 Show that the Sankoff algorithm gives all possible most parsimonious labelling, while there are most parsimonious labellings that cannot be obtained with the Fitch algorithm.

21.3. Algorithms on stochastic grammars

Below we give algorithms on stochastic transformational grammars. Stochastic transformational grammars play a central role in modern bioinformatics. Two types of transformational grammars are widespread, the Hidden Markov Models (HMMs) are used for protein structure prediction and gene finding, while Stochastic Context Free Grammars (SCFGs) are used for RNA secondary structure prediction.

21.3.1. Hidden Markov Models: Forward, Backward and the Viterbi algorithms

We give the formal definition of **H**idden **M**arkov **M**odels (HMM): Let X denote a finite set of states. There are two distinguished states among the states, the start and the end states. The states are divided into two parts, emitting and non-emitting states. We assume that only the start and the end states are non-emitting, we will show that this assumption is not too strict.

The $\mathbf{M}$ transformation matrix contains the transition probabilities, m_{ij}, that the Markov process will jump to state j from state i. Emitting states emit characters form a finite alphabet, Σ. The probability that the state i emits a character ω will be denoted by π^i_ω. The Markov process starts in the start state and ends in the end state, and walks according to the transition probabilities in $\mathbf{M}$. Each emitting state emits a character, the emitted characters form a sequence. The process is hidden since the observer observes only the sequence and does not observe the path that the Markov process walked on. There are three important questions for HMMs that can be answered using dynamic programming algorithms.

The first question is the following: given an HMM and a sequence, what is the most likely path that emits the given sequence? The Viterbi algorithm gives the answer for this question. Recall that A_k is the k-long prefix of sequence A, and a_k is the character in the kth position. The dynamic programming answering the first question is based on that we can calculate the $\Pr_{max}\{A_{k+1}, j\}$ probability, the probability of the most probable path emitting prefix A_{k+1} and being in state j if

we already calculated $\Pr_{max}\{\max\}(A_k, i)$ for all possible i, since

$$\Pr_{max}\{A_{k+1},\, j\} = \max_i(\Pr_{max}\{A_k,\, i\}\, m_{i,j}\pi^j_{a_{k+1}})\ . \tag{21.42}$$

The reason behind the above equation is that the probability of any path is the product of transition and emission probabilities. Among the products having the same last two terms (in our case $m_{i,j}\pi^j_{ak+1}$) the maximal is the one for which the product of the other terms is maximal.

The initialisation of the dynamic programming is

$$\Pr_{max}\{A_0, START\} = 1\ . \tag{21.43}$$

Since the end state does not emit a character, the termination of the dynamic programming algorithm is

$$\Pr_{max}\{A\} = \Pr_{max}\{A,\, END\} = \max_i(\Pr_{max}\{A,\, i\}\, m_{i,END})\ , \tag{21.44}$$

where $\Pr_{max}\{A\}$ is the probability of the most likely path emitting the given sequence. One of the most likely paths can be obtained with a trace-back.

The second question is the following: given an HMM and a sequence, what is the probability that the HMM emits the sequence? This probability is the sum of the probabilities of paths that emit the given sequence. Since the number of paths emitting a given sequence might grow exponentially with the length of the sequence, the naive algorithm that finds all the possible emitting paths and sum their probabilities would be too slow.

The dynamic programming algorithm that calculates quickly the probability in question is called the Forward algorithm. It is very similar to the Viterbi algorithm, just there is a sum instead of maximalisation in it:

$$\Pr\{A_{k+1},\, j\} = \sum_i \Pr\{A_k,\, i\}\, m_{i,j}\pi^j_{a_{k+1}}\ . \tag{21.45}$$

Since the *END* state does not emit, the termination is

$$\Pr\{A\} = \Pr\{A,\, END\} = \sum_i \Pr\{A,\, i\}\, m_{i,END}\ . \tag{21.46}$$

where $\Pr\{A\}$ is the probability that the HMM emits sequence A.

The most likely path obtained by the Viterbi algorithm has more and less reliable parts. Therefore we are interested in the probability

$$\Pr\{a_k \text{ is emitted by state } i \mid \text{the HMM emitted sequence } A\}\ .$$

This is the third question that we answer with dynamic programming algorithm. The above mentioned probability is the sum of the probabilities of paths that emit a_k in state i divided by the probability that the HMM emits sequence A. Since the number of such paths might grow exponentially, the naive algorithm that finds all the possible paths and sum their probability is too slow.

To answer the question, first we calculate for each suffix A^k and state i what

is the probability that the HMM emits suffix A^k given that state i emits a_k. This can be calculated with the Backward algorithm, which is similar to the Forward algorithm just starts the recursion with the end of the sequence:

$$\Pr\left\{A^k,\, i\right\} = \sum_j (\Pr\left\{A^{k+1},\, j\right\} m_{i,j} \pi^j_{a_{k+1}}) \,. \tag{21.47}$$

Let $\Pr\{a_k = i|A\}$ denote the probability

$$\Pr\{a_k \text{ is emitted by state } i \mid \text{the HMM emitted sequence } A\} \,.$$

Then

$$\Pr\{a_k = i|A\} \Pr\{A\} = \Pr\{A \wedge a_k = i\} = \Pr\{A_k,\, i\} \Pr\left\{A^k,\, i\right\} \,, \tag{21.48}$$

and from this

$$\Pr\{a_k = i|A\} = \frac{\Pr\{A_k,\, i\} \Pr\left\{A^k,\, i\right\}}{\Pr\{A\}} \,, \tag{21.49}$$

which is the needed probability.

21.3.2. Stochastic context-free grammars: the Inside, Outside and the CYK algorithms

It can be shown that every context-free grammar can be rewritten into *Chomsky normal form*. Each rule of a grammar in Chomsky normal form has the form $W_v \to W_y W_z$ or $W_w \to a$, where the Ws are non-terminal symbols, and a is a terminal symbol. In a stochastic grammar, each derivation rule has a probability, a non-negative number such that the probabilities of derivation rules for each non-terminal sum up to 1.

Given a SCFG and a sequence, we can ask the questions analogous to the three questions we asked for HMMs: what is the probability of the most likely derivation, what is the probability of the derivation of the sequence and what is the probability that a sub-string has been derivated starting with W_x non-terminal, given that the SCFG derivated the sequence. The first question can be answered with the CYK (Cocke-Younger-Kasami) algorithm which is the Viterbi-equivalent algorithm for SCFGs. The second question can be answered with the Inside algorithm, this is the Forward-equivalent for SCFGs. The third question can be answered with the combination of the Inside and Outside algorithms, as expected, the Outside algorithm is analogous to the Backward algorithm. Though the introduced algorithms are equivalent with the algorithms used for HMMs, their running time is significantly greater.

Let $t_v(y, z)$ denote the probability of the rule $W_v \to W_y W_z$, and let $e_v(a)$ denote the probability of the rule $W_v \to a$. The Inside algorithm calculates $\alpha(i, j, v)$ for all $i \leq j$ and v, this is the probability that non-terminal W_v derives the substring from a_i till a_j. The initial conditions are:

$$\alpha(i, i, v) = e_v(a_i) \,, \tag{21.50}$$

for all i and v. The main recursion is:

$$\alpha(i,j,v) = \sum_{y=1}^{M}\sum_{z=1}^{M}\sum_{k=i}^{j-1}\alpha(i,k,y)t_v(y,z)\alpha(k+1,j,z)\ , \tag{21.51}$$

where M is the number of non-terminals. The dynamic programming table is an upper triangle matrix for each non-terminal, the filling-in of the table starts with the main diagonal, and is continued with the other diagonals. The derivation probability is $\alpha(1,L,1)$, where L is the length of the sequence, and W_1 is the starting non-terminal. The running time of the algorithm is $\Theta(L^3M^3)$, the memory requirement is $\Theta(L^2M)$.

The Outside algorithm calculates $\beta(i,j,v)$ for all $i \leq j$ and v, this is the part of the derivation probability of deriving sequence A which is "outside" of the derivation of substring from a_i till a_j, starting the derivation from W_v. A more formal definition for $\beta(i,j,v)$ is that this is the sum of derivation probabilities in whom the substring from a_i till a_j is derived from W_v, divided by $\alpha(i,j,v)$. Here we define $0/0$ as 0. The initial conditions are:

$$\beta(1,L,1) = 1 \tag{21.52}$$

$$\beta(1,L,v) = 0 \quad \texttt{ha}\ v \neq 1\ . \tag{21.53}$$

The main recursion is:

$$\begin{aligned}\beta(i,j,v) \quad = \quad & \sum_{y=1}^{M}\sum_{z=1}^{M}\sum_{k=1}^{i-1}\alpha(k,i-1,z)t_y(z,v)\beta(k,j,y) + \\ & \sum_{y=1}^{M}\sum_{z=1}^{M}\sum_{k=j+1}^{L}\alpha(j+1,k,z)t_y(z,v)\beta(i,k,y)\ .\end{aligned} \tag{21.54}$$

The reasoning for Eqn. 21.54 is the following. The W_v non-terminal was derivated from a W_y non-terminal together with a W_z non-terminal, and their derivation order could be both W_zW_v and W_vW_z. The outside probability of non-terminal W_v is product of the outside probability of W_y, the derivation probability and the inside probability of W_z. As we can see, inside probabilities are needed to calculate outside probabilities, this is a significant difference from the Backward algorithm that can be used without a Forward algorithm.

The CYK algorithm is very similar to the Inside algorithm, just there are maximalisations instead of summations:

$$\alpha_{\max}(i,j,v) = \max_y \max_z \max_{i\leq k\leq j-1} \alpha_{\max}(i,k,y)t_v(y,z)\alpha_{\max}(k+1,j,z)\ , \tag{21.55}$$

The probability of the most likely derivation is $\alpha_{\max}(1,L,1)$. The most likely derivation can be obtained with a trace-back.

Finally, the probability that the substring from a_i till a_j has been derived by W_v given that the SCFG derived the sequence is:

$$\frac{\alpha(i,j,v)\beta(i,j,v)}{\alpha(1,L,1)}\ . \tag{21.56}$$

Exercises

21.3-1 In a regular grammar, each derivation rule is either in a form $W_v \to aW_y$ or in a form $W_v \to a$. Show that each HMM can be rewritten as a stochastic regular grammar. On the other hand, there are stochastic regular grammars that cannot be described as HMMs.

21.3-2 Give a dynamic programming algorithm that calculate for a stochastic regular grammar and a sequence A

- the most likely derivation,
- the probability of derivation,
- the probability that character a_i is derived by non-terminal W.

21.3-3 An HMM can contain silent states that do not emit any character. Show that any HMM containing silent states other than the start and end states can be rewritten to an HMM that does not contain silent states above the start and end states and emits sequences with the same probabilities.

21.3-4 Pair Hidden Markov models are Markov models in which states can emit characters not only to one but two sequences. Some states emit only into one of the sequences, some states emit into both sequences. The observer sees only the sequences and does not see which state emits which characters and which characters are co-emitted. Give the Viterbi, Forward and Backward algorithms for pair-HMMs.

21.3-5 The Viterbi algorithm does not use that probabilities are probabilities, namely, they are non-negative and sum up to one. Moreover, the Viterbi algorithm works if we replace multiplications to additions (say that we calculate the logarithm of the probabilities). Give a modified HMM, namely, in which "probabilities" not necessary sum up to one, and they might be negative, too, and the Viterbi algorithm with additions are equivalent with the Gotoh algorithm.

21.3-6 Secondary structures of RNA sequences are set of basepairings, in which for all basepairing positions $i\dot{j}$ and $i'\dot{j}'$, $i < i'$ implies that either $i < j < i' < j'$ or $i < i' < j' < i$. The possible basepairings are $A-U$, $U-A$, $C-G$, $G-C$, $G-U$ and $U-G$. Give a dynamic programming algorithm that finds the secondary structure containing the maximum number of basepairings for an RNA sequence. This problem was first solved by Nussionov *et al.* [255].

21.3-7 The derivation rules of the Knudsen-Hein grammar are [199], [200]

$$\begin{aligned} S &\to LS|L \\ F &\to dFd|LS \\ L &\to s|dFd \end{aligned}$$

where s has to be substituted with the possible characters of RNA sequences, and the ds in the dFd expression have to be replaced by possible basepairings. Show that the probability of the derivation of a sequence as well as the most likely derivation can be obtained without rewriting the grammar into Chomsky normal form.

21.4. Comparing structures

In this section, we give dynamic programming algorithms for comparing structures. As we can see, aligning labelled rooted trees is a generalisation of sequence alignment. The recursions in the dynamic programming algorithm for comparing HMMs yields a linear equation system due to circular dependencies. However, we still can call it dynamic programming algorithm.

21.4.1. Aligning labelled, rooted trees

Let Σ be a finite alphabet, and $\Sigma^- = \Sigma \cup \{-\}$, $\Sigma^2 = \Sigma^- \times \Sigma^- \backslash \{-,-\}$. Labelling of tree F is a function that assigns a character of Σ to each node $n \in V_F$. If we delete a node from the tree, then the children of the node will become children of the parental node. If we delete the root of the tree, then the tree becomes a forest. Let A be a rooted tree labelled with characters from Σ^2, and let $c : V_A \to \Sigma^2$ represent the labelling. A is an alignment of trees F and G labelled with characters from Σ if restricting the labeling of A to the first (respectively, second) coordinates and deleting nodes labelled with '$-$' yields tree F (respectively, G). Let $s : \Sigma^2 \to R$ be a similarity function. An optimal alignment of trees F and G is the tree A labelled with Σ^2 for which

$$\sum_{n \in V_A} s(c(n)) \tag{21.57}$$

is maximal. This tree is denoted by $A_{F,G}$. Note that a sequence can be represented with a unary tree, which has a single leaf. Therefore aligning trees is a generalisation of aligning sequences (with linear gap penalty).

Below we will concern only with trees in which each node has a degree at most 3. The recursion in the dynamic programming algorithm goes on rooted subtrees. A rooted subtree of a tree contains a node n of the tree and all nodes that are smaller than n. The tree obtained by root r is denoted by t_r.

A tree to an empty tree can be aligned only in one way. Two leafes labelled by a and b can be aligned in three different way. The alignment might contain only one node labelled with (a, b) or might contain two nodes, one of them is labelled with $(a, -)$, the other with $(-, b)$. One of the points is the root, the other the leaf.

Similarly, when we align a single leaf to a tree, then in the alignment A either the single character of the node is labelled together with a character of the tree or labelled together with '$-$' in an independent node. This node can be placed in several ways on tree A, however the score of any of them is the same.

After this initialisation, the dynamic programming algorithm aligns greater rooted subtrees using the alignments of smaller rooted subtrees. We assume that we already know the score of the optimal alignments A_{t_r,t_x}, A_{t_r,t_y}, A_{t_u,t_s}, A_{t_v,t_s}, A_{t_u,t_x}, A_{t_u,t_y}, A_{t_v,t_x} and A_{t_v,t_y} when aligning subtrees t_r and t_s, where u and v are the children of r and x and y are the children of s. Should one of the nodes have only one child, the dynamic programming reduces the problem of aligning t_r and t_s to less subproblems. We assume that the algorithm also knows the score of the optimal alignments of t_r to the empty tree and the score of the optimal alignment of t_s to the empty tree. Let the labelling of r be a and the labelling of s be b. We have to

consider constant many subproblems for obtaining the score of the optimal alignment of t_r and t_s. If one of the tree is aligned to one of the children's subtree of the other tree, then the other child and the root of the other tree is labelled together with '$-$'. If character of r is co-labelled with the character of s, then the children nodes are aligned together, as well. The last situation is the case when the roots are not aligned in A_{t_r,t_s} but one of the roots is the root of A_{t_r,t_s} and the other root is its only child. The children might or might not be aligned together, this is five possible cases altogether.

Since the number of rooted subtrees equals to the number of nodes of the tree, the optimal alignment can be obtained in $\Theta(|F||G|)$ time, where $|F|$ and $|G|$ are the number of nodes in F and G.

21.4.2. Co-emission probability of two HMMs

Let M_1 and M_2 be Hidden Markov Models. The co-emission probability of the two models is

$$C(M_1, M_2) = \sum_s \Pr_{M_1}\{s\} \Pr_{M_2}\{s\} \ , \tag{21.58}$$

where the summation is over all possible sequences and $\Pr_M\{s\}$ is the probability that model M emitted sequence s. The probability that path p emitted sequence s is denoted by $e(p) = s$, a path from the $START$ state till the x state is denoted by $[x]$. Since state x can be reached on several paths, this definition is not well-defined, however, this will not cause a problem later on. Since the coemission probability is the sum of the product of emission of paths,

$$\begin{aligned} C(M_1, M_2) &= \sum_s \left(\sum_{p_1 \in M_1, e(p_1)=s} \Pr_{M_1}\{p_1\} \right) \left(\sum_{p_2 \in M_2, e(p_2)=s} \Pr_{M_2}\{p_2\} \right) = \\ &= \sum_{p_1 \in M_1, p_2 \in M_2, e(p_1)=e(p_2)} \Pr_{M_1}\{p_1\} \Pr_{M_2}\{p_2\} \ . \end{aligned} \tag{21.59}$$

Let $\bar{p}_1$ denote the path that can be obtained with removing the last state from p_1, and let x_1 be the state before END_1 in path p_1. (We define similarly $\bar{p}_2$ and x_2.) Hence

$$\begin{aligned} C(M_1, M_2) &= \sum_{p_1 \in M_1, p_2 \in M_2, e(p_1)=e(p_2)} m_{x_1, END_1} m_{x_2, END_2} \Pr_{M_1}\{\bar{p}_1\} \Pr_{M_2}\{\bar{p}_2\} = \\ &= \sum_{x_1, x_2} m_{x_1, END_1} m_{x_2, END_2} C(x_1, x_2) \ , \end{aligned} \tag{21.60}$$

where $m_{x,END}$ is the probability of jumping to $END$from x, and

$$C(x_1, x_2) = \sum_{[x_1] \in M_1, [x_2] \in M_2, e([x_1])=e([x_2])} \Pr_{M_1}\{[x_1]\} \Pr_{M_2}\{[x_2]\} \, . \tag{21.61}$$

$C(x_1, x_2)$ can be also obtained with this equation:

$$C(x_1, x_2) = \sum_{y_1, y_2} m_{y_1, x_1} m_{y_2, x_2} C(y_1, y_2) \sum_{\sigma \in \Sigma} \Pr\{\sigma | x_1\} \Pr\{\sigma | x_2\} \ , \tag{21.62}$$

where $\Pr\{\sigma|x_i\}$ is the probability that x_i emitted σ. Equation 21.62 defines a linear equation system for all pairs of emitting states x_1 and x_2 . The initial conditions are:

$$C(START_1, START_2) = 1, \tag{21.63}$$
$$C(START_1, x_2) = 0, \quad x_2 \neq START_2\ , \tag{21.64}$$
$$C(x_1, START_2) = 0, \quad x_1 \neq START_1\ . \tag{21.65}$$

Unlike the case of traditional dynamic programming, we do not fill in a dynamic programming table, but solve a linear equation system defined by Equation 21.62. Hence, the coemission probability can be calculated in $O\left((n_1 n_2)^3\right)$ time, where n_i and M_i are the number of emitting states of the models.

Exercises

21.4-1 Give a dynamic programming algorithm for the local similarities of two trees. This is the score of the most similar subtrees of the two trees. Here subtrees are any consecutive parts of the tree.

21.4-2 Ordered trees are rooted trees in which the children of a node are ordered. The ordered alignment of two ordered trees preserve the orderings in the aligned trees. Give an algorithm that obtains the optimal ordered alignment of two ordered trees and has running time being polynomial with both the maximum number of children and number of nodes.

21.4-3 Consider the infinite Euclidean space whose coordinates are the possible sequences. Each Hidden Markov model is a vector in this space the coordinates of the vector are the emission probabilities of the corresponding sequences. Obtain the angle between two HMMs in this space.

21.4-4 Give an algorithm that calculates the generating function of the length of the emitted sequences of an HMM, that is

$$\sum_{i=0}^{\infty} p_i \xi^i$$

where p_i is the probability that the Markov model emitted a sequence with length i.

21.4-5 Give an algorithm that calculates the generating function of the length of the emitted sequences of a pair-HMM, that is

$$\sum_{i=0}^{\infty} \sum_{j=0}^{\infty} p_{i,j} \xi^i \eta^j$$

where $p_{i,j}$ is the probability that the first emitted sequence has length i, and the second emitted sequence has length j.

21.5. Distance based algorithms for constructing evolutionary trees

In this section, we shell introduce algorithms whose input is a set of objects and distances between objects. The distances might be obtained from pairwise alignments of sequences, however, the introduced algorithms work for any kind of distances. The leaves of the tree are the given objects, and the topology and the lengths of the edges are obtained from the distances. Every weighted tree defines a metric on the leaves of the tree, we define the distance between two leaves as the sum of the weights of edges on the path connecting them. The goodness of algorithms can be measured as the deviation between the input distances and the distances obtained on the tree.

We define two special metrics, the ultrametric and additive metric. The clustering algorithms generate a tree that is always ultrametric. We shell prove that clustering algorithms gives back the ultrametric if the input distances follow a ultrametric, namely, the tree obtained by a clustering algorithm defines exactly the input distances.

Similarly, the Neighbour Joining algorithm creates a tree that represents an additive metric, and whenever the input distances follow an additive metric, the generated tree gives back the input distances.

For both proves, we need the following lemma:

Lemma 21.3 *For any metric, there is at most one tree that represents it and has positive weights.*

Proof The proof is based on induction, the induction starts with three points. For three points, there is exactly one possible topology, a star-tree. Let the lengths of the edges connecting points i, j and k with the internal node of the star three be x, y and z, respectively. The lengths of the edges defined by the

$$x + y = d_{i,j} \tag{21.66}$$

$$x + z = d_{i,k} \tag{21.67}$$

$$y + z = d_{k,l} \tag{21.68}$$

equation system, which has a unique solution since the determinant

$$\begin{vmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{vmatrix} \tag{21.69}$$

is not 0.

For $n > 3$ number of points, let us assume that there are two trees representing the same metric. We find a ***cherry motif*** on the first tree, with cherries i and j. A cherry motif is a motif with two leafes whose connecting path has exactly one internal node. Every tree contains at least two cherry motives, a path on the tree that has the maximal number of internal nodes has cherry motives at both ends.

If there is only one internal node on the path connecting i and j on the other tree, then the length of the corresponding edges in the two cherry motives must be the same, since for any additional point k, we must get the same subtree. We

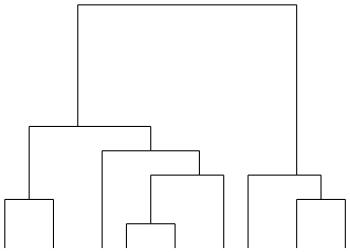


Figure 21.2. A dendrogram.

define a new metric by deleting points i and j, and adding a new point u. The distance between u and any point k is $d_{i,k} - d_{i,u}$, where $d_{i,u}$ is the length of the edge connecting i with the internal point in the cherry motif. If we delete nodes i and j, we get a tree that represent this metric and they are the same, according to the induction.

If the path between i and j contains more than one internal node on the other tree, then we find a contradiction. There is a u_1 point on the second tree for which $d_{i,u} \neq d_{i,u_1}$. Consider a k such that the path connecting i and k contains node u. From the first tree

$$d_{i,k} - d_{j,k} = d_{i,u} - d_{j,u} = 2d_{i,u} - d_{i,j} \ , \tag{21.70}$$

while on the second tree

$$d_{i,k} - d_{j,k} = d_{i,u_1} - d_{j,u_1} = 2d_{i,u_1} - d_{i,j} \ , \tag{21.71}$$

which contradicts that $d_{i,u} \neq d_{i,u_1}$. ■

21.5.1. Clustering algorithms

Definition 21.4 *A metric is* ***ultrametric*** *if for any three points, i, j and k*

$$d_{i,j} \leq \max\{d_{i,k}, d_{j,k}\} \tag{21.72}$$

It is easy to prove that the three distances between any three points are all equal or two of them equal and the third is smaller in any ultrametric.

Theorem 21.5 *If the metric on a finite set of points is ultrametric, then there is exactly one tree that represents it. Furthermore, this tree can be rooted such that the distance between a point and the root is the same for all points.*

Proof Based on the Lemma 21.3, it is enough to construct one ultrametric tree for any ultrametric. We represent ultrametric trees as ***dendrograms*** , in this representation, the horizontal edges has length zero. For an example dendrogram, see Fig. 21.2. The proof is based on the induction on the number of leaves. Obviously, we can construct a dendrogram for two leaves. After constructing the dendrogram for n

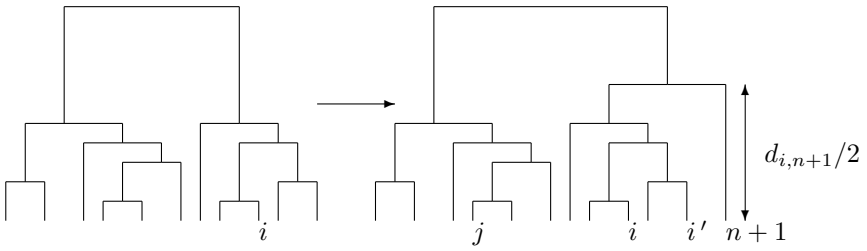


Figure 21.3. Connecting leaf $n+1$ to the dendrogram.

leaves, we add leaf $n+1$ to the dendrogram in the following way. We find a leaf i in the dendrogram, for which $d_{i,n+1}$ is minimal. Then we walk up on the dendrogram till we reach the $d_{i,n+1}/2$ distance (we might go upper than the root). The node i is connected to the dendrogram at this point, see Fig. 21.3. This dendrogram represents properly the distances between leaf $n+1$ and any other leaf. Indeed, if leaf i that is below the new internal node that bonnets leaf $n+1$, then $d_{i,i'} \leq d_{i,n+1}$ and from the ultrametric property and the minimality of $d_{i,n+1}$ it follows that $d_{i,n+1} = d_{i',n+1}$. On the other hand, if leaf j is not below the new internal point joining leaf $n+1$, then $d_{i,j} > d_{i,n+1}$, and from the ultrametric property it comes that $d_{j,n+1} = d_{i,j}$. ■
It is easy to see that the construction in the proof needs $O(n^2)$ running time, where n is the number of objects. We shell give another algorithm that finds the pair of objects i and j for which $d_{i,j}$ is minimal. From the ultrametric property, for any $k \neq i, j$, $d_{i,k} = d_{j,k} (\geq d_{i,j})$, hence we can replace the pair of objects i and j to a new object, and the distance between this new object and any other object k is well defined, it is $d_{i,k} = d_{j,k}$. The objects i and j are connected at height $d_{i,j}/2$, and we treat this sub-dendrogram as a single object. We continue the iteration till we have a single object. This algorithm is slower than the previous algorithm, however, this is the basis of the clustering algorithms. The clustering algorithms create a dendrogram even if the input distances do not follow a ultrametric. On the other hand, if the input distances follow a ultrametric, then most of the clustering algorithms gives back this ultrametric.

As we mentioned, the clustering algorithms find the object pair i and j for which $d_{i,j}$ is minimal. The differences come from how the algorithms define the distance between the new object replacing the pair of objects i and j and any other object. If the new object is denoted by u, then the introduced clustering methods define $d_{u,k}$ in the following way:

- SINGLE LINK: $d_{u,k} = \min\{d_{i,k}, d_{j,k}\}$.
- COMPLETE LINK: $d_{u,k} = \max\{d_{i,k}, d_{j,k}\}$.
- UPGMA: the new distance is the arithmetic mean of the distances between the elements in u and k : $d_{u,k} = \frac{d_{i,k} \times |i| + d_{j,k} \times |j|}{|i| + |j|}$, where $|i|$ and $|j|$ are the number of elements in i and j.
- SINGLE AVERAGE: $d_{u,k} = \frac{d_{i,k} + d_{j,k}}{2}$.
- CENTROID: This method is used when the objects can be embedded into a

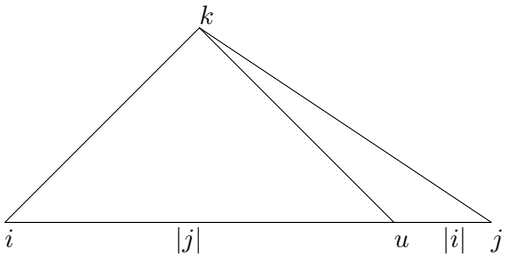


Figure 21.4. Calculating $d_{u,k}$ according to the CENTROID method.

Euclidean space. Then the distance between two objects can be defined as the distance between the centroids of the elements of the objects. It is not necessary to use the coordinates of the Euclidean space since the distance $d_{u,k}$ in question is the distance between point k and the point intersecting the ij edge in $|j| : |i|$ proportion in the triangle obtained by points i, j és k (see Fig. 21.4). This length can be calculated using only $d_{i,j}$, $d_{i,k}$ and $d_{j,k}$. Hence the algorithm can be used even if the objects cannot be embedded into a Euclidean space.

- MEDIAN: The centroid of u is the centroid of the centroids of i and j. This method is related to the centroid method as the single average is related to the UPGMA method. It is again not necessary to know the coordinates of the elements, hence this method can be applied to distances that cannot be embedded into a Euclidean space.

It is easy to show that the first four method gives the dendrogram of the input distances whenever the input distances follow a ultrametric since $d_{i,k} = d_{j,k}$ in this case. However, the CENTROID and MEDIAN methods do not give the corresponding dendrogram for ultrametric input distances, since $d_{u,k}$ will be smaller than $d_{i,k}$ (which equals to $d_{j,k}$).

The central problem of the clustering algorithms is that they give a dendrogram that might not be biologically correct. Indeed, the evolutionary tree of biological sequences can be a dendrogram only if the *molecular clock* hypothesis holds. The molecular clock hypothesis says that the sequences evolve with the same tempo at each branches of the tree, namely they collect the same number of mutations at a given time span. However, this is usually not true. Therefore biologists want an algorithm that give a ultrametric tree only if the input distances follow a ultrametric. The most popular such method is the ***Neighbour-Joining*** algorithm.

21.5.2. Neighbour joining

Definition 21.6 *A metric is called* ***additive*** *or* ***four-point metric****, if for any four points i, j, k and l*

$$d_{i,j} + d_{k,l} \leq \max\{d_{i,k} + d_{j,l},\, d_{i,l} + d_{j,k}\} \tag{21.73}$$

Theorem 21.7 *If a metric is additive on a finite set of objects, then there is exactly one tree that represents it.*

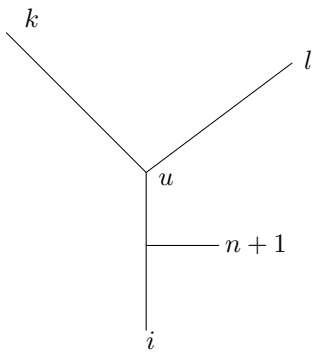


Figure 21.5. Connecting leaf $n+1$ for constructing an additive tree.

Proof Due to Lemma 21.3, there is at most one such tree, therefore it is enough to construct it. First we give the construction then we prove its goodness.

For three objects we can construct a tree according to Equations 21.66–21.68. Assume that we constructed the tree for $n \geq 3$ objects, and we want to add leaf $n+1$ to the tree. First we find the topology and then we give the length of the new edge. For obtaining the new topology we start with any leaf i, and let denote u the neighbour of leaf i. There are at least two other edges going out from u, we find two leaves on the paths starting with these two outgoing edges, let k and l denote these leaves, see Fig. 21.5. The leaf is connected to the edges between i and u if

$$d_{i,n+1} + d_{k,l} < d_{i,k} + d_{n+1,l} \tag{21.74}$$

Using similar inequalities, we can decide if leaf $n+1$ is before u looking from k or looking from l. If the degree of u is greater than 3, then we find leaves l' on the other paths and we do the same investigations for i, $n+1$, k and l' points. >From the additive property, it follows that inequality can hold at most for one cases. If it holds for i, then we connect leaf $n+1$ to the edge connecting u and i. If the inequality holds for another case, then we derive the maximal subtree of the tree that contains u as a leaf and also contains the leaf for which the inequality holds. We define $d_{u,n+1}$ as $d_{i,n+1} - d_{i,u}$, then renaming u to i we continue the searching for the connection place of leaf $n+1$. If we get equality for all outgoing edges of u, then we connect leaf $n+1$ to u.

After finding the topology, we obtain the length of the new edge. Leaf $n+1$ is connected to the edge between i and u, let u_1 denote the new internal point, see Fig. 21.6/b. We define $d_{u,n+1}$ as $d_{l,n+1} - d_{l,u}$. then the distances d_{u,u_1}, d_{i,u_1}, and $d_{u_1,n+1}$ can be calculated using Equations 21.66–21.68. If the leaf $n+1$ is connected to u, then $d_{u,n+1} = d_{i,n+1} - d_{i,u}$.

Now we prove the correctness of the construction. First we show that $d_{u,n+1}$ is well-defined, namely, for all node j that is not in the new subtree containing leaves $n+1$ and u, $d_{j,n+1} - d_{j,u} = d_{i,n+1} - d_{i,u}$. If the new subtree contains l then for $j = k$ that was used to find the place of leaf $n+1$ will obviously hold (see Fig. 21.6/a).

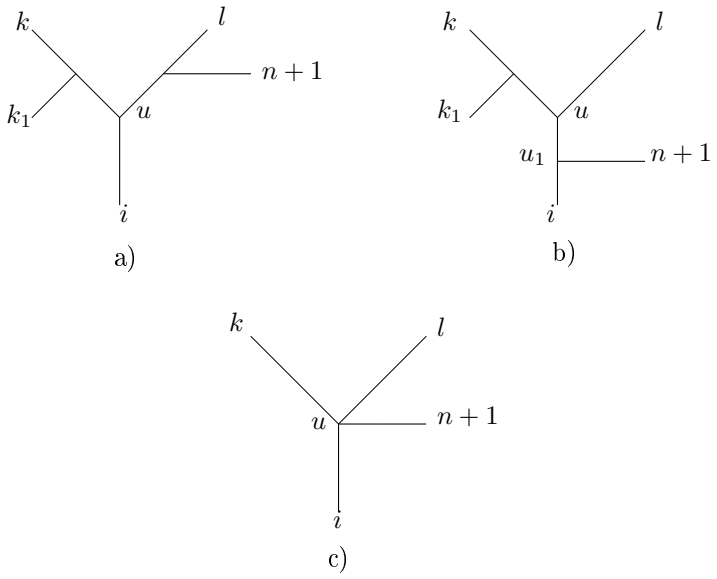


Figure 21.6. Some tree topologies for proving Theorem 21.7.

Due to the additive metric property and the place of leaf $n+1$

$$d_{k,n+1} + d_{i,l} = d_{i,n+1} + d_{k,l} \; . \tag{21.75}$$

Using inequalities $d_{i,l} = d_{i,u} + d_{u,l}$ és a $d_{k,l} = d_{k,u} + d_{u,l}$, it follows that

$$d_{k,n+1} - d_{k,u} = d_{i,n+1} - d_{i,u}. \tag{21.76}$$

Similarly for all leaves k_1 that are not separated from k by u, it holds that

$$d_{k_1,n+1} + d_{i,l} = d_{i,n+1} + d_{k_1,l} \tag{21.77}$$

It is due to the additive metric and the inequality

$$d_{k,k_1} + d_{l,n+1} < d_{k,n+1} + d_{k_1,l} \tag{21.78}$$

this later inequality comes from these inequalities

$$d_{k,k_1} + d_{i,l} < d_{k_1,l} + d_{k,i} \tag{21.79}$$

$$d_{l,n+1} + d_{k,i} < d_{i,l} + d_{k,n+1} \tag{21.80}$$

If the degree of u is greater than 3, then similar inequalities hold.

Due to the way of calculating the new edge lengths, $d_{i,n+1}$ is represented properly on the new tree, hence $d_{j,n+1}$ is represented properly for all j that is separated from leaf $n+1$ by i. Note that i might be an earlier u.

If leaf $n+1$ is connected to the edge between i and u (Fig. 21.6/b), then due to

the definition $d_{u,n+1}$, $d_{l,n+1}$ is represented properly. >From the equation

$$d_{k,n+1} + d_{i,l} = d_{k,i} + d_{l,n+1} \tag{21.81}$$

it follows that

$$d_{k,n+1} = d_{k,u} + d_{u,n+1} \ , \tag{21.82}$$

hence $d_{k,n+1}$ is represented properly. It can be similarly shown that for all points j that are separated from $n+1$ by u, the $d_{j,n+1}$ is represented properly on the tree.

If leaf $n+1$ is connected to node u (Fig. 21.6/c), then from the equations

$$d_{i,n+1} + d_{k,l} = d_{k,i} + d_{l,n+1} = d_{k,n+1} + d_{j,i} \tag{21.83}$$

it comes that both $d_{k,n+1}$ and $d_{l,n+1}$ are represents properly on the new tree, and with similar reasoning, it is easy to show that actually for all nodes j that is separated from $n+1$ by u, $d_{j,n+1}$ is represented properly on the tree.

Hence we construct a tree containing leaf $n+1$ from the tree containing the first n leaves, thus proving Theorem 21.7. ■

It is easy to show that the above algorithm that constructs the tree representing an additive metric takes $O(n^2)$ running time. However, it works only if the input distances follow an additive metric, other wise inequality 21.74 might hold several times, hence we cannot decide where to join leaf $n+1$ to. We shell introduce an algorithm that has $\Theta(n^3)$ running time and gives back the additive tree whenever the input distances follow an additive metric, moreover it generates an additive tree that approximates the input distances if those are not follow an additive metric.

The **Neighbour-Joining** algorithm works in the following way: Given a set with n points and a distance function d on the points. First we calculate the for each point i the sum of the distances from the other points:

$$v_i = \sum_{j=1}^{n} d_{i,j} \ . \tag{21.84}$$

Then we find the pair of points for which

$$s_{i,j} = (n-2)d_{i,j} - v_i - v_j \tag{21.85}$$

is minimal. The length of the edges from points i and j to the new point u are

$$e_{i,u} = \frac{d_{i,j}}{2} - \frac{v_i - v_j}{2n-4} \tag{21.86}$$

and

$$e_{j,u} = \frac{d_{i,j}}{2} - e_{i,u} \tag{21.87}$$

Then we recalculate distances. We drop points i and j, and add point u. The distance between u and any other point k is defined as

$$d_{k,u} = \frac{d_{k,i} + d_{k,j} - d_{i,j}}{2} \ . \tag{21.88}$$

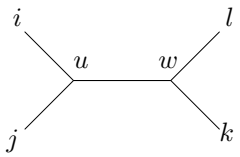


Figure 21.7. The configuration of nodes i, j, k and l if i and j follows a cherry motif.

Theorem 21.8 *If d follows an additive metric, then the* Neighbour Joining *algorithm generates a tree that gives back d.*

Proof From Theorem 21.7 there is exactly one tree that represents the distances. It is enough to prove that the Neighbour-Joining algorithm always pick a cherry motif on this tree, since a straightforward calculation shows that in this case the calculated edge lengths are proper.

First we prove if i and j follows a cherry motif then for all k, $s_{i,j} < s_{i,k}$ és $s_{i,j} < s_{k,j}$. Indeed, rearranging s, we have to prove that

$$\sum_{l \neq i,j} (d_{i,j} - d_{i,l} - d_{j,l}) - 2d_{i,j} - \sum_{m \neq j,k} (d_{j,k} - d_{j,m} - d_{k,m}) + 2d_{j,k} < 0 \qquad (21.89)$$

If $l = m \neq i, j, k$, then we get that

$$(d_{i,j} - d_{i,l} - d_{j,l}) - d_{j,k} + d_{j,l} + d_{k,l} = 2d_{w,l} - 2d_{u,l} < 0 \,, \qquad (21.90)$$

(see also Fig. 21.7). $2d_{j,k} - 2d_{i,j}$ and the cases $l = k$ and $m = i$ inside the sums cancel each other, hence we prove that the 21.89 inequality holds.

Now we prove the Theorem 21.8 in an indirect way. Suppose that i and j does not follow a cherry motif, however, $s_{i,j}$ is minimal. From the previous lemma, neither i nor j are in a cherry motif with other leaves. We find a cherry motif with leaves k and l and internal node w. Let v denote the last common node of paths going from w to i and to j. Since $s_{i,j}$ is minimal,

$$s_{k,l} - s_{i,j} > 0 \,. \qquad (21.91)$$

Rearranging this we get that

$$\sum_{m_1 \neq k,l} (d_{k,l} - d_{m_1,k} - d_{m_1,l}) - 2d_{k,l} - \sum_{m_2 \neq i,j} (d_{i,j} - d_{m_2,i} - d_{m_2,k}) + 2d_{i,j} > 0 \,. \qquad (21.92)$$

$2d_{i,j} - 2d_{k,l}$ and cases $m_1 = k$, $m_1 = l$, $m_2 = i$ and $m_2 = j$ inside the sum cancel each other. For the other $m = m_1 = m_2 \neq i, j, k, l$, the left hand side is

$$d_{k,l} - d_{m,k} - d_{m,l} - d_{i,j} + d_{m,i} + d_{m,k} \,. \qquad (21.93)$$

If m joins to the tree via the path connecting i and j, then the expression 21.93 will be always negative, see also Fig. 21.8. Let these cases be called I. class cases. If m joins to the tree via the path between v and w, then the expression 21.93 might

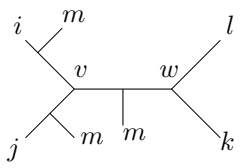


Figure 21.8. The possible places for node m on the tree.

be positive. Let these cases called II. class cases. To avoid contradiction, the sum of absolute values from I. class cases must be less than the sum from the II. class cases.

There is another v' node on the path connecting i and j, and we can find a cherry motif after node v' with leaves k' and l' and internal node w'. Here again the II. class cases have to be more than the I. class cases, but this contradict to the situation with the first cherry motif. Hence i and j form a cherry motif and we prove Theorem 21.8. ■

Exercises

21.5-1 Show that in a ultrametric, three distances coming from three points are all equal or two of them equal and the third is smaller. Prove the similar claim for the three sum of distances coming from four points in an additive metric.

21.5-2 Show that a ultrametric is always an additive metric.

21.5-3 Give an example for a metric that is not additive.

21.5-4 Is it true that all additive metric is a Euclidean metric?

21.5-5 Give the formula that calculates $d_{u,k}$ from $d_{i,j}$, $d_{i,k}$ and $d_{j,k}$ for the centroid method.

21.5-6 Give algorithms that decide in $O(n^2)$ whether or not a metric is

- additive
- ultrametric

(n is the number of points.)

21.6. Miscellaneous topics

In this section, we cover topics that are usually not mentioned in bioinformatics books. We only mention the main results in a nutshell and do not prove theorems.

21.6.1. Genome rearrangement

The genome of an organism consists of several genes. For each gene, only one strand of the double stranded DNA contains meaningful information, the other strand is the reverse complement. Since the DNA is chemically oriented, we can talk about the direction of a gene. If each gene has one copy in the genome then we can describe the order and directions of genes as a signed permutation, where the signs give the directions of genes.

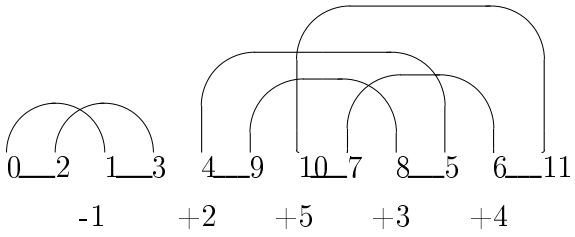


Figure 21.9. Representation of the $-1, +2, +5, +3, +4$ signed permutation with an unsigned permutation, and its graph of desire and reality.

Given two genomes with the same gene content, represented as a signed permutation then the problem is to give the minimal series of mutations that transform one genome into another. We consider three types of mutations:

- **Reversal** A reversal acts on a consecutive part of the signed permutation. It reverse the order of genes on the given part as well as the signs of the genes.
- **Transposition** A transposition swaps two consecutive block of genes.
- **Reverted transposition** It swaps two consecutive blocks and one of the blocks is reverted. As for reversals, the signs in the reverted block also change.

If we assume that only mutations happened, then we can give an $O(n^2)$ running time algorithm that obtains a shortest series of mutations transforming one genome into another, where n is the number of genes.

If we consider other types of mutations, then the complexity of problems is unknown. For transpositions, the best approximation is an 1.375 approximation [96], if we consider all possible types of mutations, then the best approximation is a 2-approximation [145]. For a wide range of and biologically meaningful weights, the weighted sorting problem for all types of mutations has a 1.5-approximation [27].

If we do not know the signs, then the problem is proved to be NP-complete [57]. Similarly, the optimal reversal median problem even for three genomes and signed permutations is NP-complete [56]. The optimal reversal median is a genome that minimises the sum of distances from a set of genomes.

Below we describe the Hannenhalli-Pevzner theorem for the reversal distance of two genomes. Instead of transforming permutation π_1 into π_2, we transform $\pi_2^{-1}\pi_1$ into the identical permutation. Based on elementary group theory, it is easy to show that the two problems are equivalent. We assume that we already calculated $\pi_2^{-1}\pi_1$, and we will denote it simply by π.

We transform an n long signed permutation into a $2n$ long unsigned permutation by replacing $+i$ to $2i-1, 2i$ and $-i$ to $2i, 2i-1$. Additionally, we frame the unsigned permutation into 0 and $2n+1$. The vertexes of the so-called graph of desire and reality are the numbers of the unsigned permutation together with 0 and $2n+1$. Starting with 0, we connect every other number in the graph, these are the reality edges. Starting also with 0, we connect $2i$ with $2i+1$ with an arc, these are the desire edges. An example graph can be seen on Fig. 21.9. Since each vertex in the graph of

desire and reality has a degree of two, the graph can be unequivocally decomposed into cycles. We call a cycle a directed cycle if on a walk on the cycle, we go at least once from left to right on a reality cycle and we go at least once from right to left on a reality cycle. Other cycles are unoriented cycles.

The span of a desire edge is the interval between its left and right vertexes. Two cycles overlap if there are two reality edges in the two cycles whose spans intersect. The vertexes of the overlap graph of a signed permutation are the cycles in its graph of desire and reality, two nodes are connected if the two cycles overlap. The overlap graph can be decomposed into components. A component is directed if it contains a directed cycle, otherwise it is unoriented. The span of a component is the interval between its leftmost and rightmost nodes in the graph of desire and reality. An unoriented component is a hurdle if its span does not contain any unoriented component or it contains all unoriented component. Other components are called protected non-hurdles.

A super-hurdle is hurdle for which it is true that if we delete this hurdle then one of the protected non-hurdles becomes a hurdle. A fortress is a permutation in which all hurdles are super-hurdles and their number is odd.

The Hannenhalli-Pevzner theorem is the following:

Theorem 21.9 *Given a signed permutation π. The minimum number of mutations sorting this permutation into the identical permutation is*

$$n + 1 - c_\pi + h_\pi + f_\pi \tag{21.94}$$

where n is the length of the permutation, c_π is the number of cycles, h_π is the number of hurdles, and $f_\pi = 1$ if the permutation is a fortress, otherwise 0

The proof of the theorem can be found in book [271].

The reversal distance can be calculated in $\Theta(n)$ time [26]. It is very easy to obtain c_π in $\Theta(n)$ time. The hard part is to calculate h_π and f_π. The source of the problem is that the overlap graph might contain $\Omega(n^2)$ edges. Therefore the fast algorithm does not obtain the entire overlap graph, only a spanning tree on each component of it.

21.6.2. Shotgun sequencing

A genome of an organism usually contain significantly more than one million nucleic acids. Using a special biochemical technology, the order of nucleic acids can be obtained, however, the uncertainty grows with the length of the DNA, and becomes absolutely unreliable after about 500 nucleic acids.

A possible solution for overcoming this problem is the following: several copies are made from the original DNA sequence and they are fragmented into small parts that can be sequenced in the above described way. Then the original sequence must be reconstructed from its overlapping fragments. This technique is called ***shotgun sequencing*** .

The mathematical definition of the problem is that we want to find the shortest common super-sequence of a set of sequences. Sequence B is a super-sequence of A if A is subsequence of B. Recall that a subsequence is not necessarily a consecutive part

of the sequence. Maier proved that the shortest common super-sequence problem is NP-complete is the size of the alphabet is at least 5 and conjectured that it is the case if the size is at least 3. Later on it has been proved that the problem is NP-complete for all non-trivial alphabet [281].

Similar problem is the shortest common super-string problem, that is also an NP-complete problem [125]. This later has biological relevance, since we are looking for overlapping substrings. Several approximation algorithms have been published for the shortest common super-string problem. A greedy algorithm finds for each pair of strings the maximal possible overlap, then it tries to find a shortest common super-string by merging the overlapping strings in a greedy way [318]. The running time of the algorithm is $O(Nm)$, where N is the number of sequences and m is the total length of the sequences. This greedy method is proved to be a 4-approximation [47]. A modified version being a 3-approximation also exist, and the conjecture is that the modified version is a 2-approximation [47].

The sequencing of DNA is not perfect, insertions, deletions and substitutions might happen during sequencing. Therefore Jiang and Li suggested the shortest k-approximative common super-string problem [180]. Kececioglu and Myers worked out a software package including several heuristic algorithm for the problem [193]. Later on Myers worked for Celera, which played a very important role in sequencing the human genome. A review paper on the topic can be found in [341].

Exercises

21.6-1 Show that a fortress contains at least three super-hurdle.

21.6-2 At least how long is a fortress?

Problems

21-1 Concave Smith–Waterman

G ive the Smith–Waterman algorithm for concave gap penalty.

21-2 Concave Spouge

G ive Spouge algorithm for concave gap penalty.

21-3 Serving at a petrol station

T here are two rows at a petrol station. Each car needs either petrol or diesel oil. At most two cars can be served at the same time, but only if they need different type of fuel, and the two cars are the first ones in the two rows or the first two in the same row. The serving time is the same not depending on whether two cars are being served or only one. Give a pair-HMM for which the Viterbi algorithm provides a shortest serving scenario.

21-4 Moments of an HMM

G iven an HMM and a sequence. Obtain the mean, variance, kth moment of the probabilities of paths emitting the given sequence.

21-5 Moments of a SCFG

G iven a SCFG and a sequence. Obtain the mean, variance, kth moment of the probabilities of derivations of the given sequence.

21-6 Co-emission probability of two HMMs
C an this probability be calculated in $O((n_1 n_2)^2)$ time where n_1 and n_2 are the number of states in the HMMs?

21-7 Sorting reversals
A sorting reversal is a reversal that decreases the reversal distance of a signed permutation. How can a sorting reversal change the number of cycles and hurdles?

Chapter notes

The first dynamic programming algorithm for aligning biological sequences was given by Needleman and Wunch in 1970 [251]. Though the concave gap penalty function is biologically more relevant, the affine gap penalty has been the standard soring scheme for aligning biological sequences. For example, one of the most popular multiple alignment program, CLUSTAL-W uses affine gap penalty and iterative sequence alignment [322]. The edit distance of two strings can calculated faster than $\Theta(l^2)$ time, that is the famous "Four Russians' speedup" [19]. The running time of the algorithm is $O(n^2/\log(n))$, however, it has such a big constant in the running time that it is not worth using it for sequence lengths appear in biological applications. The longest common subsequence problem can be solved using a dynamic programming algorithm similar to the dynamic programming algorithm for aligning sequences. Unlike that algorithm, the algorithm of Hunt and Szymanski creates a graph whose points are the characters of the sequences A and B, and a_i is connected to b_j iff $a_i = b_j$. Using this graph, the longest common subsequence can be find in $\Theta((r+n)\log(n))$ time, where r is the number of edges in the graph and n is the number of nodes [170]. Although the running time of this algorithm is $O(n^2 \log(n))$, since the number of edges might be $O(n^2)$, in many cases the number of edges is only $O(n)$, and in this cases the running time is only $O(n \log(n))$. A very sophisticated version of the corner-cutting method is the diagonal extension technique, which fills in the dynamic programming table by diagonals and does not need a test value. An example for such an algorithm is the algorithm of Wu at al. [347]. the `diff` command in the Unix operating system is also based on diagonal extension [244], having a running time $O(n+m+d_e^2)$, where n and m are the lengths of the sequences and d_e is the edit distance between the two sequences. The Knuth-Morris-Pratt string-searching algorithm searches a small pattern P in a long string M. Its running time is $\Theta(p+m)$, where p and m are the length of the sequences [201]. Landau and Vishkin modified this algorithm such that the modified version can find a pattern in M that differs at most in k position [213]. The running time of the algorithm is $\Theta(k(p\log(p)+m))$, the memory requirement is $\Theta(k(p+m))$. Although dynamic programming algorithms are the most frequently used techniques for aligning sequences, it is also possible to attack the problem with integer linear programming. Kececioglu and his colleges gave the first linear programming algorithm for aligning sequences [194]. Their method has been extended to arbitrary gap penalty functions [9]. Lancia wrote a review paper on the topic [212] and Pachter and Sturmfels showed the relationship between the dynamic programming and integer linear programming approach in their book *Algebraic Statistics for Computational Biology* [262]. The structural alignment consi-

ders only the 3D structure of sequences. The optimal structural alignment problem is to find an alignment where we penalise gaps, however, the aligned characters scored not by their similarity but by how close their are in the superposed 3D structures. Several algorithms have been developed for the problem, one of them is the combinatorial extension (CE) algorithm [301]. For a given topology it is possible to find the Maximum Likelihood labeling [278]. This algorithm has been integrated into PAML, which is one of the most popular software package for phylogenetic analysis (http://abacus.gene.ucl.ac.uk/software/paml.html). The Maximum Likelihood tree problem is to find for a substitution model and a set of sequences the tree topology and edge lengths for which the likelihood is maximal. Surprisingly, it has only recently been proved that the problem is NP-complete [64, 287]. The similar problem, the Ancestral Maximum Likelihood problem has been showed to be NP-complete also only recently [4]. The AML problem is to find the tree topology, edge lengths and labellings for which the likelihood of a set of sequences is maximal in a given substitution model. The two most popular sequence alignment algorithms based on HMMs are the SAM [169] and the HMMER (http://hmmer.wustl.edu/) packages. An example for HMM for genome annotation is the work of Pedersen and Hein [268]. Comparative genome annotation can be done with pair-HMMs like the DoubleScan [241], (http://www.sanger.ac.uk/Software/analysis/doublescan/) and the Projector [242], (http://www.sanger.ac.uk/Software/analysis/projector/) programs. Goldman, Thorne and Jones were the first who published an HMM in which the emission probabilities are calculated from evolutionary informations [134]. It was used for protein secondary structure prediction. The HMM emits alignment columns, the emission probabilities can be calculated with the Felsenstein algorithm. The Knudsen-Hein grammar is used in the PFold program, which is for predicting RNA secondary structures [200]. This SCFG generates RNA multiple alignments, where the terminal symbols are alignment columns. The derivation probabilities can be calculated with the Felsenstein algorithm, the corresponding substitution model is a single nucleotide or a dinucleotide model, according to the derivation rules. The running time of the FORWARD algorithm grows squarely with the number of states in the HMM. However, this is not always the fastest algorithm. For a biologically important HMM, it is possible to reduce the $\Theta(5^n L^n)$ running time of the FORWARD algorithm to $\Theta(2^n L^n)$ with a more sophisticated algorithm[226], [225]. However, it is unknown whether or not similar acceleration exist for the VITERBI algorithm. The Zuker-Tinoco model [323] defines free energies for RNA secondary structure elements, and the free energy of an RNA structure is the sum of free energies of the elements. The Zuker-Sankoff algorithm calculates in $\Theta(l^4)$ time the minimum free energy structure, using $\Theta(l^2)$ memory, where l is the length of the RNA sequence. It is also possible to calculate the partition function of the Boltzmann distribution with the same running time and memory requirement [239]. For a special case of free energies, both the optimal structure and the partition function can be calculated in $\Theta(l^3)$ time, using still only $\Theta(l^2)$ memory [230]. Two base-pairings, $i \cdot j$ and $i' \cdot j'$ forms a pseudo-knot if $i < i' < j < j'$. Predicting the optimal RNA secondary structure in which arbitrary pseudo-knots are allowed is NP-complete [229]. For special types of pseudo-knots, polynomial running time algorithms exist [8, 229, 285, 329]. RNA secondary structures can be compared with aligning ordered forests [158].

Atteson gave a mathematical definition for the goodness of tree-constructing methods, and showed that the NEIGHBOR-JOINING algorithm is the best one for some definitions [22]. Elias and Lagergren recently published an improved algorithm for NEIGHBOR-JOINING that has only $O(n^2)$ running time [97]. There are three possible tree topologies for four species that are called quartets. If we know all the quartets of the tree, it is possible to reconstruct it. It is proved that it is enough to know only the short quartets of a tree that are the quartets of closely related species [98]. A genome might contain more than one DNA sequences, the DNA sequences are called chromosomes. A genome rearrangement might happen between chromosomes, too, such mutations are called translocations. Hannenhalli gave a $\Theta(n^3)$ running time algorithm for calculating the translocation and reversal distance [156]. Pisanti and Sagot generalised the problem and gave results for the translocation diameter [273]. The generalisation of sorting permutations is the problem of finding the minimum length generating word for an element of a group. The problem is known to be NP-complete [179]. Above the reversal distance and translocation distance problem, only for the block interchange distance exists a polynomial running time algorithm [65]. We mention that Bill Gates, the owner of Microsoft worked also on sorting permutations, actually, with prefix reversals [129]. We wrote only about the most important topics of bioinformatics, and we did not cover several topics like recombination, pedigree analysis, character-based tree reconstructing methods, partial digesting, protein threading methods, DNA chip analysis, knowledge representation, biochemical pathways, scale-free networks, etc. We close the chapter with the words of Donald Knuth: "It is hard for me to say confidently that, after fifty more years of explosive growth of computer science, there will still be a lot of fascinating unsolved problems at peoples' fingertips, that it won't be pretty much working on refinements of well-explored things. Maybe all of the simple stuff and the really great stuff has been discovered. It may not be true, but I can't predict an unending growth. I can't be as confident about computer science as I can about biology. Biology easily has 500 years of exciting problems to work on, it's at that level."

22. Computer Graphics

Computer Graphics algorithms create and render ***virtual worlds*** stored in the computer memory. The virtual world model may contain ***shapes*** (points, line segments, surfaces, solid objects etc.), which are represented by digital numbers. ***Rendering*** computes the displayed ***image*** of the virtual world from a given virtual camera. The image consists of small rectangles, called ***pixels***. A pixel has a unique colour, thus it is sufficient to solve the rendering problem for a single point in each pixel. This point is usually the centre of the pixel. Rendering finds that shape which is visible through this point and writes its visible colour into the pixel. In this chapter we discuss the creation of virtual worlds and the determination of the visible shapes.

22.1. Fundamentals of analytic geometry

The base set of our examination is the Euclidean ***space***. In computer algorithms the elements of this space should be described by numbers. The branch of geometry describing the elements of space by numbers is the ***analytic geometry***. The basic concepts of analytic geometry are the vector and the coordinate system.

Definition 22.1 *A **vector** is a **translation** that is defined by its direction and length. A vector is denoted by $\vec{v}$.*

The length of the vector is also called its ***absolute value***, and is denoted by $|\vec{v}|$. Vectors can be added, resulting in a new vector that corresponds to subsequent translations. Addition is denoted by $\vec{v}_1 + \vec{v}_2 = \vec{v}$. Vectors can be multiplied by scalar values, resulting also in a vector ($\lambda \cdot \vec{v}_1 = \vec{v}$), which translates at the same direction as $\vec{v}_1$, but the length of translation is scaled by λ.

The ***dot product*** of two vectors is a *scalar* that is equal to the product of the lengths of the two vectors and the cosine of their angle:

$$\vec{v}_1 \cdot \vec{v}_2 = |\vec{v}_1| \cdot |\vec{v}_2| \cdot \cos\alpha, \qquad \text{where } \alpha \text{ is the angle between } \vec{v}_1 \text{ and } \vec{v}_2 \ .$$

Two vectors are said to be ***orthogonal*** if their dot product is zero.

On the other hand, the ***cross product*** of two vectors is a *vector* that is orthogonal to the plane of the two vectors and its length is equal to the product of the

lengths of the two vectors and the sine of their angle:

$$\vec{v}_1 \times \vec{v}_2 = \vec{v}, \qquad \text{where } \vec{v} \text{ is orthogonal to } \vec{v}_1 \text{ and } \vec{v}_2, \text{ and } |\vec{v}| = |\vec{v}_1| \cdot |\vec{v}_2| \cdot \sin\alpha \ .$$

There are two possible orthogonal vectors, from which that alternative is selected where our middle finger of the right hand would point if our thumb were pointing to the first and our forefinger to the second vector (***right hand rule***). Two vectors are said to be ***parallel*** if their cross product is zero.

22.1.1. Cartesian coordinate system

Any vector $\vec{v}$ of a plane can be expressed as the linear combination of two, non-parallel vectors $\vec{i}$, $\vec{j}$ in this plane, that is

$$\vec{v} = x \cdot \vec{i} + y \cdot \vec{j} \ .$$

Similarly, any vector $\vec{v}$ in the three-dimensional space can be unambiguously defined by the linear combination of three, not coplanar vectors:

$$\vec{v} = x \cdot \vec{i} + y \cdot \vec{j} + z \cdot \vec{k} \ .$$

Vectors $\vec{i}$, $\vec{j}$, $\vec{k}$ are called ***basis vectors***, while scalars x, y, z are referred to as ***coordinates***. We shall assume that the basis vectors have unit length and they are orthogonal to each other. Having defined the basis vectors any other vector can unambiguously be expressed by three scalars, i.e. by its coordinates.

A ***point*** is specified by that vector which translates the reference point, called ***origin***, to the given point. In this case the translating vector is the ***place vector*** of the given point.

The origin and the basis vectors constitute the ***Cartesian coordinate system***, which is the basic tool to describe the points of the Euclidean plane or space by numbers.

The Cartesian coordinate system is the algebraic basis of the Euclidean geometry, which means that scalar triplets of Cartesian coordinates can be paired with the points of the space, and having made a correspondence between algebraic and geometric concepts, the theorems of the Euclidean geometry can be proven by algebraic means.

Exercises

22.1-1 Prove that there is a one-to-one mapping between Cartesian coordinate triplets and points of the three-dimensional space.

22.1-2 Prove that if the basis vectors have unit length and are orthogonal to each other, then $(x_1, y_1, z_1) \cdot (x_2, y_2, z_2) = x_1x_2 + y_1y_2 + z_1z_2$.

22.2. Description of point sets with equations

Coordinate systems provide means to specify points by numbers. Conditions on these numbers, on the other hand, may define sets of points. Conditions are formulated by

solid	$f(x,y,z)$ implicit function
sphere of radius R	$R^2 - x^2 - y^2 - z^2$
block of size $2a, 2b, 2c$	$\min\{a - \lvert x\rvert, b - \lvert y\rvert, c - \lvert z\rvert\}$
torus of axis z, radii r (tube) and R (hole)	$r^2 - z^2 - (R - \sqrt{x^2 + y^2})^2$

Figure 22.1. Functions defining the sphere, the block, and the torus.

equations. The coordinates found as the solution of these equations define the point set.

Let us now consider how these equations can be established.

22.2.1. Solids

A ***solid*** is a subset of the three-dimensional Euclidean space. To define this subset, continuous function f is used which maps the coordinates of points onto the set of real numbers. We say that a point belongs to the solid if the coordinates of the point satisfy the following implicit inequality:

$$f(x,y,z) \geq 0 \ .$$

Points satisfying inequality $f(x,y,z) > 0$ are the ***internal points,*** while points defined by $f(x,y,z) < 0$ are the ***external points.*** Because of the continuity of function f, points satisfying equality $f(x,y,z) = 0$ are between external and internal points and are called the ***boundary surface*** of the solid. Intuitively, function f describes the signed distance between a point and the boundary surface.

We note that we usually do not consider any point set as a solid, but also require that the point set does not have lower dimensional degeneration (e.g. hanging lines or surfaces), i.e. that arbitrarily small neighborhoods of each point of the boundary surface contain internal points.

Figure 22.1 lists the defining functions of the sphere, the box, and the torus.

22.2.2. Surfaces

Points having coordinates that satisfy equation $f(x,y,z) = 0$ are on the boundary ***surface***. Surfaces can thus be defined by this ***implicit equation.*** Since points can also be given by the place vectors, the implicit equation can be formulated for the place vectors as well:

$$f(\vec{r}) = 0 \ .$$

A surface may have many different equations. For example, equations $f(x,y,z) = 0$, $f^2(x,y,z) = 0$, and $2 \cdot f^3(x,y,z) = 0$ are algebraically different, but they have the same roots and thus define the same set of points.

A plane of normal vector $\vec{n}$ and place vector $\vec{r}_0$ contains those points for which vector $\vec{r} - \vec{r}_0$ is perpendicular to the normal, thus their dot product is zero. Based on this, the points of a plane are defined by the following vector or scalar equations:

$$(\vec{r} - \vec{r}_0) \cdot \vec{n} = 0, \qquad n_x \cdot x + n_y \cdot y + n_z \cdot z + d = 0 \ , \tag{22.1}$$

solid	$x(u,v)$	$y(u,v)$	$z(u,v)$
sphere of radius R	$R \cdot \cos 2\pi u \cdot \sin \pi v$	$R \cdot \sin 2\pi u \cdot \sin \pi v$	$R \cdot \cos \pi v$
cylinder of radius R, axis z, and of height h	$R \cdot \cos 2\pi u$	$R \cdot \sin 2\pi u$	$h \cdot v$
cone of radius R, axis z, and of height h	$R \cdot (1-v) \cdot \cos 2\pi u$	$R \cdot (1-v) \cdot \sin 2\pi u$	$h \cdot v$

Figure 22.2. Parametric forms of the sphere, the cylinder, and the cone, where $u, v \in [0,1]$.

where n_x, n_y, n_z are the coordinates of the normal and $d = -\vec{r}_0 \cdot \vec{n}$. If the normal vector has unit length, then d expresses the signed distance between the plane and the origin of the coordinate system. Two planes are said to be ***parallel*** if their normals are parallel.

In addition to using implicit equations, surfaces can also be defined by ***parametric forms***. In this case, the Cartesian coordinates of surface points are functions of two independent variables. Denoting these free parameters by u and v, the parametric equations of the surface are:

$$x = x(u,v), \qquad y = y(u,v), \qquad z = z(u,v), \qquad u \in [u_{\min}, u_{\max}], \quad v \in [v_{\min}, v_{\max}] \ .$$

The implicit equation of a surface can be obtained from the parametric equations by eliminating free parameters u, v. Figure 22.2 includes the parametric forms of the sphere, the cylinder and the cone.

Parametric forms can also be defined directly for the place vectors:

$$\vec{r} = \vec{r}(u,v) \ .$$

Points of a ***triangle*** are the ***convex combinations*** of points $\vec{p}_1, \vec{p}_2$, and $\vec{p}_3$, that is

$$\vec{r}(\alpha,\beta,\gamma) = \alpha \cdot \vec{p}_1 + \beta \cdot \vec{p}_2 + \gamma \cdot \vec{p}_3, \qquad \text{where} \quad \alpha,\beta,\gamma \geq 0 \quad \text{and} \quad \alpha+\beta+\gamma = 1 \ .$$

From this definition we can obtain the usual two-variate parametric form of a triangle substituting α by u, β by v, and γ by $(1-u-v)$:

$$\vec{r}(u,v) = u \cdot \vec{p}_1 + v \cdot \vec{p}_2 + (1-u-v) \cdot \vec{p}_3, \qquad \text{where} \quad u,v \geq 0 \quad \text{and} \quad u+v \leq 1 \ .$$

22.2.3. Curves

By intersecting two surfaces, we obtain a ***curve*** that may be defined formally by the implicit equations of the two intersecting surfaces

$$f_1(x,y,z) = f_2(x,y,z) = 0 \ ,$$

but this is needlessly complicated. Instead, let us consider the parametric forms of the two surfaces, given as $\vec{r}_1(u_1,v_1)$ and $\vec{r}_2(u_2,v_2)$, respectively. The points of the intersection satisfy vector equation $\vec{r}_1(u_1,v_1) = \vec{r}_2(u_2,v_2)$, which corresponds to three scalar equations, one for each coordinate of the three-dimensional space. Thus we can eliminate three from the four unknowns (u_1,v_1,u_2,v_2), and obtain a one-variate parametric equation for the coordinates of the curve points:

test	$x(u,v)$	$y(u,v)$	$z(u,v)$
ellipse of main axes $2a$, $2b$ on plane $z=0$	$a \cdot \cos 2\pi t$	$b \cdot \sin 2\pi t$	0
helix of radius R, axis z, and elevation h	$R \cdot \cos 2\pi t$	$R \cdot \sin 2\pi t$	$h \cdot t$
line segment between points (x_1,y_1,z_1) and (x_2,y_2,z_2)	$x_1(1-t)+x_2 t$	$y_1(1-t)+y_2 t$	$z_1(1-t)+z_2 t$

Figure 22.3. Parametric forms of the ellipse, the helix, and the line segment, where $t \in [0,1]$.

$$x = x(t), \qquad y = y(t), \qquad z = z(t), \qquad t \in [t_{\min}, t_{\max}] \ .$$

Similarly, we can use the vector form:

$$\vec{r} = \vec{r}(t), \qquad t \in [t_{\min}, t_{\max}] \ .$$

Figure 22.3 includes the parametric equations of the ellipse, the helix, and the line segment.

Note that we can define curves on a surface by fixing one of free parameters u, v. For example, by fixing v the parametric form of the resulting curve is $\vec{r}_v(u) = \vec{r}(u,v)$. These curves are called ***iso-parametric curves***.

Two points define a ***line***. Let us select one point and call the place vector of this point the ***place vector of the line.*** On the other hand, the vector between the two points is the ***direction vector***. Any other point of the line can be obtained by a translation of the point of the place vector parallel to the direction vector. Denoting the place vector by $\vec{r}_0$ and the direction vector by $\vec{v}$, the equation of the ***line*** is:

$$\vec{r}(t) = r_0 + \vec{v} \cdot t, \qquad t \in (-\infty, \infty) \ . \tag{22.2}$$

Two lines are said to be ***parallel*** if their direction vectors are parallel.

Instead of the complete line, we can also specify the points of a line segment if parameter t is restricted to an interval. For example, the ***equation of the line segment*** between points $\vec{r}_1$ and $\vec{r}_2$ is:

$$\vec{r}(t) = \vec{r}_1 + (\vec{r}_2 - \vec{r}_1) \cdot t = \vec{r}_1 \cdot (1-t) + \vec{r}_2 \cdot t, \qquad t \in [0,1] \ . \tag{22.3}$$

According to this definition, the points of a line segment are the ***convex combinations*** of the endpoints.

22.2.4. Normal vectors

In computer graphics we often need the normal vectors of the surfaces (i.e. the normal vector of the tangent plane of the surface). Let us take an example. A mirror reflects light in a way that the incident direction, the normal vector, and the reflection direction are in the same plane, and the angle between the normal and the incident direction equals to the angle between the normal and the reflection direction. To carry out such and similar computations, we need methods to obtain the normal of the surface.

The equation of the tangent plane is obtained as the first order Taylor approximation of the implicit equation around point (x_0, y_0, z_0):

$$f(x,y,z) = f(x_0 + (x-x_0), y_0 + (y-y_0), z_0 + (z-z_0)) \approx$$

$$f(x_0, y_0, z_0) + \frac{\partial f}{\partial x} \cdot (x - x_0) + \frac{\partial f}{\partial y} \cdot (y - y_0) + \frac{\partial f}{\partial z} \cdot (z - z_0) \ .$$

Points (x_0, y_0, z_0) and (x, y, z) are on the surface, thus $f(x_0, y_0, z_0) = 0$ and $f(x, y, z) = 0$, resulting in the following equation of the ***tangent plane:***

$$\frac{\partial f}{\partial x} \cdot (x - x_0) + \frac{\partial f}{\partial y} \cdot (y - y_0) + \frac{\partial f}{\partial z} \cdot (z - z_0) = 0 \ .$$

Comparing this equation to equation (22.1), we can realize that the normal vector of the tangent plane is

$$\vec{n} = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) = \mathrm{grad} f \ . \tag{22.4}$$

The normal vector of parametric surfaces can be obtained by examining the iso-parametric curves. The tangent of curve $\vec{r}_v(u)$ defined by fixing parameter v is obtained by the first-order Taylor approximation:

$$\vec{r}_v(u) = \vec{r}_v(u_0 + (u - u_0)) \approx \vec{r}_v(u_0) + \frac{d\vec{r}_v}{du} \cdot (u - u_0) = \vec{r}_v(u_0) + \frac{\partial \vec{r}}{\partial u} \cdot (u - u_0) \ .$$

Comparing this approximation to equation (22.2) describing a line, we conclude that the direction vector of the tangent line is $\partial \vec{r}/\partial u$. The tangent lines of the curves running on a surface are in the tangent plane of the surface, making the normal vector perpendicular to the direction vectors of these lines. In order to find the normal vector, both the tangent line of curve $\vec{r}_v(u)$ and the tangent line of curve $\vec{r}_u(v)$ are computed, and their cross product is evaluated since the result of the cross product is perpendicular to the multiplied vectors. The normal of surface $\vec{r}(u, v)$ is then

$$\vec{n} = \frac{\partial \vec{r}}{\partial u} \times \frac{\partial \vec{r}}{\partial v} \ . \tag{22.5}$$

22.2.5. Curve modelling

Parametric and implicit equations trace back the geometric design of the virtual world to the establishment of these equations. However, these equations are often not intuitive enough, thus they cannot be used directly during design. It would not be reasonable to expect the designer working on a human face or on a car to directly specify the equations of these objects. Clearly, indirect methods are needed which require intuitive data from the designer and define these equations automatically. One category of these indirect approaches apply ***control points***. Another category of methods work with elementary building blocks (box, sphere, cone, etc.) and with set operations.

Let us discuss first how the method based on control points can define curves. Suppose that the designer specified points $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_m$, and that parametric curve of equation $\vec{r} = \vec{r}(t)$ should be found which "follows" these points. For the time being, the curve is not required to go through these control points.

We use the analogy of the centre of mass of mechanical systems to construct our curve. Assume that we have sand of unit mass, which is distributed at the control

points. If a control point has most of the sand, then the centre of mass is close to this point. Controlling the distribution of the sand as a function of parameter t to give the main influence to different control points one after the other, the centre of mass will travel through a curve running close to the control points.

Let us put weights $B_0(t), B_1(t), \ldots, B_m(t)$ at control points at parameter t. These weighting functions are also called the ***basis functions*** of the curve. Since unit weight is distributed, we require that for each t the following identity holds:

$$\sum_{i=0}^{m} B_i(t) = 1 \ .$$

For some t, the respective point of the curve curve is the centre of mass of this mechanical system:

$$\vec{r}(t) = \frac{\sum_{i=0}^{m} B_i(t) \cdot \vec{r}_i}{\sum_{i=0}^{m} B_i(t)} = \sum_{i=0}^{m} B_i(t) \cdot \vec{r}_i \ .$$

Note that the reason of distributing sand of unit mass is that this decision makes the denominator of the fraction equal to 1. To make the analogy complete, the basis functions cannot be negative since the mass is always non-negative. The centre of mass of a point system is always in the ***convex hull***[1] of the participating points, thus if the basis functions are non-negative, then the curve remains in the convex hull of the control points.

The properties of the curves are determined by the basis functions. Let us now discuss two popular basis function systems, namely the basis functions of the Bézier-curves and the B-spline curves.

Bézier-curve

Pierre Bézier, a designer working at Renault, proposed the ***Bernstein polynomials*** as basis functions. Bernstein polynomials can be obtained as the expansion of $1^m = (t + (1-t))^m$ according to binomial theorem:

$$(t + (1-t))^m = \sum_{i=0}^{m} \binom{m}{i} \cdot t^i \cdot (1-t)^{m-i} \ .$$

The basis functions of ***Bézier curves*** are the terms of this sum ($i = 0, 1, \ldots, m$):

$$B_{i,m}^{\text{Bezier}}(t) = \binom{m}{i} \cdot t^i \cdot (1-t)^{m-i} \ . \tag{22.6}$$

According to the introduction of Bernstein polynomials, it is obvious that they really meet condition $\sum_{i=0}^{m} B_i(t) = 1$ and $B_i(t) \geq 0$ in $t \in [0,1]$, which guarantees that Bézier curves are always in the convex hulls of their control points. The basis functions and the shape of the Bézier curve are shown in Figure 22.4. At parameter value $t = 0$ the first basis function is 1, while the others are zero, therefore the curve

[1] The convex hull of a point system is by definition the minimal convex set containing the point system.

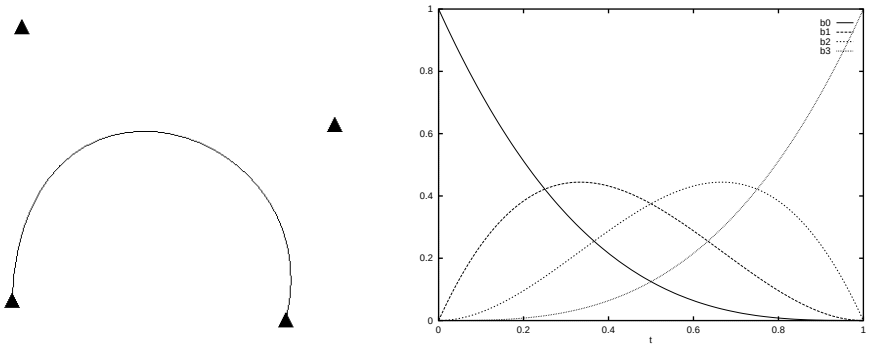


Figure 22.4. A Bézier curve defined by four control points and the respective basis functions ($m = 3$).

starts at the first control point. Similarly, at parameter value $t = 1$ the curve arrives at the last control point. At other parameter values, all basis functions are positive, thus they simultaneously affect the curve. Consequently, the curve usually does not go through the other control points.

B-spline

The basis functions of the ***B-spline*** can be constructed applying a sequence of linear blending. A B-spline weights the $m + 1$ number of control points by $(k - 1)$-degree polynomials. Value k is called the ***order*** of the curve. Let us take a non-decreasing series of $m + k + 1$ parameter values, called the ***knot vector***:

$$\mathbf{t} = [t_0, t_1, \ldots, t_{m+k}], \quad t_0 \le t_1 \le \cdots \le t_{m+k} \ .$$

By definition, the ith first order basis function is 1 in the ith interval, and zero elsewhere (Figure 22.5):

$$B_{i,1}^{\mathrm{BS}}(t) = \begin{cases} 1, & \text{if } t_i \le t < t_{i+1} \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Using this definition, $m + k$ number of first order basis functions are established, which are non-negative zero-degree polynomials that sum up to 1 for all $t \in [t_0, t_{m+k})$ parameters. These basis functions have too low degree since the centre of mass is not even a curve, but jumps from control point to control point.

The order of basis functions, as well as the smoothness of the curve, can be increased by blending two consecutive basis functions with linear weighting (Figure 22.5). The first basis function is weighted by linearly increasing factor $(t - t_i)/(t_{i+1} - t_i)$ in domain $t_i \le t < t_{i+1}$, where the basis function is non-zero. The next basis function, on the other hand, is scaled by linearly decreasing factor $(t_{i+2} - t)/(t_{i+2} - t_{i+1})$ in its domain $t_{i+1} \le t < t_{i+2}$ where it is non zero. The two weighted basis

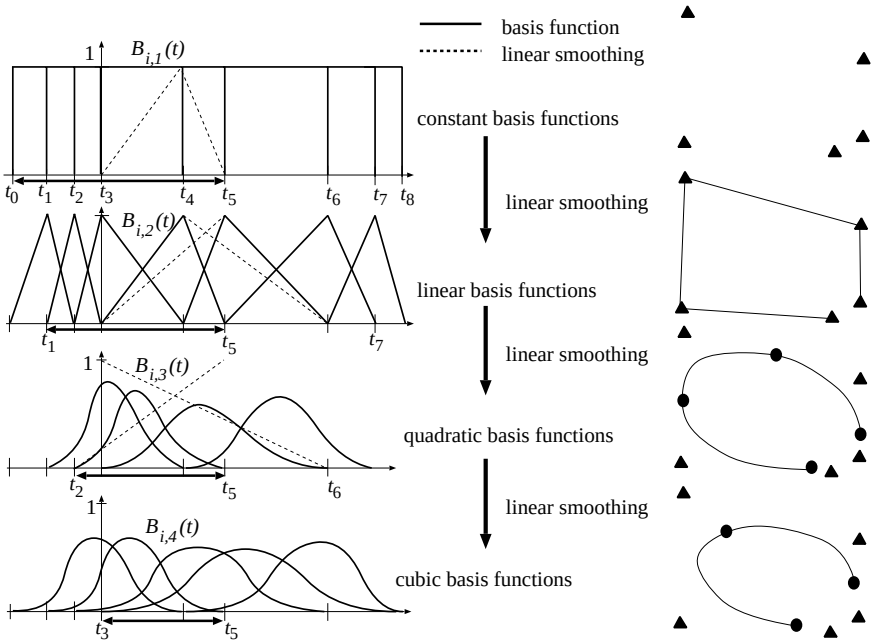


Figure 22.5. Construction of B-spline basis functions. A higher order basis function is obtained by blending two consecutive basis functions on the previous level using a linearly increasing and a linearly decreasing weighting, respectively. Here the number of control points is 5, i.e. $m = 4$. Arrows indicate useful interval $[t_{k-1}, t_{m+1}]$ where we can find $m + 1$ number of basis functions that add up to 1. The right side of the figure depicts control points with triangles and curve points corresponding to the knot values by circles.

functions are added to obtain the tent-like second order basis functions. Note that while a first order basis function is non-zero in a single interval, the second order basis functions expand to two intervals. Since the construction makes a new basis function from every pair of consecutive lower order basis functions, the number of new basis functions is one less than that of the original ones. We have just $m+k-1$ second order basis functions. Except for the first and the last first order basis functions, all of them are used once with linearly increasing and once with linearly decreasing weighting, thus with the exception of the first and the last intervals, i.e. in $[t_1, t_{m+k-1}]$, the new basis functions also sum up to 1.

The second order basis functions are first degree polynomials. The degree of basis functions, i.e. the order of the curve, can be arbitrarily increased by the recursive application of the presented blending method. The dependence of the next order basis functions on the previous order ones is as follows:

$$B_{i,k}^{\mathrm{BS}}(t) = \frac{(t - t_i)B_{i,k-1}^{\mathrm{BS}}(t)}{t_{i+k-1} - t_i} + \frac{(t_{i+k} - t)B_{i+1,k-1}^{\mathrm{BS}}(t)}{t_{i+k} - t_{i+1}}, \qquad \text{if } k > 1\ .$$

Note that we always take two consecutive basis functions and weight them in their non-zero domain (i.e. in the interval where they are non-zero) with linearly increasing factor $(t-t_i)/(t_{i+k-1}-t_i)$ and with linearly decreasing factor $(t_{i+k}-t)/(t_{i+k}-t_{i+1})$,

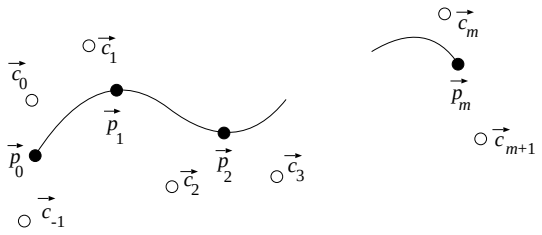


Figure 22.6. A B-spline interpolation. Based on points $\vec{p}_0, \ldots, \vec{p}_m$ to be interpolated, control points $\vec{c}_{-1}, \ldots, \vec{c}_{m+1}$ are computed to make the start and end points of the segments equal to the interpolated points.

respectively. The two weighted functions are summed to obtain the higher order, and therefore smoother basis function. Repeating this operation $(k-1)$ times, k-order basis functions are generated, which sum up to 1 in interval $[t_{k-1}, t_{m+1}]$. The knot vector may have elements that are the same, thus the length of the intervals may be zero. Such intervals result in $0/0$ like fractions, which must be replaced by value 1 in the implementation of the construction.

The value of the ith k-order basis function at parameter t can be computed with the following ***Cox-deBoor-Mansfield recursion***:

B-SPLINE$(i, k, t, \mathbf{t})$

```
1  if k = 1                                                    ▷ Trivial case.
2     then if t_i ≤ t < t_{i+1}
3             then return 1
4             else return 0
5  if t_{i+k-1} − t_i > 0
6     then b_1 ← (t − t_i)/(t_{i+k-1} − t_i)   ▷ Previous with linearly increasing weight.
7     else  b_1 ← 1                                             ▷ Here: 0/0 = 1.
8  if t_{i+k} − t_{i+1} > 0
9     then b_2 ← (t_{i+k} − t)/(t_{i+k} − t_{i+1})    ▷ Next with linearly decreasing weight.
10    else  b_2 ← 1                                             ▷ Here: 0/0 = 1.
11 B ← b_1 · B-SPLINE(i, k − 1, t, t) + b_2 · B-SPLINE(i + 1, k − 1, t, t)    ▷ Recursion.
12 return B
```

In practice, we usually use fourth-order basis functions ($k = 4$), which are third-degree polynomials, and define curves that can be continuously differentiated twice. The reason is that bent rods and motion paths following the Newton laws also have this property.

While the number of control points is greater than the order of the curve, the basis functions are non-zero only in a part of the valid parameter set. This means that a control point affects just a part of the curve. Moving this control point, the change of the curve is ***local.*** Local control is a very important property since the designer can adjust the shape of the curve without destroying its general form.

A fourth-order B-spline usually does not go through its control points. If we wish to use it for interpolation, the control points should be calculated from the points to

be interpolated. Suppose that we need a curve which visits points $\vec{p}_0, \vec{p}_1, \ldots, \vec{p}_m$ at parameter values $t_0 = 0, t_1 = 1, \ldots, t_m = m$, respectively (Figure 22.6). To find such a curve, control points $[\vec{c}_{-1}, \vec{c}_0, \vec{c}_1, \ldots, \vec{c}_{m+1}]$ should be found to meet the following interpolation criteria:

$$\vec{r}(t_j) = \sum_{i=-1}^{m+1} \vec{c}_i \cdot B_{i,4}^{\mathrm{BS}}(t_j) = \vec{p}_j, \qquad j = 0, 1, \ldots, m \ .$$

These criteria can be formalized as $m + 1$ linear equations with $m + 3$ unknowns, thus the solution is ambiguous. To make the solution unambiguous, two additional conditions should be imposed. For example, we can set the derivatives (for motion paths, the speed) at the start and end points.

B-spline curves can be further generalized by defining the influence of the ith control point as the product of B-spline basis function $B_i(t)$ and additional weight w_i of the control point. The curve obtained this way is called the Non-Uniform Rational B-Spline, abbreviated as ***NURBS***, which is very popular in commercial geometric modelling systems.

Using the mechanical analogy again, the mass put at the ith control point is $w_i B_i(t)$, thus the centre of mass is:

$$\vec{r}(t) = \frac{\sum_{i=0}^{m} w_i B_i^{\mathrm{BS}}(t) \cdot \vec{r}_i}{\sum_{j=0}^{m} w_j B_j^{\mathrm{BS}}(t)} = \sum_{i=0}^{m} B_i^{\mathrm{NURBS}}(t) \cdot \vec{r}_i \ .$$

The correspondence between B-spline and NURBS basis functions is as follows:

$$B_i^{\mathrm{NURBS}}(t) = \frac{w_i B_i^{\mathrm{BS}}(t)}{\sum_{j=0}^{m} w_j B_j^{\mathrm{BS}}(t)} \ .$$

Since B-spline basis functions are piece-wise polynomial functions, NURBS basis functions are piece-wise rational functions. NURBS can describe quadratic curves (e.g. circle, ellipse, etc.) without any approximation error.

22.2.6. Surface modelling

Parametric surfaces are defined by two variate functions $\vec{r}(u, v)$. Instead of specifying this function directly, we can take finite number of control points $\vec{r}_{ij}$ which are weighted with the basis functions to obtain the parametric function:

$$\vec{r}(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \vec{r}_{ij} \cdot B_{ij}(u, v) \ . \tag{22.7}$$

Similarly to curves, basis functions are expected to sum up to 1, i.e. $\sum_{i=0}^{n} \sum_{j=0}^{m} B_{ij}(u, v) = 1$ everywhere. If this requirement is met, we can imagine that the control points have masses $B_{ij}(u, v)$ depending on parameters u, v, and the centre of mass is the surface point corresponding to parameter pair u, v.

Basis functions $B_{ij}(u, v)$ are similar to those of curves. Let us fix parameter v. Changing parameter u, curve $\vec{r}_v(u)$ is obtained on the surface. This curve can be

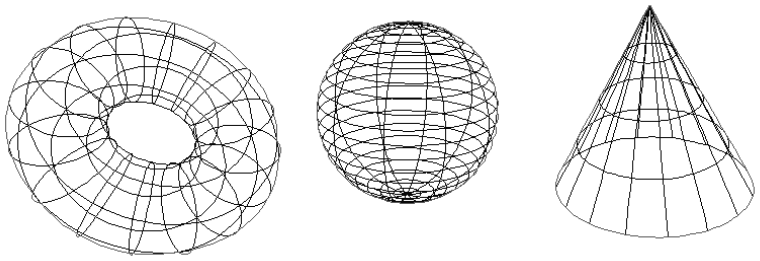


Figure 22.7. Iso-parametric curves of surface.

defined by the discussed curve definition methods:

$$\vec{r}_v(u) = \sum_{i=0}^{n} B_i(u) \cdot \vec{r}_i \ , \tag{22.8}$$

where $B_i(u)$ is the basis function of the selected curve type.

Of course, fixing v differently we obtain another curve of the surface. Since a curve of a given type is unambiguously defined by the control points, control points $\vec{r}_i$ must depend on the fixed v value. As parameter v changes, control point $\vec{r}_i = \vec{r}_i(v)$ also runs on a curve, which can be defined by control points $\vec{r}_{i,0}, \vec{r}_{i,2}, \ldots, \vec{r}_{i,m}$:

$$\vec{r}_i(v) = \sum_{j=0}^{m} B_j(v) \cdot \vec{r}_{ij} \ .$$

Substituting this into equation (22.8), the parametric equation of the surface is:

$$\vec{r}(u,v) = \vec{r}_v(u) = \sum_{i=0}^{n} B_i(u) \left(\sum_{j=0}^{m} B_j(v) \cdot \vec{r}_{ij} \right) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i(u) B_j(v) \cdot \vec{r}_{ij} \ .$$

Unlike curves, the control points of a surface form a two-dimensional grid. The two-dimensional basis functions are obtained as the product of one-variate basis functions parameterized by u and v, respectively.

22.2.7. Solid modelling with blobs

Free form solids – similarly to parametric curves and surfaces – can also be specified by finite number of control points. For each control point $\vec{r}_i$, let us assign influence function $h(R_i)$, which expresses the influence of this control point at distance $R_i = |\vec{r} - \vec{r}_i|$. By definition, the solid contains those points where the total influence of the control points is not smaller than threshold T (Figure 22.8):

$$f(\vec{r}) = \sum_{i=0}^{m} h_i(R_i) - T \geq 0, \qquad \text{where } R_i = |\vec{r} - \vec{r}_i| \ .$$

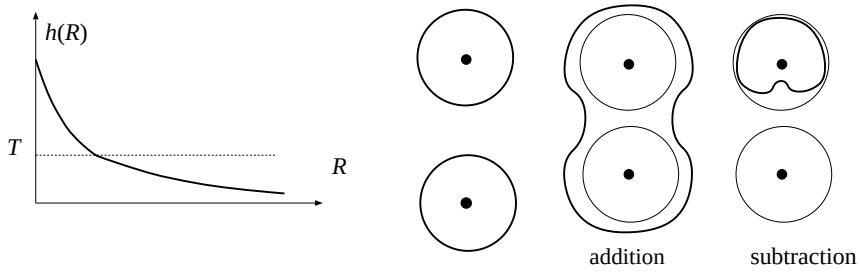


Figure 22.8. The influence decreases with the distance. Spheres of influence of similar signs increase, of different signs decrease each other.

With a single control point a sphere can be modeled. Spheres of multiple control points are combined together to result in an object having smooth surface (Figure 22.8). The influence of a single point can be defined by an arbitrary decreasing function that converges to zero at infinity. For example, Blinn proposed the

$$h_i(R) = a_i \cdot e^{-b_i R^2}$$

influence functions for his ***blob method.***

22.2.8. Constructive solid geometry

Another type of solid modelling is ***constructive solid geometry*** (***CSG*** for short), which builds complex solids from primitive solids applying set operations (e.g. union, intersection, difference, complement, etc.) (Figures 22.9 and 22.10). Primitives usually include the box, the sphere, the cone, the cylinder, the half-space, etc. whose functions are known.

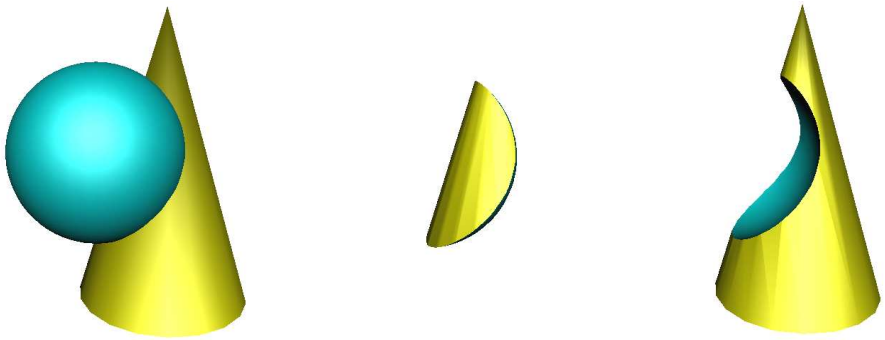


Figure 22.9. The operations of constructive solid geometry for a cone of implicit function f and for a sphere of implicit function g: union ($\max(f,g)$), intersection ($\min(f,g)$), and difference ($\min(f,-g)$).

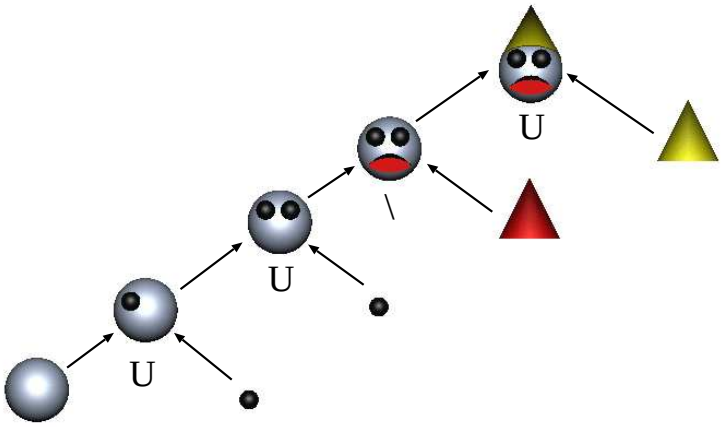


Figure 22.10. Constructing a complex solid by set operations. The root and the leaf of the ***CSG tree*** represents the complex solid, and the primitives, respectively. Other nodes define the set operations (U: union, \: difference).

The results of the set operations can be obtained from the implicit functions of the solids taking part of this operation:

- intersection of f and g: $\min(f, g)$;
- union of f and g: $\max(f, g)$.
- complement of f: $-f$.
- difference of f and g: $\min(f, -g)$.

Implicit functions also allow to ***morph*** between two solids. Suppose that two objects, for example, a box of implicit function f_1 and a sphere of implicit function f_2 need to be morphed. To define a new object, which is similar to the first object with percentage t and to the second object with percentage $(1-t)$, the two implicit equations are weighted appropriately:

$$f^{morph}(x, y, z) = t \cdot f_1(x, y, z) + (1 - t) \cdot f_2(x, y, z) \ .$$

Exercises

22.2-1 Find the parametric equation of a torus.

22.2-2 Prove that the fourth-order B-spline with knot-vector [0,0,0,0,1,1,1,1] is a Bézier curve.

22.2-3 Give the equations for the surface points and the normals of the waving flag and waving water disturbed in a single point.

22.2-4 Prove that the tangents of a Bézier curve at the start and the end are the lines connecting the first two and the last two control points, respectively.

22.2-5 Give the algebraic forms of the basis functions of the second, the third, and the fourth-order B-splines.

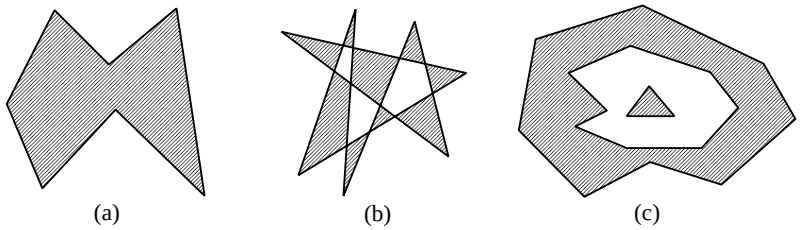


Figure 22.11. Types of polygons. **(a)** simple; **(b)** complex, single connected; **(c)** multiply connected.

22.2-6 Develop an algorithm computing the path length of a Bézier curve and a B-spline. Based on the path length computation move a point along the curve with uniform speed.

22.3. Geometry processing and tessellation algorithms

In Section 22.2 we met free-form surface and curve definition methods. During image synthesis, however, line segments and polygons play important roles. In this section we present methods that bridge the gap between these two types of representations. These methods convert geometric models to lines and polygons, or further process line and polygon models. Line segments connected to each other in a way that the end point of a line segment is the start point of the next one are called ***polylines.*** Polygons connected at edges, on the other hand, are called ***meshes. Vectorization*** methods approximate free-form curves by polylines. A polyline is defined by its vertices. ***Tessellation*** algorithms, on the other hand, approximate free-form surfaces by meshes. For illumination computation, we often need the normal vector of the original surface, which is usually stored with the vertices. Consequently, a mesh contains a list of polygons, where each polygon is given by its vertices and the normal of the original surface at these vertices. Methods processing meshes use other topology information as well, for example, which polygons share an edge or a vertex.

22.3.1. Polygon and polyhedron

Definition 22.2 *A **polygon** is a bounded part of the plane, i.e. it does not contain a line, and is bordered by line segments. A polygon is defined by the vertices of the bordering polylines.*

Definition 22.3 *A polygon is **single connected** if its border is a single closed polyline (Figure 22.11).*

Definition 22.4 *A polygon is **simple** if it is single connected and the bordering polyline does not intersect itself (Figure 22.11(a)).*

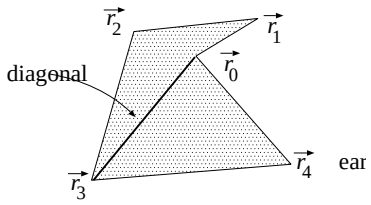


Figure 22.12. Diagonal and ear of a polygon.

For a point of the plane, we can detect whether or not this point is inside the polygon by starting a half-line from this point and counting the number of intersections with the boundary. If the number of intersections is an odd number, then the point is inside, otherwise it is outside.

In the three-dimensional space we can form meshes, where different polygons are in different planes. In this case, two polygons are said to be neighboring if they share an edge.

Definition 22.5 *A **polyhedron** is a bounded part of the space, which is bordered by polygons.*

Similarly to polygons, a point can be tested for polyhedron inclusion by casting a half line from this point and counting the number of intersections with the face polygons. If the number of intersections is odd, then the point is inside the polyhedron, otherwise it is outside.

22.3.2. Vectorization of parametric curves

Parametric functions map interval $[t_{\min}, t_{\max}]$ onto the points of the curve. During vectorization the parameter interval is discretized. The simplest discretization scheme generates $N+1$ evenly spaced parameter values $t_i = t_{\min} + (t_{\max} - t_{\min}) \cdot i/N$ $(i = 0, 1, \ldots, N)$, and defines the approximating polyline by the points obtained by substituting these parameter values into parametric equation $\vec{r}(t_i)$.

22.3.3. Tessellation of simple polygons

Let us first consider the conversion of simple polygons to triangles. This is easy if the polygon is convex since we can select an arbitrary vertex and connect it with all other vertices, which decomposes the polygon to triangles in linear time. Unfortunately, this approach does not work for concave polygons since in this case the line segment connecting two vertices may go outside the polygon, thus cannot be the edge of one decomposing triangle.

Let us start the discussion of triangle conversion algorithms with two definitions:

Definition 22.6 *The **diagonal** of a polygon is a line segment connecting two vertices and is completely contained by the polygon (line segment $\vec{r}_0$ and $\vec{r}_3$ of Figure 22.12).*

The diagonal property can be checked for a line segment connecting two vertices by

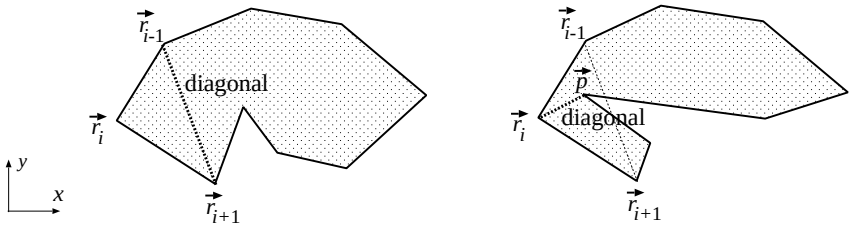


Figure 22.13. The proof of the existence of a diagonal for simple polygons.

trying to intersect the line segment with all edges and showing that intersection is possible only at the endpoints, and additionally showing that one internal point of the candidate is inside the polygon. For example, this test point can be the midpoint of the line segment.

Definition 22.7 *A vertex of the polygon is an* ***ear*** *if the line segment between the previous and the next vertices is a diagonal (vertex $\vec{r}_4$ of Figure 22.12).*

Clearly, only those vertices may be ears where the inner angle is not greater than 180 degrees. Such vertices are called ***convex vertices***.

For simple polygons the following theorems hold:

Theorem 22.8 *A simple polygon always has a diagonal.*

Proof Let the vertex standing at the left end (having the minimal x coordinate) be $\vec{r}_i$, and its two neighboring vertices be $\vec{r}_{i-1}$ and $\vec{r}_{i+1}$, respectively (Figure 22.13). Since $\vec{r}_i$ is standing at the left end, it is surely a convex vertex. If $\vec{r}_i$ is an ear, then line segment $(\vec{r}_{i-1}, \vec{r}_{i+1})$ is a diagonal (left of Figure 22.13), thus the theorem is proven for this case. Since $\vec{r}_i$ is a convex vertex, it is not an ear only if triangle $\vec{r}_{i-1}$, $\vec{r}_i$, $\vec{r}_{i+1}$ contains at least one polygon vertex (right of Figure 22.13). Let us select from the contained vertices that vertex $\vec{p}$ which is the farthest from the line defined by points $\vec{r}_{i-1}, \vec{r}_{i+1}$. Since there are no contained points which are farther from line $(\vec{r}_{i-1}, \vec{r}_{i+1})$ than $\vec{p}$, no edge can be between points $\vec{p}$ and $\vec{r}_i$, thus $(\vec{p}, \vec{r}_i)$ must be a diagonal. ■

Theorem 22.9 *A simple polygon can always be decomposed to triangles with its diagonals. If the number of vertices is n, then the number of triangles is $n-2$.*

Proof This theorem is proven by induction. The theorem is obviously true when $n = 3$, i.e. when the polygon is a triangle. Let us assume that the statement is also true for polygons having m $(m = 3, \ldots, n-1)$ number of vertices, and consider a polygon with n vertices. According to Theorem 22.8, this polygon of n vertices has a diagonal, thus we can subdivide this polygon into a polygon of n_1 vertices and a polygon of n_2 vertices, where $n_1, n_2 < n$, and $n_1 + n_2 = n + 2$ since the vertices at the ends of the diagonal participate in both polygons. According to the assumption of the induction, these two polygons can be separately decomposed to triangles. Joining the two sets of triangles, we can obtain the triangle decomposition of the

original polygon. The number of triangles is $n_1 - 2 + n_2 - 2 = n - 2$. ■

The discussed proof is constructive thus it inspires a subdivision algorithm: let us find a diagonal, subdivide the polygon along this diagonal, and continue the same operation for the two new polygons.

Unfortunately the running time of such an algorithm is in $\Theta(n^3)$ since the number of diagonal candidates is $\Theta(n^2)$, and the time needed by checking whether or not a line segment is a diagonal is in $\Theta(n)$.

We also present a better algorithm, which decomposes a convex or concave polygon defined by vertices $\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_n$. This algorithm is called ***ear cutting***. The algorithm looks for ear triangles and cuts them until the polygon gets simplified to a single triangle. The algorithm starts at vertex $\vec{r}_2$. When a vertex is processed, it is first checked whether or not the previous vertex is an ear. If it is not an ear, then the next vertex is chosen. If the previous vertex is an ear, then the current vertex together with the two previous ones form a triangle that can be cut, and the previous vertex is deleted. If after deletion the new previous vertex has index 0, then the next vertex is selected as the current vertex.

The presented algorithm keeps cutting triangles until no more ears are left. The termination of the algorithm is guaranteed by the following ***two ears theorem***:

Theorem 22.10 *A simple polygon having at least four vertices always has at least two not neighboring ears that can be cut independently.*

Proof The proof presented here has been given by Joseph O'Rourke. According to theorem 22.9, every simple polygon can be subdivided to triangles such that the edges of these triangles are either the edges or the diagonals of the polygon. Let us make a correspondence between the triangles and the nodes of a graph where two nodes are connected if and only if the two triangles corresponding to these nodes share an edge. The resulting graph is connected and cannot contain circles. Graphs of these properties are trees. The name of this tree graph is the ***dual tree.*** Since the polygon has at least four vertices, the number of nodes in this tree is at least two. Any tree containing at least two nodes has at least two leaves[2]. Leaves of this tree, on the other hand, correspond to triangles having an ear vertex. ■

According to the two ears theorem, the presented algorithm finds an ear in $O(n)$ steps. Cutting an ear the number of vertices is reduced by one, thus the algorithm terminates in $O(n^2)$ steps.

22.3.4. Tessellation of parametric surfaces

Parametric forms of surfaces map parameter rectangle $[u_{\min}, u_{\max}] \times [v_{\min}, v_{\max}]$ onto the points of the surface.

In order to tessellate the surface, first the parameter rectangle is subdivided to triangles. Then applying the parametric equations for the vertices of the parameter triangles, the approximating triangle mesh can be obtained. The simplest subdivision of the parametric rectangle decomposes the domain of parameter u to N parts, and

[2] A leaf is a node connected by exactly one edge.

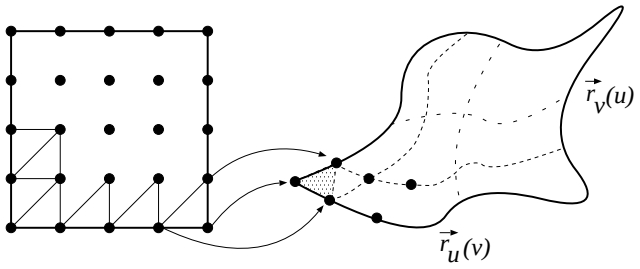


Figure 22.14. Tessellation of parametric surfaces.

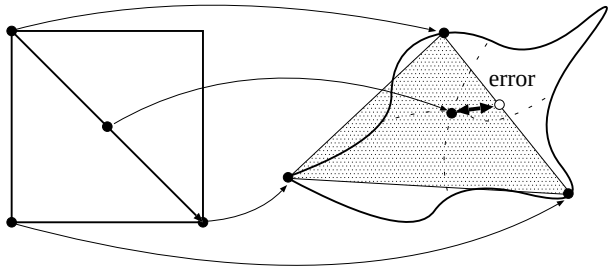


Figure 22.15. Estimation of the tessellation error.

the domain of parameter v to M intervals, resulting in the following parameter pairs:

$$[u_i, v_j] = \left[u_{\min} + (u_{\max} - u_{\min}) \frac{i}{N},\ v_{\min} + (v_{\max} - v_{\min}) \frac{j}{M} \right].$$

Taking these parameter pairs and substituting them into the parametric equations, point triplets $\vec{r}(u_i, v_j)$, $\vec{r}(u_{i+1}, v_j)$, $\vec{r}(u_i, v_{j+1})$, and point triplets $\vec{r}(u_{i+1}, v_j)$, $\vec{r}(u_{i+1}, v_{j+1})$, $\vec{r}(u_i, v_{j+1})$ are used to define triangles.

The tessellation process can be made ***adaptive*** as well, which uses small triangles only where the high curvature of the surface justifies them. Let us start with the parameter rectangle and subdivide it to two triangles. In order to check the accuracy of the resulting triangle mesh, surface points corresponding to the edge midpoints of the parameter triangles are compared to the edge midpoints of the approximating triangles. Formally the following distance is computed (Figure 22.15):

$$\left| \vec{r}\left(\frac{u_1 + u_2}{2}, \frac{v_1 + v_2}{2} \right) - \frac{\vec{r}(u_1, v_1) + \vec{r}(u_2, v_2)}{2} \right|,$$

where (u_1, v_1) and (u_2, v_2) are the parameters of the two endpoints of the edge.

A large distance value indicates that the triangle mesh poorly approximates the parametric surface, thus triangles must be subdivided further. This subdivision can

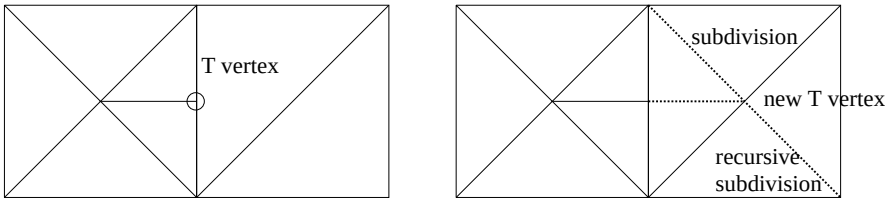


Figure 22.16. T vertices and their elimination with forced subdivision.

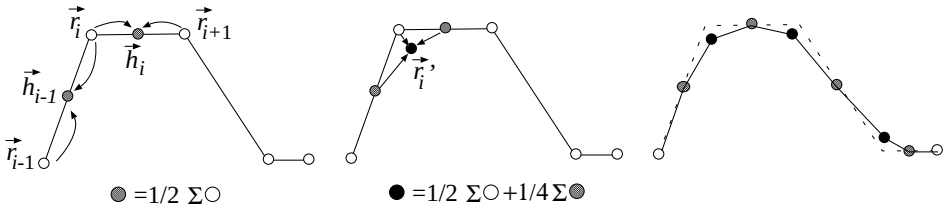


Figure 22.17. Construction of a subdivision curve: at each step midpoints are obtained, then the original vertices are moved to the weighted average of neighbouring midpoints and of the original vertex.

be executed by cutting the triangle to two triangles by a line connecting the midpoint of the edge of the largest error and the opposing vertex. Alternatively, a triangle can be subdivided to four triangles with its halving lines. The adaptive tessellation is not necessarily robust since it can happen that the distance at the midpoint is small, but at other points is still quite large.

When the adaptive tessellation is executed, it may happen that one triangle is subdivided while its neighbour is not, which results in holes. Such problematic midpoints are called ***T vertices*** (Figure 22.16).

If the subdivision criterion is based only on edge properties, then T vertices cannot show up. However, if other properties are also taken into account, then T vertices may appear. In such cases, T vertices can be eliminated by recursively forcing the subdivision also for those neighbouring triangles that share subdivided edges.

22.3.5. Subdivision curves and meshes

This section presents algorithms that smooth polyline and mesh models.

Let us consider a polyline of vertices $\vec{r}_0, \ldots, \vec{r}_m$. A smoother polyline is generated by the following vertex doubling approach (Figure 22.17). Every line segment of the polyline is halved, and midpoints $\vec{h}_0, \ldots, \vec{h}_{m-1}$ are added to the polyline as new vertices. Then the old vertices are moved taking into account their old position and the positions of the two enclosing midpoints, applying the following weighting:

$$\vec{r}_i{}' \;=\; \frac{1}{2}\vec{r}_i + \frac{1}{4}\vec{h}_{i-1} + \frac{1}{4}\vec{h}_i \;=\; \frac{3}{4}\vec{r}_i + \frac{1}{8}\vec{r}_{i-1} + \frac{1}{8}\vec{r}_{i+1}\ .$$

The new polyline looks much smoother. If we should not be satisfied with the smoothness yet, the same procedure can be repeated recursively. As can be shown, the

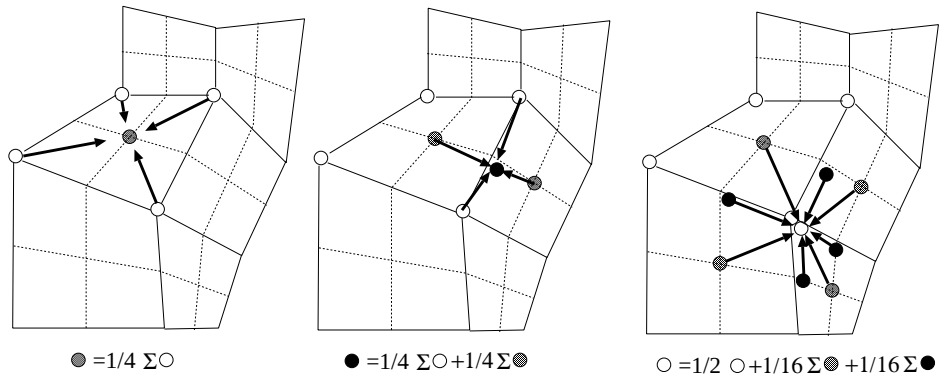


Figure 22.18. One smoothing step of the Catmull-Clark subdivision. First the face points are obtained, then the edge midpoints are moved, and finally the original vertices are refined according to the weighted sum of its neighbouring edge and face points.

result of the recursive process converges to the B-spline curve.

The polyline subdivision approach can also be extended for smoothing three-dimensional meshes. This method is called ***Catmull-Clark subdivision algorithm.*** Let us consider a three-dimensional quadrilateral mesh (Figure 22.18). In the first step the midpoints of the edges are obtained, which are called ***edge points***. Then ***face points*** are generated as the average of the vertices of each face polygon. Connecting the edge points with the face points, we still have the original surface, but now defined by four times more quadrilaterals. The smoothing step modifies first the edge points setting them to the average of the vertices at the ends of the edge and of the face points of those quads that share this edge. Then the original vertices are moved to the weighted average of the face points of those faces that share this vertex, and of edge points of those edges that are connected to this vertex. The weight of the original vertex is $1/2$, the weights of edge and face points are $1/16$. Again, this operation may be repeated until the surface looks smooth enough (Figure 22.19).

If we do not want to smooth the mesh at an edge or around a vertex, then the averaging operation ignores the vertices on the other side of the edge to be preserved.

The Catmull-Clark subdivision surface usually does not interpolate the original vertices. This drawback is eliminated by the ***butterfly subdivision***, which works on triangle meshes. First the butterfly algorithm puts new edge points close to the midpoints of the original edges, then the original triangle is replaced by four triangles defined by the original vertices and the new edge points (Figure 22.20). The position of the new edge points depend on the vertices of those two triangles incident to this edge, and on those four triangles which share edges with these two. The arrangement of the triangles affecting the edge point resembles a butterfly, hence the name of this algorithm. The edge point coordinates are obtained as a weighted sum of the edge endpoints multiplied by $1/2$, the third vertices of the triangles sharing this edge using weight $1/8 + 2w$, and finally of the other vertices of the additional triangles with weight $-1/16 - w$. Parameter w can control the curvature of the resulting mesh.

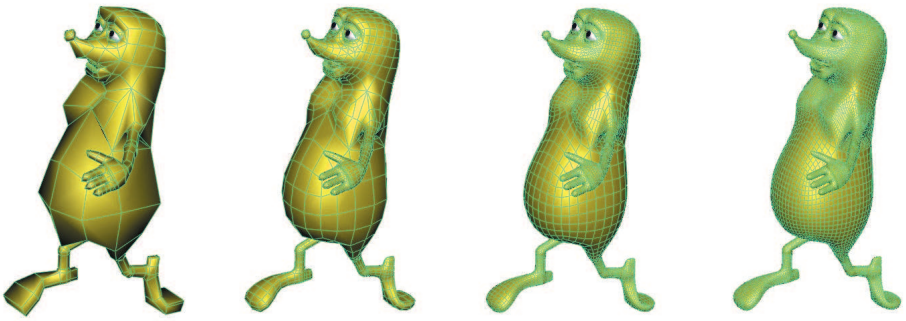


Figure 22.19. Original mesh and its subdivision applying the smoothing step once, twice and three times, respectively.

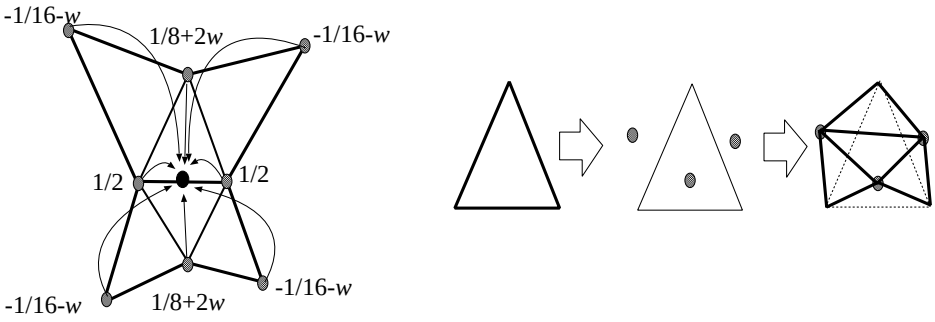


Figure 22.20. Generation of the new edge point with butterfly subdivision.

Setting $w = -1/16$, the mesh keeps its original faceted look, while $w = 0$ results in strong rounding.

22.3.6. Tessellation of implicit surfaces

A surface defined by implicit equation $f(x, y, z) = 0$ can be converted to a triangle mesh by finding points on the surface densely, i.e. generating points satisfying $f(x, y, z) \approx 0$, then assuming the close points to be vertices of the triangles.

First function f is evaluated at the grid points of the Cartesian coordinate system and the results are stored in a three-dimensional array, called ***voxel array.*** Let us call two grid points as neighbours if two of their coordinates are identical and the difference in their third coordinate is 1. The function is evaluated at the grid points and is assumed to be linear between them. The normal vectors needed for shading are obtained as the gradient of function f (equation 22.4), which are also interpolated between the grid points.

When we work with the voxel array, original function f is replaced by its ***tri-***

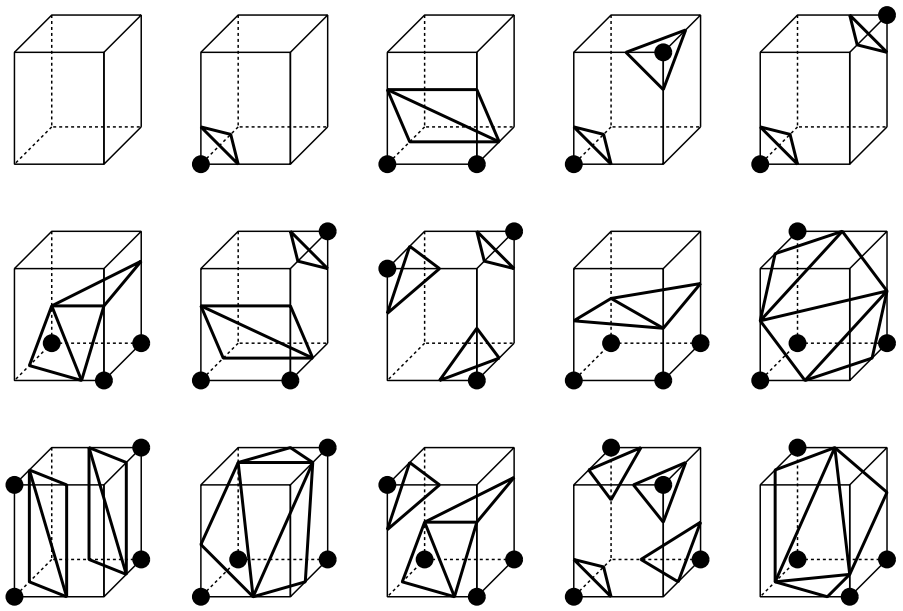


Figure 22.21. Possible intersections of the per-voxel tri-linear implicit surface and the voxel edges. From the possible 256 cases, these 15 topologically different cases can be identified, from which the others can be obtained by rotations. Grid points where the implicit function has the same sign are depicted by circles.

linear approximation (tri-linear means that fixing any two coordinates the function is linear with respect to the third coordinate). Due to the linear approximation an edge connecting two neighbouring grid points can intersect the surface at most once since linear equations may have at most one root. The density of the grid points should reflect this observation, then we have to define them so densely not to miss roots, that is, not to change the topology of the surface.

The method approximating the surface by a triangle mesh is called ***marching cubes algorithm.*** This algorithm first decides whether a grid point is inside or outside of the solid by checking the sign of function f. If two neighbouring grid points are of different type, the surface must go between them. The intersection of the surface and the edge between the neighbouring points, as well as the normal vector at the intersection are determined by linear interpolation. If one grid point is at $\vec{r}_1$, the other is at $\vec{r}_2$, and function f has different signs at these points, then the intersection of the tri-linear surface and line segment $(\vec{r}_1, \vec{r}_2)$ is:

$$\vec{r}_i = \vec{r}_1 \cdot \frac{f(\vec{r}_2)}{f(\vec{r}_2) - f(\vec{r}_1)} + \vec{r}_2 \cdot \frac{f(\vec{r}_1)}{f(\vec{r}_2) - f(\vec{r}_1)} \,.$$

The normal vector here is:

$$\vec{n}_i = \mathrm{grad} f(\vec{r}_1) \cdot \frac{f(\vec{r}_2)}{f(\vec{r}_2) - f(\vec{r}_1)} + \mathrm{grad} f(\vec{r}_2) \cdot \frac{f(\vec{r}_1)}{f(\vec{r}_2) - f(\vec{r}_1)} \,.$$

Having found the intersection points, triangles are defined using these points

as vertices. When defining these triangles, we have to take into account that a trilinear surface may intersect the voxel edges at most once. Such intersection occurs if function f has different signs at the two grid points. The number of possible variations of positive/negative signs at the 8 vertices of a cube is 256, from which 15 topologically different cases can be identified (Figure 22.21).

The algorithm inspects grid points one by one and assigns the sign of the function to them encoding negative sign by 0 and non-negative sign by 1. The resulting 8 bit code is a number in 0–255 which identifies the current case of intersection. If the code is 0, all voxel vertices are outside the solid, thus no voxel surface intersection is possible. Similarly, if the code is 255, the solid is completely inside, making the intersections impossible. To handle other codes, a table can be built which describes where the intersections show up and how they form triangles.

Exercises

22.3-1 Prove the two ears theorem by induction.
22.3-2 Develop an adaptive curve tessellation algorithm.
22.3-3 Prove that the Catmull-Clark subdivision curve and surface converge to a B-spline curve and surface, respectively.
22.3-4 Build a table to control the marching cubes algorithm, which describes where the intersections show up and how they form triangles.
22.3-5 Propose a marching cubes algorithm that does not require the gradients of the function, but estimates these gradients from its values.

22.4. Containment algorithms

When geometric models are processed, we often have to determine whether or not one object contains points belonging to the other object. If only yes/no answer is needed, we have a ***containment test*** problem. However, if the contained part also needs to be obtained, the applicable algorithm is called ***clipping.***

Containment test is also known as discrete time ***collision detection*** since if one object contains points from the other, then the two objects must have been collided before. Of course, checking collisions just at discrete time instances may miss certain collisions. To handle the collision problem robustly, continuous time collision detection is needed which also computes the time of the collision. Continuous time collision detection may use ray tracing (Section 22.6). In this section we only deal with the discrete time collision detection and the clipping of simple objects.

22.4.1. Point containment test

A solid defined by function f contains those (x, y, z) points which satisfy inequality $f(x, y, z) \geq 0$. It means that point containment test requires the evaluation of function f and the inspection of the sign of the result.

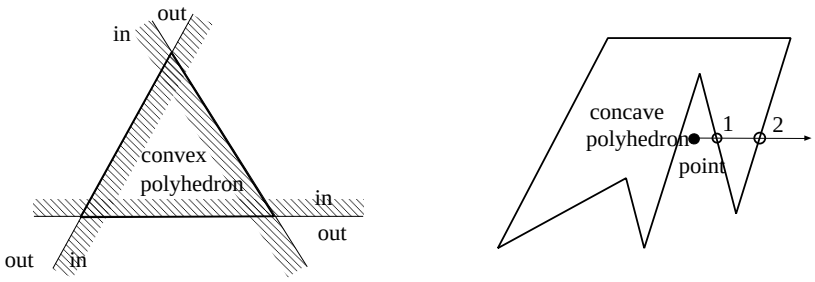


Figure 22.22. Polyhedron-point containment test. A convex polyhedron contains a point if the point is on that side of each face plane where the polyhedron is. To test a concave polyhedron, a half line is cast from the point and the number of intersections is counted. If the result is an odd number, then the point is inside, otherwise it is outside.

Half space

Based on equation (22.1), points belonging to a half space are identified by inequality

$$(\vec{r} - \vec{r}_0) \cdot \vec{n} \geq 0, \qquad n_x \cdot x + n_y \cdot y + n_z \cdot z + d \geq 0, \tag{22.9}$$

where the normal vector is supposed to point inward.

Convex polyhedron

Any convex polyhedron can be constructed as the intersection of halfspaces (left of Figure 22.22). The plane of each face subdivides the space into two parts, to an inner part where the polyhedron can be found, and to an outer part. Let us test the point against the planes of the faces. If the point is in the inner part with respect to all planes, then the point is inside the polyhedron. However, if the point is in the outer part with respect to at least one plane, then the point is outside of the polyhedron.

Concave polyhedron

As shown in Figure 22.22, let us cast a half line from the tested point and count the number of intersections with the faces of the polyhedron (the calculation of these intersections is discussed in Section 22.6). If the result is an odd number, then the point is inside, otherwise it is outside. Because of numerical inaccuracies we might have difficulties to count the number of intersections when the half line is close to the edges. In such cases, the simplest solution is to find another half line and carry out the test with that.

Polygon

The methods proposed to test the point in polyhedron can also be used for polygons limiting the space to the two-dimensional plane. For example, a point is in a general polygon if the half line originating at this point and lying in the plane of the polygon intersects the edges of the polygon odd times.

In addition to those methods, containment in convex polygons can be tested by adding the angles subtended by the edges from the point. If the sum is 360 degrees,

then the point is inside, otherwise it is outside. For convex polygons, we can also test whether the point is on the same side of the edges as the polygon itself. This algorithm is examined in details for a particularly important special case, when the polygon is a triangle.

Triangle

Let us consider a triangle of vertices $\vec{a}, \vec{b}$ and $\vec{c}$, and point $\vec{p}$ lying in the plane of the triangle. The point is inside the triangle if and only if it is on the same side of the boundary lines as the third vertex. Note that cross product $(\vec{b}-\vec{a}) \times (\vec{p}-\vec{a})$ has a different direction for point $\vec{p}$ lying on the different sides of oriented line $\vec{ab}$, thus the direction of this vector can be used to classify points (should point $\vec{p}$ be on line $\vec{ab}$, the result of the cross product is zero). During classification the direction of $(\vec{b}-\vec{a}) \times (\vec{p}-\vec{a})$ is compared to the direction of vector $\vec{n} = (\vec{b}-\vec{a}) \times (\vec{c}-\vec{a})$ where tested point $\vec{p}$ is replaced by third vertex $\vec{c}$. Note that vector $\vec{n}$ happens to be the normal vector of the triangle plane (Figure 22.23).

We can determine whether two vectors have the same direction (their angle is zero) or they have opposite directions (their angle is 180 degrees) by computing their scalar product and looking at the sign of the result. The scalar product of vectors of similar directions is positive. Thus if scalar product $((\vec{b}-\vec{a}) \times (\vec{p}-\vec{a})) \cdot \vec{n}$ is positive, then point $\vec{p}$ is on the same side of oriented line $\vec{ab}$ as $\vec{c}$. On the other hand, if this scalar product is negative, then $\vec{p}$ and $\vec{c}$ are on the opposite sides. Finally, if the result is zero, then point $\vec{p}$ is on line $\vec{ab}$. Point $\vec{p}$ is inside the triangle if and only if all the following three conditions are met:

$$\begin{aligned} ((\vec{b}-\vec{a}) \times (\vec{p}-\vec{a})) \cdot \vec{n} &\geq 0 , \\ ((\vec{c}-\vec{b}) \times (\vec{p}-\vec{b})) \cdot \vec{n} &\geq 0 , \\ ((\vec{a}-\vec{c}) \times (\vec{p}-\vec{c})) \cdot \vec{n} &\geq 0 . \end{aligned} \tag{22.10}$$

This test is robust since it gives correct result even if – due to numerical precision problems – point $\vec{p}$ is not exactly in the plane of the triangle as long as point $\vec{p}$ is in the prism obtained by perpendicularly extruding the triangle from the plane.

The evaluation of the test can be speeded up if we work in a two-dimensional projection plane instead of the three-dimensional space. Let us project point $\vec{p}$ as well as the triangle onto one of the coordinate planes. In order to increase numerical precision, that coordinate plane should be selected on which the area of the projected triangle is maximal. Let us denote the Cartesian coordinates of the normal vector by (n_x, n_y, n_z). If n_z has the maximum absolute value, then the projection of the maximum area is on coordinate plane xy. If n_x or n_y had the maximum absolute value, then planes yz or xz would be the right choice. Here only the case of maximum n_z is discussed.

First the order of vertices are changed in a way that when travelling from vertex $\vec{a}$ to vertex $\vec{b}$, vertex $\vec{c}$ is on the left side. Let us examine the equation of line $\vec{ab}$:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (x - b_x) + b_y = y .$$

According to Figure 22.24 point $\vec{c}$ is on the left of the line if c_y is above the line

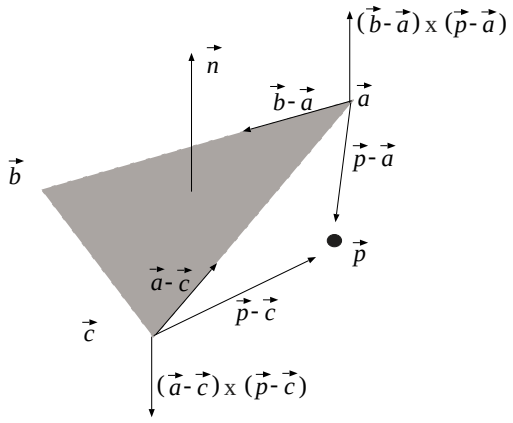


Figure 22.23. Point in triangle containment test. The figure shows that case when point $\vec{p}$ is on the left of oriented lines $\vec{ab}$ and $\vec{bc}$, and on the right of line $\vec{ca}$, that is, when it is not inside the triangle.

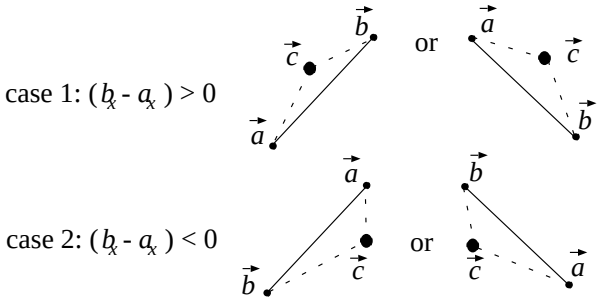


Figure 22.24. Point in triangle containment test on coordinate plane xy. Third vertex $\vec{c}$ can be either on the left or on the right side of oriented line $\vec{ab}$, which can always be traced back to the case of being on the left side by exchanging the vertices.

at $x = c_x$:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (c_x - b_x) + b_y < c_y \ .$$

Multiplying both sides by $(b_x - a_x)$, we get:

$$(b_y - a_y) \cdot (c_x - b_x) < (c_y - b_y) \cdot (b_x - a_x) \ .$$

In the second case the denominator of the slope of the line is negative. Point $\vec{c}$ is on the left of the line if c_y is below the line at $x = c_x$:

$$\frac{b_y - a_y}{b_x - a_x} \cdot (c_x - b_x) + b_y > c_y \ .$$

When the inequality is multiplied with negative denominator $(b_x - a_x)$, the relation is inverted:

$$(b_y - a_y) \cdot (c_x - b_x) < (c_y - b_y) \cdot (b_x - a_x) \ .$$

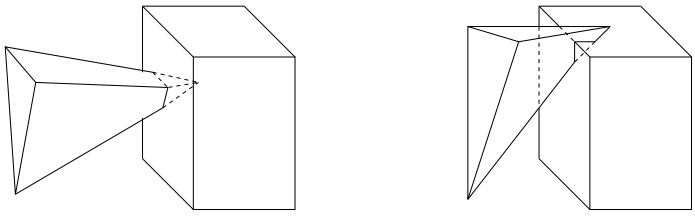


Figure 22.25. Polyhedron-polyhedron collision detection. Only a part of collision cases can be recognized by testing the containment of the vertices of one object with respect to the other object. Collision can also occur when only edges meet, but vertices do not penetrate to the other object.

Note that in both cases we obtained the same condition. If this condition is not met, then point $\vec{c}$ is not on the left of line $\vec{ab}$, but is on the right. Exchanging vertices $\vec{a}$ and $\vec{b}$ in this case, we can guarantee that $\vec{c}$ will be on the left of the new line $\vec{ab}$. It is also important to note that consequently point $\vec{a}$ will be on the left of line $\vec{bc}$ and point $\vec{b}$ will be on the left of line $\vec{ca}$.

In the second step the algorithm tests whether point $\vec{p}$ is on the left with respect to all three boundary lines since this is the necessary and sufficient condition of being inside the triangle:

$$\begin{aligned}(b_y - a_y) \cdot (p_x - b_x) &\leq (p_y - b_y) \cdot (b_x - a_x) \ , \\ (c_y - b_y) \cdot (p_x - c_x) &\leq (p_y - c_y) \cdot (c_x - b_x) \ , \\ (a_y - c_y) \cdot (p_x - a_x) &\leq (p_y - a_y) \cdot (a_x - c_x) \ .\end{aligned} \tag{22.11}$$

22.4.2. Polyhedron-polyhedron collision detection

Two polyhedra collide when a vertex of one of them meets a face of the other, and if they are not bounced off, the vertex goes into the internal part of the other object (Figure 22.25). This case can be recognized with the discussed containment test. All vertices of one polyhedron is tested for containment against the other polyhedron. Then the roles of the two polyhedra are exchanged.

Apart from the collision between vertices and faces, two edges may also meet without vertex penetration (Figure 22.25). In order to recognize this edge penetration case, all edges of one polyhedron are tested against all faces of the other polyhedron. The test for an edge and a face is started by checking whether or not the two endpoints of the edge are on opposite sides of the plane, using inequality (22.9). If they are, then the intersection of the edge and the plane is calculated, and finally it is decided whether the face contains the intersection point.

Polyhedra collision detection tests each edge of one polyhedron against each face of the other polyhedron, which results in an algorithm of quadratic time complexity with respect to the number of vertices of the polyhedra. Fortunately, the algorithm can be speeded up applying bounding volumes (Subsection 22.6.2). Let us assign a simple bounding object to each polyhedron. Popular choices for bounding volumes are the sphere and the box. During testing the collision of two objects, first their bounding volumes are examined. If the two bounding volumes do not collide, then

neither can the contained polyhedra collide. If the bounding volumes penetrate each other, then one polyhedra is tested against the other bounding volume. If this test is also positive, then finally the two polyhedra are tested. However, this last test is rarely required, and most of the collision cases can be solved by bounding volumes.

22.4.3. Clipping algorithms

Clipping takes an object defining the clipping region and removes those points from another object which are outside the clipping region. Clipping may alter the type of the object, which cannot be specified by a similar equation after clipping. To avoid this, we allow only those kinds of clipping regions and objects where the object type is not changed by clipping. Let us assume that the clipping region is a half space or a polyhedron, while the object to be clipped is a point, a line segment or a polygon.

If the object to be clipped is a point, then containment can be tested with the algorithms of the previous subsection. Based on the result of the containment test, the point is either removed or preserved.

Clipping a line segment onto a half space
Let us consider a line segment of endpoints $\vec{r}_1$ and $\vec{r}_2$, and of equation $\vec{r}(t) = \vec{r}_1 \cdot (1-t) + \vec{r}_2 \cdot t$, $(t \in [0,1])$, and a half plane defined by the following equation derived from equation (22.1):

$$(\vec{r} - \vec{r}_0) \cdot \vec{n} \geq 0, \qquad n_x \cdot x + n_y \cdot y + n_z \cdot z + d \geq 0 \ .$$

Three cases need to be distinguished:

1. If both endpoints of the line segment are in the half space, then all points of the line segment are inside, thus the whole segment is preserved.
2. If both endpoints are out of the half space, then all points of the line segment are out, thus the line segment should be completely removed.
3. If one of the endpoints is out, while the other is in, then the endpoint being out should be replaced by the intersection point of the line segment and the boundary plane of the half space. The intersection point can be calculated by substituting the equation of the line segment into the equation of the boundary plane and solving the resulting equation for the unknown parameter:

$$(\vec{r}_1 \cdot (1-t_i) + \vec{r}_2 \cdot t_i - \vec{r}_0) \cdot \vec{n} = 0, \quad \Longrightarrow \quad t_i = \frac{(\vec{r}_0 - \vec{r}_1) \cdot \vec{n}}{(\vec{r}_2 - \vec{r}_1) \cdot \vec{n}} \ .$$

Substituting parameter t_i into the equation of the line segment, the coordinates of the intersection point can also be obtained.

Clipping a polygon onto a half space
This clipping algorithm tests first whether a vertex is inside or not. If the vertex is in, then it is also the vertex of the resulting polygon. However, if it is out, it can be ignored. On the other hand, the resulting polygon may have vertices other than the vertices of the original polygon. These new vertices are the intersections of the edges and the boundary plane of the half space. Such intersection occurs when one

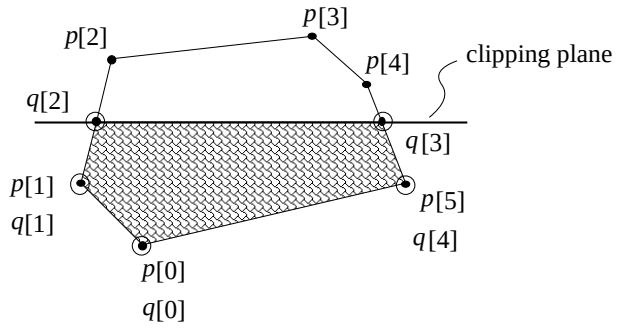


Figure 22.26. Clipping of simple convex polygon $\vec{p}[0], \ldots, \vec{p}[5]$ results in polygon $\vec{q}[0], \ldots, \vec{q}[4]$. The vertices of the resulting polygon are the inner vertices of the original polygon and the intersections of the edges and the boundary plane.

endpoint is in, but the other is out. While we are testing the vertices one by one, we should also check whether or not the next vertex is on the same side as the current vertex (Figure 22.26).

Suppose that the vertices of the polygon to be clipped are given in array $\mathbf{p} = \langle \vec{p}[0], \ldots, \vec{p}[n-1] \rangle$, and the vertices of the clipped polygon is expected in array $\mathbf{q} = \langle \vec{q}[0], \ldots, \vec{q}[m-1] \rangle$. The number of the vertices of the resulting polygon is stored in variable m. Note that the vertex followed by the ith vertex has usually index $(i+1)$, but not in the case of the last, $(n-1)$th vertex, which is followed by vertex 0. Handling the last vertex as a special case is often inconvenient. This can be eliminated by extending input array $\mathbf{p}$ by new element $\vec{p}[n] = \vec{p}[0]$, which holds the element of index 0 once again.

Using these assumptions, the ***Sutherland-Hodgeman polygon clipping algorithm*** is:

SUTHERLAND-HODGEMAN-POLYGON-CLIPPING($\mathbf{p}$)

```
1  m ← 0
2  for i ← 0 to n − 1
3      do if p⃗[i] is inside
4          then q⃗[m] ← p⃗[i] ▷ The ith vertex is the vertex of the resulting polygon.
5               m ← m + 1
6               if p⃗[i + 1] is outside
7                  then q⃗[m] ← EDGE-PLANE-INTERSECTION(p⃗[i], p⃗[i + 1])
8                       m ← m + 1
9          else if p⃗[i + 1] is inside
10                 then q⃗[m] ← EDGE-PLANE-INTERSECTION(p⃗[i], p⃗[i + 1])
11                      m ← m + 1
12 return q
```

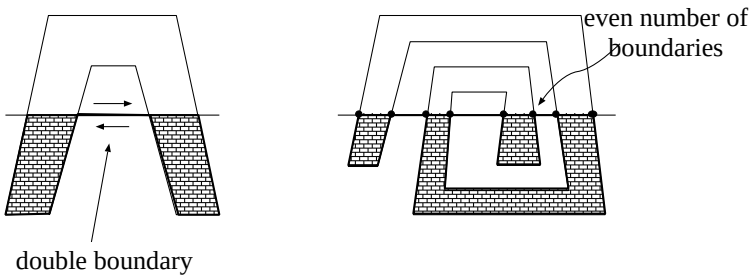


Figure 22.27. When concave polygons are clipped, the parts that should fall apart are connected by even number of edges.

Let us apply this algorithm for such a concave polygon which is expected to fall to several pieces during clipping (Figure 22.27). The algorithm storing the polygon in a single array is not able to separate the pieces and introduces even number of edges at parts where no edge could show up.

These even number of extra edges, however, pose no problems if the interior of the polygon is defined as follows: a point is inside the polygon if and only if starting a half line from here, the boundary polyline is intersected by odd number of times.

The presented algorithm is also suitable for clipping multiple connected polygons if the algorithm is executed separately for each closed polyline of the boundary.

Clipping line segments and polygons on a convex polyhedron

As stated, a convex polyhedron can be obtained as the intersection of the half spaces defined by the planes of the polyhedron faces (left of Figure 22.22). It means that clipping on a convex polyhedron can be traced back to a series of clipping steps on half spaces. The result of one clipping step on a half plane is the input of clipping on the next half space. The final result is the output of the clipping on the last half space.

Clipping a line segment on an AABB

Axis aligned bounding boxes, abbreviated as AABBs, play an important role in image synthesis.

Definition 22.11 *A box aligned parallel to the coordinate axes is called* ***AABB****. An AABB is specified with the minimum and maximum Cartesian coordinates:* $[x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}]$.

Although when an object is clipped on an AABB, the general algorithms that clip on a convex polyhedron could also be used, the importance of AABBs is acknowledged by developing algorithms specially tuned for this case.

When a line segment is clipped to a polyhedron, the algorithm would test the line segment with the plane of each face, and the calculated intersection points may turn out to be unnecessary later. We should thus find an appropriate order of planes which makes the number of unnecessary intersection calculations minimal. A simple method that implements this idea is the ***Cohen-Sutherland line clipping***

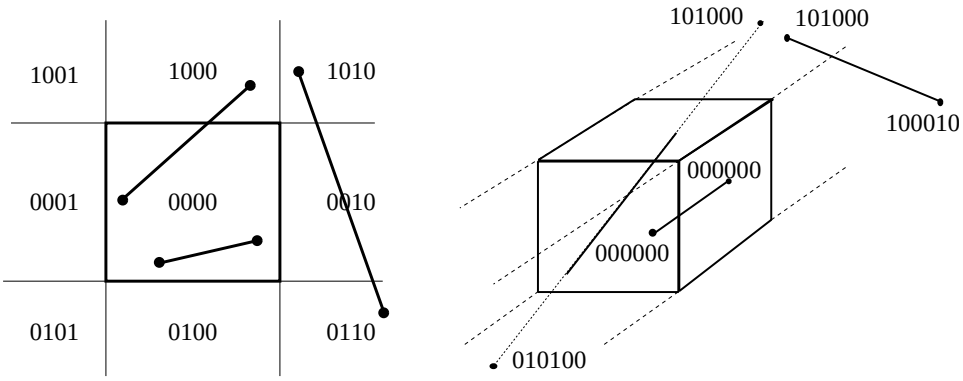


Figure 22.28. The 4-bit codes of the points in a plane and the 6-bit codes of the points in space.

algorithm.

Let us assign code bit 1 to a point that is outside with respect to a clipping plane, and code bit 0 if the point is inside with respect to this plane. Since an AABB has 6 sides, we get 6 bits forming a 6-bit code word (Figure 22.28). The interpretation of code bits $C[0], \ldots, C[5]$ is the following:

$$C[0] = \begin{cases} 1, & x \le x_{min}, \\ 0 & \text{otherwise.} \end{cases} \quad C[1] = \begin{cases} 1, & x \ge x_{max}, \\ 0 & \text{otherwise.} \end{cases} \quad C[2] = \begin{cases} 1, & y \le y_{min}\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

$$C[3] = \begin{cases} 1, & y \ge y_{max}, \\ 0 & \text{otherwise.} \end{cases} \quad C[4] = \begin{cases} 1, & z \le z_{min}, \\ 0 & \text{otherwise.} \end{cases} \quad C[5] = \begin{cases} 1, & z \ge z_{max}\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

Points of code word 000000 are obviously inside, points of other code words are outside (Figure 22.28). Let the code words of the two endpoints of the line segment be C_1 and C_2, respectively. If both of them are zero, then both endpoints are inside, thus the line segment is completely inside (trivial accept). If the two code words contain bit 1 at the same location, then none of the endpoints are inside with respect to the plane associated with this code bit. This means that the complete line segment is outside with respect to this plane, and can be rejected (trivial reject). This examination can be executed by applying the bitwise AND operation on code words C_1 and C_2 (with the notations of the C programming language C_1 & C_2), and checking whether or not the result is zero. If it is not zero, there is a bit where both code words have value 1.

Finally, if none of the two trivial cases hold, then there must be a bit which is 0 in one code word and 1 in the other. This means that one endpoint is inside and the other is outside with respect to the plane corresponding to this bit. The line segment should be clipped on this plane. Then the same procedure should be repeated starting with the evaluation of the code bits. The procedure is terminated when the conditions of either the trivial accept or the trivial reject are met.

The Cohen-Sutherland line clipping algorithm returns the endpoints of the clipped line by modifying the original vertices and indicates with TRUE return value if the line is not completely rejected:

COHEN-SUTHERLAND-LINE-CLIPPING($\vec{r}_1, \vec{r}_2$)

```
1  C_1 ← codeword of r_1                    ▷ Code bits by checking the inequalities.
2  C_2 ← codeword of r_2
3  while TRUE
4        do if C_1 = 0 AND C_2 = 0
5              then return TRUE              ▷ Trivial accept: inner line segment exists.
6           if C_1 & C_2 ≠ 0
7              then return FALSE             ▷ Trivial reject: no inner line segment exists.
8           f ← index of the first bit where C_1 and C_2 differ
9           r_i ← intersection of line segment (r_1, r_2) and the plane of index f
10          C_i ← codeword of r_i
11          if C_1[f] = 1
12             then r_1 ← r_i
13                  C_1 ← C_i                ▷ r_1 is outside w.r.t. plane f.
14             else r_2 ← r_i
15                  C_2 ← C_i                ▷ r_2 is outside w.r.t. plane f.
```

Exercises

22.4-1 Propose approaches to reduce the quadratic complexity of polyhedron-polyhedron collision detection.

22.4-2 Develop a containment test to check whether a point is in a CSG-tree.

22.4-3 Develop an algorithm clipping one polygon onto a concave polygon.

22.4-4 Find an algorithm computing the bounding sphere and the bounding AABB of a polyhedron.

22.4-5 Develop an algorithm that tests the collision of two triangles in the plane.

22.4-6 Generalize the Cohen-Sutherland line clipping algorithm to convex polyhedron clipping region.

22.4-7 Propose a method for clipping a line segment on a sphere.

22.5. Translation, distortion, geometric transformations

Objects in the virtual world may move, get distorted, grow or shrink, that is, their equations may also depend on time. To describe dynamic geometry, we usually apply two functions. The first function selects those points of space, which belong to the object in its reference state. The second function maps these points onto points defining the object in an arbitrary time instance. Functions mapping the space onto itself are called ***transformations.*** A transformation maps point $\vec{r}$ to point $\vec{r}\,' = \mathcal{T}(\vec{r})$. If the transformation is invertible, we can also find the original for some transformed point $\vec{r}\,'$ using inverse transformation $\mathcal{T}^{-1}(\vec{r}\,')$.

If the object is defined in its reference state by inequality $f(\vec{r}) \geq 0$, then the

points of the transformed object are

$$\{\vec{r}\,' : f(\mathcal{T}^{-1}(\vec{r}\,')) \geq 0\} \ , \tag{22.12}$$

since the originals belong to the set of points of the reference state.

Parametric equations define the Cartesian coordinates of the points directly. Thus the transformation of parametric surface $\vec{r} = \vec{r}(u, v)$ requires the transformations of its points

$$\vec{r}\,'(u, v) = \mathcal{T}(\vec{r}(u, v)) \ . \tag{22.13}$$

Similarly, the transformation of curve $\vec{r} = \vec{r}(t)$ is:

$$\vec{r}\,'(t) = \mathcal{T}(\vec{r}(t)) \ . \tag{22.14}$$

Transformation $\mathcal{T}$ may change the type of object in the general case. It can happen, for example, that a simple triangle or a sphere becomes a complicated shape, which are hard to describe and handle. Thus it is worth limiting the set of allowed transformations. Transformations mapping planes onto planes, lines onto lines and points onto points are particularly important. In the next subsection we consider the class of ***homogeneous linear transformations***, which meet this requirement.

22.5.1. Projective geometry and homogeneous coordinates

So far the construction of the virtual world has been discussed using the means of the Euclidean geometry, which gave us many important concepts such as distance, parallelism, angle, etc. However, when the transformations are discussed in details, many of these concepts are unimportant, and can cause confusion. For example, parallelism is a relationship of two lines which can lead to singularities when the intersection of two lines is considered. Therefore, transformations are discussed in the context of another framework, called projective geometry.

The axioms of ***projective geometry*** turn around the problem of parallel lines by ignoring the concept of parallelism altogether, and state that two different lines always have an intersection. To cope with this requirement, every line is extended by a "point at infinity" such that two lines have the same extra point if and only if the two lines are parallel. The extra point is called the ***ideal point***. The ***projective space*** contains the points of the Euclidean space (these are the so called ***affine points***) and the ideal points. An ideal point "glues" the "ends" of an Euclidean line, making it topologically similar to a circle. Projective geometry preserves that axiom of the Euclidean geometry which states that two points define a line. In order to make it valid for ideal points as well, the set of lines of the Euclidean space is extended by a new line containing the ideal points. This new line is called the ***ideal line.*** Since the ideal points of two lines are the same if and only if the two lines are parallel, the ideal lines of two planes are the same if and only if the two planes are parallel. Ideal lines are on the ***ideal plane***, which is added to the set of planes of the Euclidean space. Having made these extensions, no distinction is needed between the affine and ideal points. They are equal members of the projective space.

Introducing analytic geometry we noted that everything should be described by numbers in computer graphics. Cartesian coordinates used so far are in one to

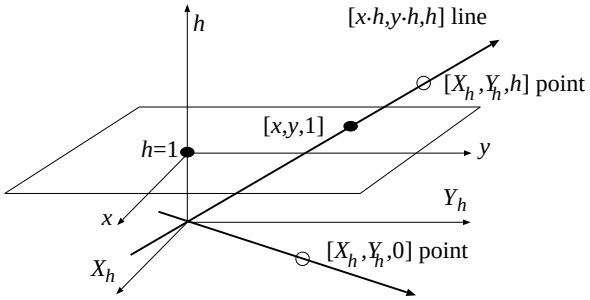


Figure 22.29. The embedded model of the projective plane: the projective plane is embedded into a three-dimensional Euclidean space, and a correspondence is established between points of the projective plane and lines of the embedding three-dimensional Euclidean space by fitting the line to the origin of the three-dimensional space and the given point.

one relationship with the points of Euclidean space, thus they are inappropriate to describe the points of the projective space. For the projective plane and space, we need a different algebraic base.

Projective plane

Let us consider first the projective plane and find a method to describe its points by numbers. To start, a Cartesian coordinate system x, y is set up in this plane. Simultaneously, another Cartesian system X_h, Y_h, h is established in the three-dimensional space embedding the plane in a way that axes X_h, Y_h are parallel to axes x, y, the plane is perpendicular to axis h, the origin of the Cartesian system of the plane is in point $(0, 0, 1)$ of the three-dimensional space, and the points of the plane satisfy equation $h = 1$. The projective plane is thus embedded into a three-dimensional Euclidean space where points are defined by Descartes-coordinates (Figure 22.29). To describe a point of the projective plane by numbers, a correspondence is found between the points of the projective plane and the points of the embedding Euclidean space. An appropriate correspondence assigns that line of the Euclidean space to either affine or ideal point P of the projective plane, which is defined by the origin of the coordinate system of the space and point P.

Points of an Euclidean line that crosses the origin can be defined by parametric equation $[t \cdot X_h, t \cdot Y_h, t \cdot h]$ where t is a free real parameter. If point P is an affine point of the projective plane, then the corresponding line is not parallel with plane $h = 1$ (i.e. h is not constant zero). Such line intersects the plane of equation $h = 1$ at point $[X_h/h, Y_h/h, 1]$, thus the Cartesian coordinates of point P in planar coordinate system x, y are $(X_h/h, Y_h/h)$. On the other hand, if point P is ideal, then the corresponding line is parallel to the plane of equation $h = 1$ (i.e. $h = 0$). The direction of the ideal point is given by vector (X_h, Y_h).

The presented approach assigns three dimensional lines crossing the origin and eventually $[X_h, Y_h, h]$ triplets to both the affine and the ideal points of the projective plane. These triplets are called the ***homogeneous coordinates*** of a point in the projective plane. Homogeneous coordinates are enclosed by brackets to distinguish them from Cartesian coordinates.

A three-dimensional line crossing the origin and describing a point of the pro-

jective plane can be defined by its arbitrary point except the origin. Consequently, all three homogeneous coordinates cannot be simultaneously zero, and homogeneous coordinates can be freely multiplied by the same non-zero scalar without changing the described point. This property justifies the name "homogeneous".

It is often convenient to select that triplet from the homogeneous coordinates of an affine point, where the third homogeneous coordinate is 1 since in this case the first two homogeneous coordinates are identical to the Cartesian coordinates:

$$X_h = x, \qquad Y_h = y, \qquad h = 1 \; . \tag{22.15}$$

>From another point of view, Cartesian coordinates of an affine point can be converted to homogeneous coordinates by extending the pair by a third element of value 1.

The embedded model also provides means to define the equations of the lines and line segments of the projective space. Let us select two different points on the projective plane and specify their homogeneous coordinates. The two points are different if homogeneous coordinates $[X_h^1, Y_h^1, h^1]$ of the first point cannot be obtained as a scalar multiple of homogeneous coordinates $[X_h^2, Y_h^2, h^2]$ of the other point. In the embedding space, triplet $[X_h, Y_h, h]$ can be regarded as Cartesian coordinates, thus the ***equation of the line*** fitted to points $[X_h^1, Y_h^1, h^1]$ and $[X_h^2, Y_h^2, h^2]$ is:

$$\begin{aligned} X_h(t) &= X_h^1 \cdot (1-t) + X_h^2 \cdot t \; , \\ Y_h(t) &= Y_h^1 \cdot (1-t) + Y_h^2 \cdot t \; , \\ h(t) &= h^1 \cdot (1-t) + h^2 \cdot t \; . \end{aligned} \tag{22.16}$$

If $h(t) \neq 0$, then the affine points of the projective plane can be obtained by projecting the three-dimensional space onto the plane of equation $h = 1$. Requiring the two points be different, we excluded the case when the line would be projected to a single point. Hence projection maps lines to lines. Thus the presented equation really identifies the homogeneous coordinates defining the points of the line. If $h(t) = 0$, then the equation expresses the ideal point of the line.

If parameter t has an arbitrary real value, then the points of a line are defined. If parameter t is restricted to interval $[0, 1]$, then we obtain the line segment defined by the two endpoints.

Projective space

We could apply the same method to introduce homogeneous coordinates of the projective space as we used to define the homogeneous coordinates of the projective plane, but this approach would require the embedding of the three-dimensional projective space into a four-dimensional Euclidean space, which is not intuitive. We would rather discuss another construction, which works in arbitrary dimensions. In this construction, a point is described as the centre of mass of a mechanical system. To identify a point, let us place weight X_h at reference point $\vec{p}_1$, weight Y_h at reference point $\vec{p}_2$, weight Z_h at reference point $\vec{p}_3$, and weight w at reference point $\vec{p}_4$. The centre of mass of this mechanical system is:

$$\vec{r} = \frac{X_h \cdot \vec{p}_1 + Y_h \cdot \vec{p}_2 + Z_h \cdot \vec{p}_3 + w \cdot \vec{p}_4}{X_h + Y_h + Z_h + w} \; .$$

Let us denote the total weight by $h = X_h + Y_h + Z_h + w$. By definition, elements of quadruple $[X_h, Y_h, Z_h, h]$ are the ***homogeneous coordinates*** of the centre of mass.

To find the correspondence between homogeneous and Cartesian coordinates, the relationship of the two coordinate systems (the relationship of the basis vectors and the origin of the Cartesian coordinate system and of the reference points of the homogeneous coordinate system) must be established. Let us assume, for example, that the reference points of the homogeneous coordinate system are in points (1,0,0), (0,1,0), (0,0,1), and (0,0,0) of the Cartesian coordinate system. The centre of mass (assuming that total weight h is not zero) is expressed in Cartesian coordinates as follows:

$$\vec{r}[X_h, Y_h, Z_h, h] = \frac{1}{h} \cdot (X_h \cdot (1,0,0) + Y_h \cdot (0,1,0) + Z_h \cdot (0,0,1) + w \cdot (0,0,0)) = \left(\frac{X_h}{h}, \frac{Y_h}{h}, \frac{Z_h}{h} \right).$$

Hence the correspondence between homogeneous coordinates $[X_h, Y_h, Z_h, h]$ and Cartesian coordinates (x, y, z) is $(h \neq 0)$:

$$x = \frac{X_h}{h}, \qquad y = \frac{Y_h}{h}, \qquad z = \frac{Z_h}{h} \ . \tag{22.17}$$

The equations of ***lines in the projective space*** can be obtained either deriving them from the embedding four-dimensional Cartesian space, or using the centre of mass analogy:

$$\begin{aligned} X_h(t) &= X_h^1 \cdot (1-t) + X_h^2 \cdot t \ , \\ Y_h(t) &= Y_h^1 \cdot (1-t) + Y_h^2 \cdot t \ , \\ Z_h(t) &= Z_h^1 \cdot (1-t) + Z_h^2 \cdot t \ , \\ h(t) &= h^1 \cdot (1-t) + h^2 \cdot t \ . \end{aligned} \tag{22.18}$$

If parameter t is restricted to interval $[0, 1]$, then we obtain the equation of the ***projective line segment.***

To find the equation of the ***projective plane***, the equation of the Euclidean plane is considered (equation 22.1). The Cartesian coordinates of the points on an Euclidean plane satisfy the following implicit equation

$$n_x \cdot x + n_y \cdot y + n_z \cdot z + d = 0 \ .$$

Using the correspondence between the Cartesian and homogeneous coordinates (equation 22.17) we still describe the points of the Euclidean plane but now with homogeneous coordinates:

$$n_x \cdot \frac{X_h}{h} + n_y \cdot \frac{Y_h}{h} + n_z \cdot \frac{Z_h}{h} + d = 0 \ .$$

Let us multiply both sides of this equation by h, and add those points to the plane which have $h = 0$ coordinate and satisfy this equation. With this step the set of points of the Euclidean plane is extended with the ideal points, that is, we obtained the set of points belonging to the projective plane. Hence the equation of the projective plane is a homogeneous linear equation:

$$n_x \cdot X_h + n_y \cdot Y_h + n_z \cdot Z_h + d \cdot h = 0 \ , \tag{22.19}$$

or in matrix form:

$$[X_h, Y_h, Z_h, h] \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = 0 \ . \tag{22.20}$$

Note that points and planes are described by row and column vectors, respectively. Both the quadruples of points and the quadruples of planes have the homogeneous property, that is, they can be multiplied by non-zero scalars without altering the solutions of the equation.

22.5.2. Homogeneous linear transformations

Transformations defined as the multiplication of the homogeneous coordinate vector of a point by a constant 4×4 $\mathbf{T}$ matrix are called ***homogeneous linear transformations***:

$$[X'_h, Y'_h, Z'_h, h'] = [X_h, Y_h, Z_h, h] \cdot \mathbf{T} \ . \tag{22.21}$$

Theorem 22.12 *Homogeneous linear transformations map points to points.*

Proof A point can be defined by homogeneous coordinates in form $\lambda \cdot [X_h, Y_h, Z_h, h]$, where λ is an arbitrary, non-zero constant. The transformation results in $\lambda \cdot [X'_h, Y'_h, Z'_h, h'] = \lambda \cdot [X_h, Y_h, Z_h, h] \cdot \mathbf{T}$ when a point is transformed, which are the λ-multiples of the same vector, thus the result is a single point in homogeneous coordinates. ■

Note that due to the homogeneous property, homogeneous transformation matrix $\mathbf{T}$ is not unambiguous, but can be freely multiplied by non-zero scalars without modifying the realized mapping.

Theorem 22.13 *Invertible homogeneous linear transformations map lines to lines.*

Proof Let us consider the parametric equation of a line:

$$[X_h(t), Y_h(t), Z_h(t), h(t)] = [X_h^1, Y_h^1, Z_h^1, h^1] \cdot (1-t) + [X_h^2, Y_h^2, Z_h^2, h^2] \cdot t, \quad t = (-\infty, \infty) \ ,$$

and transform the points of this line by multiplying the quadruples with the transformation matrix:

$$\begin{aligned} [X'_h(t), Y'_h(t), Z'_h(t), h'(t)] &= [X_h(t), Y_h(t), Z_h(t), h(t)] \cdot \mathbf{T} \\ &= [X_h^1, Y_h^1, Z_h^1, h^1] \cdot \mathbf{T} \cdot (1-t) + [X_h^2, Y_h^2, Z_h^2, h^2] \cdot \mathbf{T} \cdot t \\ &= [X_h^{1'}, Y_h^{1'}, Z_h^{1'}, h^{1'}] \cdot (1-t) + [X_h^{2'}, Y_h^{2'}, Z_h^{2'}, h^{2'}] \cdot t \ , \end{aligned}$$

where $[X_h^{1'}, Y_h^{1'}, Z_h^{1'}, h^{1'}]$ and $[X_h^{2'}, Y_h^{2'}, Z_h^{2'}, h^{2'}]$ are the transformations of $[X_h^1, Y_h^1, Z_h^1, h^1]$ and $[X_h^2, Y_h^2, Z_h^2, h^2]$, respectively. Since the transformation is invertible, the two points are different. The resulting equation is the equation of a line fitted to the transformed points. ■

We note that if we had not required the invertibility of the the transformation, then it could have happened that the transformation would have mapped the two

points to the same point, thus the line would have degenerated to single point.

If parameter t is limited to interval $[0, 1]$, then we obtain the equation of the projective line segment, thus we can also state that a homogeneous linear transformation maps a line segment to a line segment. Even more generally, a homogeneous linear transformation maps convex combinations to convex combinations. For example, triangles are also mapped to triangles.

However, we have to be careful when we try to apply this theorem in the Euclidean plane or space. Let us consider a line segment as an example. If coordinate h has different sign at the two endpoints, then the line segment contains an ideal point. Such projective line segment can be intuitively imagined as two half lines and an ideal point sticking the "endpoints" of these half lines at infinity, that is, such line segment is the complement of the line segment we are accustomed to. It may happen that before the transformation, coordinates h of the endpoints have similar sign, that is, the line segment meets our intuitive image about Euclidean line segments, but after the transformation, coordinates h of the endpoints will have different sign. Thus the transformation wraps around our line segment.

Theorem 22.14 *Invertible homogeneous linear transformations map planes to planes.*

Proof The originals of transformed points $[X'_h, Y'_h, Z'_h, h']$ defined by $[X_h, Y_h, Z_h, h] = [X'_h, Y'_h, Z'_h, h'] \cdot \mathbf{T}^{-1}$ are on a plane, thus satisfy the original equation of the plane:

$$[X_h, Y_h, Z_h, h] \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = [X'_h, Y'_h, Z'_h, h'] \cdot \mathbf{T}^{-1} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} = 0 \ .$$

Due to the associativity of matrix multiplication, the transformed points also satisfy equation

$$[X'_h, Y'_h, Z'_h, h'] \cdot \begin{bmatrix} n'_x \\ n'_y \\ n'_z \\ d' \end{bmatrix} = 0 \ ,$$

which is also a plane equation, where

$$\begin{bmatrix} n'_x \\ n'_y \\ n'_z \\ d' \end{bmatrix} = \mathbf{T}^{-1} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ d \end{bmatrix} \ .$$

This result can be used to obtain the normal vector of a transformed plane. ■

An important subclass of homogeneous linear transformations is the set of ***affine transformations,*** where the Cartesian coordinates of the transformed point are linear functions of the original Cartesian coordinates:

$$[x', y', z'] = [x, y, z] \cdot \mathbf{A} + [p_x, p_y, p_z] \ , \tag{22.22}$$

where vector $\vec{p}$ describes translation, $\mathbf{A}$ is a matrix of size 3×3 and expresses rotation, scaling, mirroring, etc., and their arbitrary combination. For example, the rotation around axis (t_x, t_y, t_z), $(|(t_x, t_y, t_z)| = 1)$ by angle ϕ is given by the following matrix

$$\mathbf{A} = \begin{bmatrix} (1-t_x^2)\cos\phi + t_x^2 & t_x t_y(1-\cos\phi) + t_z\sin\phi & t_x t_z(1-\cos\phi) - t_y\sin\phi \\ t_y t_x(1-\cos\phi) - t_z\sin\phi & (1-t_y^2)\cos\phi + t_y^2 & t_x t_z(1-\cos\phi) + t_x\sin\phi \\ t_z t_x(1-\cos\phi) + t_y\sin\phi & t_z t_y(1-\cos\phi) - t_x\sin\phi & (1-t_z^2)\cos\phi + t_z^2 \end{bmatrix}.$$

This expression is known as the ***Rodrigues formula***.

Affine transformations map the Euclidean space onto itself, and transform parallel lines to parallel lines. Affine transformations are also homogeneous linear transformations since equation (22.22) can also be given as a 4×4 matrix operation, having changed the Cartesian coordinates to homogeneous coordinates by adding a fourth coordinate of value 1:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \begin{bmatrix} A_{11} & A_{12} & A_{13} & 0 \\ A_{21} & A_{22} & A_{23} & 0 \\ A_{31} & A_{32} & A_{33} & 0 \\ p_x & p_y & p_z & 1 \end{bmatrix} = [x, y, z, 1] \cdot \mathbf{T}\,. \qquad (22.23)$$

A further specialization of affine transformations is the set of ***congruence transformations*** (isometries) which are distance and angle preserving.

Theorem 22.15 *In a congruence transformation the rows of matrix* $\mathbf{A}$ *have unit length and are orthogonal to each other.*

Proof Let us use the property that a congruence is distance and angle preserving for the case when the origin and the basis vectors of the Cartesian system are transformed. The transformation assigns point (p_x, p_y, p_z) to the origin and points $(A_{11} + p_x, A_{12} + p_y, A_{13} + p_z)$, $(A_{21} + p_x, A_{22} + p_y, A_{23} + p_z)$, and $(A_{31} + p_x, A_{32} + p_y, A_{33} + p_z)$ to points $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, respectively. Because the distance is preserved, the distances between the new points and the new origin are still 1, thus $|(A_{11}, A_{12}, A_{13})| = 1$, $|(A_{21}, A_{22}, A_{23})| = 1$, and $|(A_{31}, A_{32}, A_{33})| = 1$. On the other hand, because the angle is also preserved, vectors (A_{11}, A_{12}, A_{13}), (A_{21}, A_{22}, A_{23}), and (A_{31}, A_{32}, A_{33}) are also perpendicular to each other. ■

Exercises

22.5-1 Using the Cartesian coordinate system as an algebraic basis, prove the axioms of the Euclidean geometry, for example, that two points define a line, and that two different lines may intersect each other at most at one point.

22.5-2 Using the homogeneous coordinates as an algebraic basis, prove an axiom of the projective geometry stating that two different lines intersect each other in exactly one point.

22.5-3 Prove that homogeneous linear transformations map line segments to line segments using the centre of mass analogy.

22.5-4 How does an affine transformation modify the volume of an object?

22.5-5 Give the matrix of that homogeneous linear transformation which translates by vector $\vec{p}$.

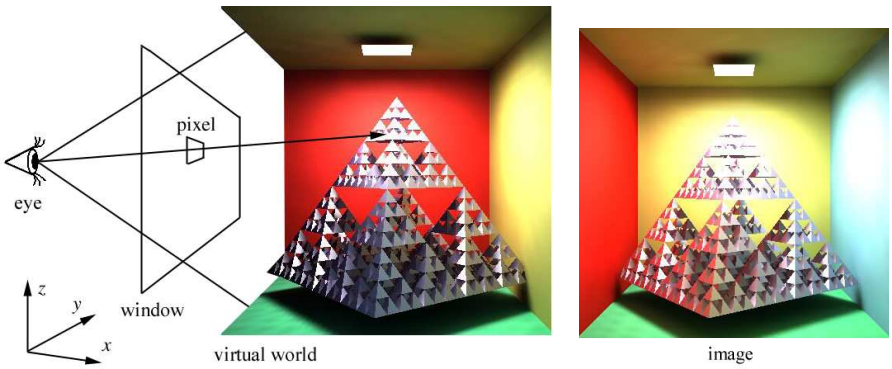


Figure 22.30. Ray tracing.

22.5-6 Prove the Rodrigues formula.
22.5-7 A solid defined by inequality $f(\vec{r}) \geq 0$ in time $t = 0$ moves with uniform constant velocity $\vec{v}$. Let us find the inequality of the solid at an arbitrary time instance t.
22.5-8 Prove that if the rows of matrix $\mathbf{A}$ are of unit length and are perpendicular to each other, then the affine transformation is a congruence. Show that for such matrices $\mathbf{A}^{-1} = \mathbf{A}^T$.
22.5-9 Give that homogeneous linear transformation which projects the space from point $\vec{c}$ onto a plane of normal $\vec{n}$ and place vector $\vec{r}_0$.
22.5-10 Show that five point correspondences unambiguously identify a homogeneous linear transformation if no four points are co-planar.

22.6. Rendering with ray tracing

When a virtual world is rendered, we have to identify the surfaces visible in different directions from the virtual eye. The set of possible directions is defined by a rectangle shaped window which is decomposed to a grid corresponding to the pixels of the screen (Figure 22.30). Since a pixel has a unique colour, it is enough to solve the visibility problem in a single point of each pixel, for example, in the points corresponding to pixel centres.

The surface visible at a direction from the eye can be identified by casting a half line, called ***ray***, and identifying its intersection closest to the eye position. This operation is called ***ray tracing***. Ray tracing has many applications. For example, ***shadow*** computation tests whether or not a point is occluded from the light source, which requires a ray to be sent from the point at the direction of the light source and the determination whether this ray intersects any surface closer than the light source. Ray tracing is also used by ***collision detection*** since a point moving with constant and uniform speed collides that surface which is first intersected by the ray describing the motion of the point.

A ray is defined by the following equation:

$$r\vec{a}y(t) = \vec{s} + \vec{v} \cdot t, \qquad (t > 0) \ , \tag{22.24}$$

where $\vec{s}$ is the place vector of the ***ray origin***, $\vec{v}$ is the ***direction of the ray***, and ***ray parameter*** t characterizes the distance from the origin. Let us suppose that direction vector $\vec{v}$ has unit length. In this case parameter t is the real distance, otherwise it would only be proportional to the distance[3]. If parameter t is negative, then the point is behind the eye and is obviously not visible. The identification of the closest intersection with the ray means the determination of the intersection point having the smallest, positive ray parameter. In order to find the closest intersection, the intersection calculation is tried with each surface, and the closest is retained. This algorithm obtaining the first intersection is:

RAY-FIRST-INTERSECTION$(\vec{s}, \vec{v})$

```
1  t ← t_max                          ▷ Initialization to the maximum size in the virtual world.
2  for each object o
3      do t_o ← RAY-SURFACE-INTERSECTION(s, v)
                                       ▷ Negative if no intersection exists.
4          if 0 ≤ t_o < t              ▷ Is the new intersection closer?
5             then t ← t_o             ▷ Ray parameter of the closest intersection so far.
6                  o_visible ← o       ▷ Closest object so far.
7  if t < t_max then                   ▷ Has been intersection at all?
8     then x ← s + v · t               ▷ Intersection point using the ray equation.
9          return t, x, o_visible
10    else return "no intersection"    ▷ No intersection.
```

This algorithm inputs the ray defined by origin $\vec{s}$ and direction $\vec{v}$, and outputs the ray parameter of the intersection in variable t, the intersection point in $\vec{x}$, and the visible object in $o_{visible}$. The algorithm calls function RAY-SURFACE-INTERSECTION for each object, which determines the intersection of the ray and the given object, and indicates with a negative return value if no intersection exists. Function RAY-SURFACE-INTERSECTION should be implemented separately for each surface type.

22.6.1. Ray surface intersection calculation

The identification of the intersection between a ray and a surface requires the solution of an equation. The intersection point is both on the ray and on the surface, thus it can be obtained by inserting the ray equation into the equation of the surface and solving the resulting equation for the unknown ray parameter.

Intersection calculation for implicit surfaces
For implicit surfaces of equation $f(\vec{r}) = 0$, the intersection can be calculated by solving the following scalar equation for t: $f(\vec{s} + \vec{v} \cdot t) = 0$.

[3] In collision detection $\vec{v}$ is not a unit vector, but the velocity of the moving point since this makes ray parameter t express the collision time.

Let us take the example of ***quadrics*** that include the sphere, the ellipsoid, the cylinder, the cone, the paraboloid, etc. The implicit equation of a general quadric contains a quadratic form:

$$[x, y, z, 1] \cdot \mathbf{Q} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0 \ ,$$

where $\mathbf{Q}$ is a 4×4 matrix. Substituting the ray equation into the equation of the surface, we obtain

$$[s_x + v_x \cdot t, s_y + v_y \cdot t, s_z + v_z \cdot t, 1] \cdot \mathbf{Q} \cdot \begin{bmatrix} s_x + v_x \cdot t \\ s_y + v_y \cdot t \\ s_z + v_z \cdot t \\ 1 \end{bmatrix} = 0 \ .$$

Rearranging the terms, we get a second order equation for unknown parameter t:

$$t^2 \cdot (\mathbf{v} \cdot \mathbf{Q} \cdot \mathbf{v}^T) + t \cdot (\mathbf{s} \cdot \mathbf{Q} \cdot \mathbf{v}^T + \mathbf{v} \cdot \mathbf{Q} \cdot \mathbf{s}^T) + (\mathbf{s} \cdot \mathbf{Q} \cdot \mathbf{s}^T) = 0 \ ,$$

where $\mathbf{v} = [v_x, v_y, v_z, 0]$ and $\mathbf{s} = [s_x, s_y, s_z, 1]$.

This equation can be solved using the solution formula of second order equations. Now we are interested in only the real and positive roots. If two such roots exist, then the smaller one corresponds to the intersection closer to the origin of the ray.

Intersection calculation for parametric surfaces

The intersection of parametric surface $\vec{r} = \vec{r}(u, v)$ and the ray is calculated by first solving the following equation for unknown parameters u, v, t

$$\vec{r}(u, v) = \vec{s} + t \cdot \vec{v} \ ,$$

then checking whether or not t is positive and parameters u, v are inside the allowed parameter range of the surface.

Roots of non-linear equations are usually found by numeric methods. On the other hand, the surface can also be approximated by a triangle mesh, which is intersected by the ray. Having obtained the intersection on the coarse mesh, the mesh around this point is refined, and the intersection calculation is repeated with the refined mesh.

Intersection calculation for a triangle

To compute the ray intersection for a ***triangle*** of vertices $\vec{a}$, $\vec{b}$, and $\vec{c}$, first the ray intersection with the plane of the triangle is found. Then it is decided whether or not the intersection point with the plane is inside the triangle. The normal and a place vector of the triangle plane are $\vec{n} = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$, and $\vec{a}$, respectively, thus points $\vec{r}$ of the plane satisfy the following equation:

$$\vec{n} \cdot (\vec{r} - \vec{a}) = 0 \ . \tag{22.25}$$

The intersection of the ray and this plane is obtained by substituting the ray equation (equation (22.24)) into this plane equation, and solving it for unknown parameter t. If root t^* is positive, then it is inserted into the ray equation to get the intersection point with the plane. However, if the root is negative, then the intersection is behind the origin of the ray, thus is invalid. Having a valid intersection with the plane of the triangle, we check whether this point is inside the triangle. This is a containment problem, which is discussed in Subsection 22.4.1.

Intersection calculation for an AABB

The surface of an AABB, that is an axis aligned block, can be subdivided to 6 rectangular faces, or alternatively to 12 triangles, thus its intersection can be solved by the algorithms discussed in the previous subsections. However, realizing that in this special case the three coordinates can be handled separately, we can develop more efficient approaches. In fact, an AABB is the intersection of an x-stratum defined by inequality $x_{min} \leq x \leq x_{max}$, a y-stratum defined by $y_{min} \leq y \leq y_{max}$ and a z-stratum of inequality $z_{min} \leq z \leq z_{max}$. For example, the ray parameters of the intersections with the x-stratum are:

$$t_x^1 = \frac{x_{min} - s_x}{v_x}, \qquad t_x^2 = \frac{x_{max} - s_x}{v_x} .$$

The smaller of the two parameter values corresponds to the entry at the stratum, while the greater to the exit. Let us denote the ray parameter of the entry by t_{in}, and the ray parameter of the exit by t_{out}. The ray is inside the x-stratum while the ray parameter is in $[t_{in}, t_{out}]$. Repeating the same calculation for the y and z-strata as well, three ray parameter intervals are obtained. The intersection of these intervals determine when the ray is inside the AABB. If parameter t_{out} obtained as the result of intersecting the strata is negative, then the AABB is behind the eye, thus no ray–AABB intersection is possible. If only t_{in} is negative, then the ray starts at an internal point of the AABB, and the first intersection is at t_{out}. Finally, if t_{in} is positive, then the ray enters the AABB from outside at parameter t_{in}.

The computation of the unnecessary intersection points can be reduced by applying the Cohen–Sutherland line clipping algorithm (subsection 22.4.3). First, the ray is replaced by a line segment where one endpoint is the origin of the ray, and the other endpoint is an arbitrary point on the ray which is farther from the origin than any object of the virtual world. Then this line segment is tried to be clipped by the AABB. If the Cohen–Sutherland algorithm reports that the line segment has no internal part, then the ray has no intersection with the AABB.

22.6.2. Speeding up the intersection calculation

A naive ray tracing algorithm tests each object for a ray to find the closest intersection. If there are N objects in the space, the running time of the algorithm is $\Theta(N)$ both in the average and in the worst case. The storage requirement is also linear in terms of the number of objects.

The method would be speeded up if we could exclude certain objects from the intersection test without testing them one by one. The reasons of such exclusion include that these objects are "behind" the ray or "not in the direction of the ray".

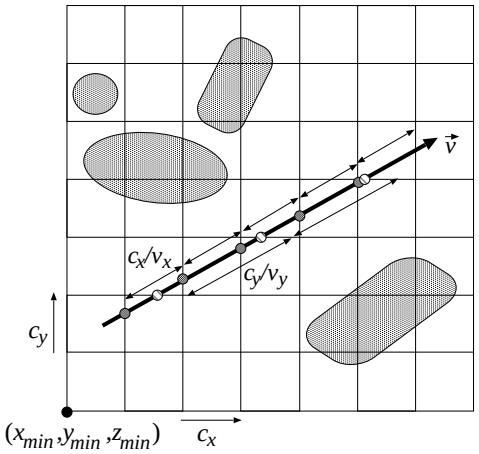


Figure 22.31. Partitioning the virtual world by a uniform grid. The intersections of the ray and the coordinate planes of the grid are at regular distances c_x/v_x,c_y/v_y, and c_z/v_z, respectively.

Additionally, the speed is also expected to improve if we can terminate the search having found an intersection supposing that even if other intersections exist, they are surely farther than the just found intersection point. To make such decisions safely, we need to know the arrangement of objects in the virtual world. This information is gathered during the pre-processing phase. Of course, pre-processing has its own computational cost, which is worth spending if we have to trace a lot of rays.

Bounding volumes

One of the simplest ray tracing acceleration technique uses ***bounding volumes***. The bounding volume is a shape of simple geometry, typically a sphere or an AABB, which completely contains a complex object. When a ray is traced, first the bounding volume is tried to be intersected. If there is no intersection with the bounding volume, then neither can the contained object be intersected, thus the computation time of the ray intersection with the complex object is saved. The bounding volume should be selected in a way that the ray intersection is computationally cheap, and it is a tight container of the complex object.

The application of bounding volumes does not alter the linear time complexity of the naive ray tracing. However, it can increase the speed by a scalar factor.

On the other hand, bounding volumes can also be organized in a hierarchy putting bounding volumes inside bigger bounding volumes recursively. In this case the ray tracing algorithm traverses this hierarchy, which is possible in sub-linear time.

Space subdivision with uniform grids

Let us find the AABB of the complete virtual world and subdivide it by an axis aligned uniform grid of cell sizes (c_x, c_y, c_z) (Figure 22.31).

In the preprocessing phase, for each cell we identify those objects that are at least

partially contained by the cell. The test of an object against a cell can be performed using a clipping algorithm (subsection 22.4.3), or simply checking whether the cell and the AABB of the object overlap.

UNIFORM-GRID-CONSTRUCTION()

```
1  Compute the minimum corner of the AABB (x_min, y_min, z_min)
   and cell sizes (c_x, c_y, c_z)
2  for each cell c
3      do object list of cell c ← empty
4          for each object o                ▷ Register objects overlapping with this cell.
5              do if cell c and the AABB of object o overlap
6                     then add object o to object list of cell c
```

During ray tracing, cells intersected by the ray are visited in the order of their distance from the ray origin. When a cell is processed, only those objects need to be tested for intersection which overlap with this cell, that is, which are registered in this cell. On the other hand, if an intersection is found in the cell, then intersections belonging to other cells cannot be closer to the ray origin than the found intersection. Thus the cell marching can be terminated. Note that when an object registered in a cell is intersected by the ray, we should also check whether the intersection point is also in this cell.

We might meet an object again in other cells. The number of ray–surface intersection can be reduced if the results of ray–surface intersections are stored with the objects and are reused when needed again.

As long as no ray–surface intersection is found, the algorithm traverses those cells which are intersected by the ray. Indices X, Y, Z of the first cell are computed from ray origin $\vec{s}$, minimum corner $(x_{min}, y_{min}, z_{min})$ of the grid, and sizes (c_x, c_y, c_z) of the cells:

UNIFORM-GRID-ENCLOSING-CELL($\vec{s}$)

```
1  X ← INTEGER((s_x − x_min)/c_x)
2  Y ← INTEGER((s_y − y_min)/c_y)
3  Z ← INTEGER((s_z − z_min)/c_z)
4  return X, Y, Z
```

The presented algorithm assumes that the origin of the ray is inside the subspace covered by the grid. Should this condition not be met, then the intersection of the ray and the scene AABB is computed, and the ray origin is moved to this point.

The initial values of ray parameters t_x, t_y, t_z are computed as the intersection of the ray and the coordinate planes by the UNIFORM-GRID-RAY-PARAMETER-INITIALIZATION algorithm:

UNIFORM-GRID-RAY-PARAMETER-INITIALIZATION($\vec{s}, \vec{v}, X, Y, Z$)

```
1  if v_x > 0
2     then t_x ← (x_min + (X + 1) · c_x − s_x)/v_x
3     else if v_x < 0
4             then t_x ← (x_min + X · c_x − s_x)/v_x
5             else  t_x ← t_max                          ▷ The maximum distance.
6  if v_y > 0
7     then  t_y ← (y_min + (Y + 1) · c_y − s_y)/v_y
8     else if v_y < 0
9             then t_y ← (y_min + Y · c_y − s_y)/v_y
10            else  t_y ← t_max
11 if v_z > 0
12    then  t_z ← (z_min + (Z + 1) · c_z − s_z)/v_z
13    else if v_z < 0
14            then t_z ← (z_min + Z · c_z − s_z)/v_z
15            else  t_z ← t_max
16 return t_x, t_y, t_z
```

The next cell of the sequence of the visited cells is determined by the ***3D line drawing algorithm (3DDDA algorithm)***. This algorithm exploits the fact that the ray parameters of the intersection points with planes perpendicular to axis x (and similarly to axes y and z) are regularly placed at distance c_x/v_x (c_y/v_y, and c_z/v_z, respectively), thus the ray parameter of the next intersection can be obtained with a single addition (Figure 22.31). Ray parameters t_x, t_y, and t_z are stored in global variables, and are incremented by constant values. The smallest from the three ray parameters of the coordinate planes identifies the next intersection with the cell.

The following algorithm computes indices X, Y, Z of the next intersected cell, and updates ray parameters t_x, t_y, t_z:

UNIFORM-GRID-NEXT-CELL(X, Y, Z, t_x, t_y, t_z)

```
1  if t_x = min(t_x, t_y, t_z) ▷ Next intersection is on the plane perpendicular to axis x.
2     then X ← X + sgn(v_x)                          ▷ Function sgn(x) returns the sign.
3          t_x ← t_x + c_x/|v_x|
4  else if t_y = min(t_x, t_y, t_z)
                          ▷ Next intersection is on the plane perpendicular to axis y.
5          then Y ← Y + sgn(v_y)
6               t_y ← t_y + c_y/|v_y|
7  else if t_z = min(t_x, t_y, t_z)
                          ▷ Next intersection is on the plane perpendicular to axis z.
8          then Z ← Z + sgn(v_z)
9               t_z ← t_z + c_z/|v_z|
```

To summarize, a complete ray tracing algorithm is presented, which exploits the uniform grid generated during preprocessing and computes the ray-surface intersec-

tion closest to the ray origin. The minimum of ray parameters (t_x, t_y, t_z) assigned to the coordinate planes, i.e. variable t_{out}, determines the distance as far as the ray is inside the cell. This parameter is used to decide whether or not a ray-surface intersection is really inside the cell.

RAY-FIRST-INTERSECTION-WITH-UNIFORM-GRID$(\vec{s}, \vec{v})$

```
1  (X, Y, Z) ← UNIFORM-GRID-ENCLOSING-CELL(s⃗)
2  (t_x, t_y, t_z) ← UNIFORM-GRID-RAY-PARAMETER-INITIALIZATION(s⃗, v⃗, X, Y, Z)
3  while X, Y, Z are inside the grid
4        do t_out ← min(t_x, t_y, t_z)              ▷ Here is the exit from the cell.
5           t ← t_out                               ▷ Initialization: no intersection yet.
6           for each object o registered in cell (X, Y, Z)
7              do t_o ←RAY-SURFACE-INTERSECTION(s⃗, v⃗, o)
                                                    ▷ Negative: no intersection.
8                 if 0 ≤ t_o < t                    ▷ Is the new intersection closer?
9                 then t ← t_o
                                ▷ The ray parameter of the closest intersection so far.
10                   o_visible ← o                  ▷ The first intersected object.
11          if t < t_out                            ▷ Was intersection in the cell?
12             then x⃗ ← s⃗ + v⃗ · t                  ▷ The position of the intersection.
13                  return t, x⃗, o_visible          ▷ Termination.
14          UNIFORM-GRID-NEXT-CELL(X, Y, Z, t_x, t_y, t_z)   ▷ 3DDDA.
15 return "no intersection"
```

Time and storage complexity of the uniform grid algorithm

The preprocessing phase of the uniform grid algorithm tests each object with each cell, thus runs in $\Theta(N \cdot C)$ time where N and C are the numbers of objects and cells, respectively. In practice, the resolution of the grid is set to make C proportional to N since in this case, the average number of objects per cell becomes independent of the total number of objects. Such resolution makes the preprocessing time quadratic, that is $\Theta(N^2)$. We note that sorting objects before testing them against cells may reduce this complexity, but this optimization is not crucial since not the preprocessing but the ray tracing time is critical. Since in the worst case all objects may overlap with each cell, the storage space is also in $O(N^2)$.

The ray tracing time can be expressed by the following equation:

$$T = T_o + N_I \cdot T_I + N_S \cdot T_S \ , \tag{22.26}$$

where T_o is the time needed to identify the cell containing the origin of the ray, N_I is the number of ray–surface intersection tests until the first intersection is found, T_I is the time required by a single ray–surface intersection test, N_S is the number of visited cells, and T_S is the time needed to step onto the next cell.

To find the first cell, the coordinates of the ray origin should be divided by the cell sizes, and the cell indices are obtained by rounding the results. This step thus runs in constant time. A single ray–surface intersection test also requires constant time. The next cell is determined by the 3DDDA algorithm in constant time as well.

Thus the complexity of the algorithm depends only on the number of intersection tests and the number of the visited cells.

Considering a worst case scenario, a cell may contain all objects, requiring $O(N)$ intersection test with N objects. In the worst case the ray tracing has linear complexity. This means that the uniform grid algorithm needs quadratic preprocessing time and storage, but solves the ray tracing problem still in linear time as the naive algorithm, which is quite disappointing. However, uniform grids are still worth using since worst case scenarios are very unlikely. The fact is that classic complexity measures describing the worst case characteristics are not appropriate to compare the naive algorithm and the uniform grid based ray tracing. For a reasonable comparison, the probabilistic analysis of the algorithms is needed.

Probabilistic model of the virtual world

To carry out the average case analysis, the scene model, i.e. the probability distribution of the possible virtual world models must be known. In practical situations, this probability distribution is not available, therefore it must be estimated. If the model of the virtual world were too complicated, we would not be able to analytically determine the average, i.e. the expected running time of the ray tracing algorithm. A simple, but also justifiable model is the following: *Objects are spheres of the same radius r, and sphere centres are uniformly distributed in space.*

Since we are interested in the asymptotic behavior when the number of objects is really high, uniform distribution in a finite space would not be feasible. On the other hand, the boundary of the space would pose problems. Thus, instead of dealing with a finite object space, the space should also be expanded as the number of objects grows to sustain constant average spatial object density. This is a classical method in probability theory, and its known result is the Poisson point process.

Definition 22.16 *A **Poisson point process** $N(A)$ counts the number of points in subset A of space in a way that*

- *$N(A)$ is a Poisson distribution of parameter $\rho V(A)$, where ρ is a positive constant called "intensity" and $V(A)$ is the volume of A, thus the probability that A contains exactly k points is*

$$\Pr\{N(A) = k\} = \frac{(\rho V(A))^k}{k!} \cdot e^{-\rho V(A)} ,$$

  and the expected number of points in volume $V(A)$ is $\rho V(A)$;
- *for disjoint $A_1, A_2, \ldots, A_n$ sets random variables $N(A_1), N(A_2), \ldots, N(A_n)$ are independent.*

Using the Poisson point process, the probabilistic model of the virtual world is:

1. The object space consists of spheres of the same radius r.
2. The sphere centres are the realizations of a Poisson point process of intensity ρ.

Having constructed a probabilistic virtual world model, we can start the analysis of the candidate algorithms assuming that the rays are uniformly distributed in space.

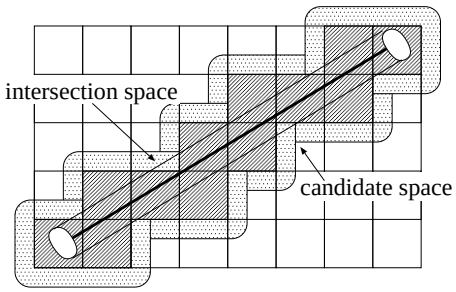


Figure 22.32. Encapsulation of the intersection space by the cells of the data structure in a uniform subdivision scheme. The intersection space is a cylinder of radius r. The candidate space is the union of those spheres that may overlap a cell intersected by the ray.

Calculation of the expected number of intersections

Looking at Figure 22.32 we can see a ray that passes through certain cells of the space partitioning data structure. The collection of those sphere centres where the sphere would have an intersection with a cell is called the ***candidate space*** associated with this cell.

Only those spheres of radius r can have intersection with the ray whose centres are in a cylinder of radius r around the ray. This cylinder is called the ***intersection space*** (Figure 22.32). More precisely, the intersection space also includes two half spheres at the bottom and at the top of the cylinder, but these will be ignored.

As the ray tracing algorithm traverses the data structure, it examines each cell that is intersected by the ray. If the cell is empty, then the algorithm does nothing. If the cell is not empty, then it contains, at least partially, a sphere which is tried to be intersected. This intersection succeeds if the centre of the sphere is inside the intersection space and fails if it is outside.

The algorithm should try to intersect objects that are in the candidate space, but this intersection will be successful only if the object is also contained by the intersection space. The probability of the success s is the ratio of the projected areas of the intersection space and the candidate space associated with this cell.

>From the probability of the successful intersection in a non-empty cell, the probability that the intersection is found in the first, second, etc. cells can also be computed. Assuming statistical independence, the probabilities that the first, second, third, etc. intersection is the first successful intersection are s, $(1-s)s$, $(1-s)^2 s$, etc., respectively. This is a geometric distribution with expected value $1/s$. Consequently, the expected number of the ray–object intersection tests is:

$$E[N_I] = \frac{1}{s} \,. \tag{22.27}$$

If the ray is parallel to one of the sides, then the projected size of the candidate space is $c^2+4cr+r^2\pi$ where c is the edge size of a cell and r is the radius of the spheres. The other extreme case happens when the ray is parallel to the diagonal of the cubic cell, where the projection is a rounded hexagon having area $\sqrt{3}c^2 + 6cr + r^2\pi$. The

success probability is then:

$$\frac{r^2\pi}{\sqrt{3}c^2 + 6cr + r^2\pi} \leq s \leq \frac{r^2\pi}{c^2 + 4cr + r^2\pi} .$$

According to equation (22.27), the average number of intersection calculations is the reciprocal of this probability:

$$\frac{1}{\pi}\left(\frac{c}{r}\right)^2 + \frac{4}{\pi}\frac{c}{r} + 1 \leq \mathrm{E}\left[N_I\right] \leq \frac{\sqrt{3}}{\pi}\left(\frac{c}{r}\right)^2 + \frac{6}{\pi}\frac{c}{r} + 1 . \tag{22.28}$$

Note that if the size of the cell is equal to the diameter of the sphere ($c = 2r$), then

$$3.54 < \mathrm{E}\left[N_I\right] < 7.03 .$$

This result has been obtained assuming that the number of objects converges to infinity. The expected number of intersection tests, however, remains finite and relatively small.

Calculation of the expected number of cell steps

In the following analysis the conditional expected value theorem will be used. An appropriate condition is the length of the ray segment between its origin and the closest intersection. Using its probability density $p_{t^*}(t)$ as a condition, the expected number of visited cells N_S can be written in the following form:

$$E[N_S] = \int_0^\infty E[N_S | t^* = t] \cdot p_{t^*}(t)\ dt ,$$

where t^* is the length of the ray and p_{t^*} is its probability density.

Since the intersection space is a cylinder if we ignore the half spheres around the beginning and the end, its total volume is $r^2\pi t$. Thus the probability that intersection occurs before t is:

$$\Pr\left\{t^* < t\right\} = 1 - e^{-\rho r^2 \pi t} .$$

Note that this function is the cumulative probability distribution function of t^*. The probability density can be computed as its derivative, thus we obtain:

$$p_{t^*}(t) = \rho r^2 \pi \cdot e^{-\rho r^2 \pi t} .$$

The expected length of the ray is then:

$$E[t^*] = \int_0^\infty t \cdot \rho r^2 \pi \cdot e^{-\rho r^2 \pi t}\ dt = \frac{1}{\rho r^2 \pi} . \tag{22.29}$$

In order to simplify the analysis, we shall assume that the ray is parallel to one of the coordinate axes. Since all cells have the same edge size c, the number of cells intersected by a ray of length t can be estimated as $E[N_S | t^* = t] \approx t/c + 1$. This estimation is quite accurate. If the the ray is parallel to one of the coordinate axes,

then the error is at most 1. In other cases the real value can be at most $\sqrt{3}$ times the given estimation. The estimated expected number of visited cells is then:

$$\mathrm{E}\left[N_S\right] \approx \int_0^\infty \left(\frac{t}{c}+1\right) \cdot \rho r^2 \pi \cdot e^{-\rho r^2 \pi t}\, dt = \frac{1}{c\rho r^2 \pi} + 1\ . \tag{22.30}$$

For example, if the cell size is similar to the object size ($c = 2r$), and the expected number of sphere centres in a cell is 0.1, then $\mathrm{E}\left[N_S\right] \approx 14$. Note that the expected number of visited cells is also constant even for infinite number of objects.

Expected running time and storage space

We concluded that the expected numbers of required intersection tests and visited cells are asymptotically constant, thus the expected time complexity of the uniform grid based ray tracing algorithm is constant after quadratic preprocessing time. The value of the running time can be controlled by cell size c according to equations (22.28) and (22.30). Smaller cell sizes reduce the average number of intersection tests, but increase the number of visited cells.

According to the probabilistic model, the average number of objects overlapping with a cell is also constant, thus the storage is proportional to the number of cells. Since the number of cells is set proportional to the number of objects, the expected storage complexity is also linear unlike the quadratic worst-case complexity.

The expected constant running time means that asymptotically the running time is independent of the number of objects, which explains the popularity of the uniform grid based ray tracing algorithm, and also the popularity of the algorithms presented in the next subsections.

Octree

Uniform grids require many unnecessary cell steps. For example, the empty spaces are not worth partitioning into cells, and two cells are worth separating only if they contain different objects. Adaptive space partitioning schemes are based on these recognitions. The space can be partitioned adaptively following a recursive approach. This results in a hierarchical data structure, which is usually a tree. The type of this tree is the base of the classification of such algorithms.

The adaptive scheme discussed in this subsection uses an octal tree (octree for short), where non-empty nodes have 8 children. An octree is constructed by the following algorithm:

- For each object, an AABB is found, and object AABBs are enclosed by a scene AABB. The scene AABB is the cell corresponding to the root of the octree.
- If the number of objects overlapping with the current cell exceeds a predefined threshold, then the cell is subdivided to 8 cells of the same size by halving the original cell along each coordinate axis. The 8 new cells are the children of the node corresponding to the original cell. The algorithm is recursively repeated for the child cells.
- The recursive tree building procedure terminates if the depth of the tree becomes too big, or when the number of objects overlapping with a cell is smaller than

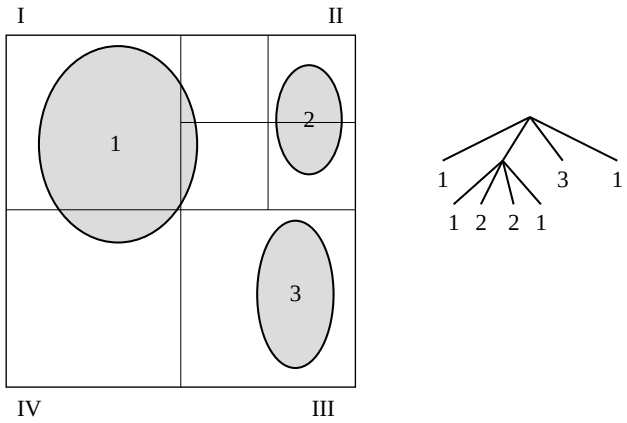


Figure 22.33. A quadtree partitioning the plane, whose three-dimensional version is the octree. The tree is constructed by halving the cells along all coordinate axes until a cell contains "just a few" objects, or the cell sizes gets smaller than a threshold. Objects are registered in the leaves of the tree.

the threshold.

The result of this construction is an ***octree*** (Figure 22.33). Overlapping objects are registered in the leaves of this tree.

When a ray is traced, those leaves of the tree should be traversed which are intersected by the ray, and ray–surface intersection test should be executed for objects registered in these leaves:

RAY-FIRST-INTERSECTION-WITH-OCTREE($\vec{s}, \vec{v}$)

```
1  q⃗ ← intersection of the ray and the scene AABB
2  while q⃗ is inside of the scene AABB                    ▷ Traversal of the tree.
3        cell ← OCTREE-CELL-SEARCH(octree root, q⃗)
4        t_out ← ray parameter of the intersection of the cell and the ray
5        t ← t_out                          ▷ Initialization: no ray–surface intersection yet.
6        for each object o registered in cell
7             do t_o ←RAY-SURFACE-INTERSECTION(s⃗, v⃗)
                                                ▷ Negative if no intersection exists.
8                if 0 ≤ t_o < t                          ▷ Is the new intersection closer?
9                   then t ← t_o ▷ Ray parameter of the closest intersection so far.
10                       o_visible ← o                  ▷ First intersected object so far.
11       if t < t_out                                   ▷ Has been intersection at all ?
12          then x⃗ ← s⃗ + v⃗ · t                          ▷ Position of the intersection.
13               return t, x⃗, o_visible
14       q⃗ ← s⃗ + v⃗ · (t_out + ε)                       ▷ A point in the next cell.
15 return "no intersection"
```

The identification of the next cell intersected by the ray is more complicated for

octrees than for uniform grids. The OCTREE-CELL-SEARCH algorithm determines that leaf cell which contains a given point. At each level of the tree, the coordinates of the point are compared to the coordinates of the centre of the cell. The results of these comparisons determine which child contains the point. Repeating this test recursively, we arrive at a leaf sooner or later.

In order to identify the next cell intersected by the ray, the intersection point of the ray and the current cell is computed. Then, ray parameter t_{out} of this intersection point is increased "a little" (this little value is denoted by ε in algorithm RAY-FIRST-INTERSECTION-WITH-OCTREE). The increased ray parameter is substituted into the ray equation, resulting in point $\vec{q}$ that is already in the next cell. The cell containing this point can be identified with OCTREE-CELL-SEARCH.

Cells of the octree may be larger than the allowed minimal cell, therefore the octree algorithm requires less number of cell steps than the uniform grid algorithm working on the minimal cells. However, larger cells reduce the probability of the successful intersection tests since in a large cell it is less likely that a random ray intersecting the cell also intersects a contained object. Smaller successful intersection probability, on the other hand, results in greater expected number of intersection tests, which affects the performance negatively. It also means that non-empty octree cells are worth subdividing until the minimum cell size is reached even if the cell contains just a single object. Following this strategy, the size of the non-empty cells are similar, thus the results of the complexity analysis made for the uniform grid remain to be applicable to the octree as well. Since the probability of the successful intersection depends on the size of the non-empty cells, the expected number of needed intersection tests is still given by inequality (22.28). It also means that when the minimal cell size of an octree equals to the cell size of a uniform grid, then the expected number of intersection tests is equal in the two algorithms.

The advantage of the ocree is the ability to skip empty spaces, which reduces the number of cell steps. Its disadvantage is, however, that the time of the next cell identification is not constant. This identification requires the traversal of the tree. If the tree construction is terminated when a cell contains small number of objects, then the number of leaf cells is proportional to the number of objects. The depth of the tree is in $O(\lg N)$, so is the time needed to step onto the next cell.

kd-tree

An octree adapts to the distribution of the objects. However, the partitioning strategy of octrees always halves the cells without taking into account where the objects are, thus the adaptation is not perfect. Let us consider a partitioning scheme which splits a cell into two cells to make the tree balanced. Such method builds a binary tree which is called ***binary space partitioning tree***, abbreviated as ***BSP-tree***. If the ***separating plane*** is always perpendicular to one of the coordinate axes, then the tree is called ***kd-tree***.

The separating plane of a kd-tree node can be placed in many different ways:

- the ***spatial median method*** halves the cell into two congruent cells.
- the ***object median method*** finds the separating plane to have the same number of objects in the two child cells.

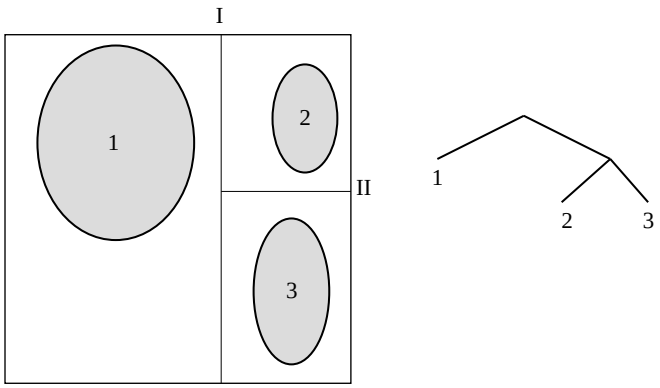


Figure 22.34. A kd-tree. A cell containing "many" objects are recursively subdivided to two cells with a plane that is perpendicular to one of the coordinate axes.

- the ***cost driven method*** estimates the average computation time needed when a cell is processed during ray tracing, and minimizes this value by placing the separating plane. An appropriate cost model suggests to separate the cell to make the probabilities of the ray–surface intersection of the two cells similar.

The probability of the ray–surface intersection can be computed using a fundamental theorem of the ***integral geometry***:

Theorem 22.17 *If convex solid A contains another convex solid B, then the probability that a uniformly distributed line intersects solid B provided that the line intersected A equals to the ratio of the surface areas of objects B and A.*

According to this theorem the cost driven method finds the separating plane to equalize the surface areas in the two children.

Let us now present a general kd-tree construction algorithm. Parameter *cell* identifies the current cell, *depth* is the current depth of recursion, and *coordinate* stores the orientation of the current separating plane. A *cell* is associated with its two children (*cell.right* and *cell.left*), and its left-lower-closer and right-upper-farther corners (*cell.min* and *cell.max*). Cells also store the list of those objects which overlap with the cell. The orientation of the separation plane is determined by a round-robin scheme implemented by function ROUND-ROBIN providing a sequence like $(x, y, z, x, y, z, x, \ldots)$. When the following recursive algorithm is called first, it gets the scene AABB in variable *cell* and the value of variable *depth* is zero:

KD-TREE-CONSTRUCTION(*cell*, *depth*, *coordinate*)

```
1  if the number of objects overlapping with cell is small or depth is large
2     then return
3  AABB of cell.left and AABB of cell.right ← AABB of cell
```

```
4  if coordinate = x
5     then cell.right.min.x ← x perpendicular separating plane of cell
6          cell.left.max.x ← x perpendicular separating plane of cell
7     else if coordinate = y
8             then cell.right.min.y ← y perpendicular separating plane of cell
9                  cell.left.max.y ← y perpendicular separating plane of cell
10    else if coordinate = z
11            then cell.right.min.z ← z perpendicular separating plane of cell
12                 cell.left.max.z ← z perpendicular separating plane of cell
13 for each object o of cell
14     do if object o is in the AABB of cell.left
15           then assign object o to the list of cell.left
16        if object o is in the AABB of cell.right
17           then assign object o to the list of cell.right
18 KD-TREE-CONSTRUCTION(cell.left, depth + 1, ROUND-ROBIN(coordinate))
19 KD-TREE-CONSTRUCTION(cell.right, depth + 1, ROUND-ROBIN(coordinate))
```

Now we discuss an algorithm that traverses the constructed kd-tree and finds the visible object. First we have to test whether the origin of the ray is inside the scene AABB. If it is not, the intersection of the ray and the scene AABB is computed, and the origin of the ray is moved there. The identification of the cell containing the ray origin requires the traversal of the tree. During the traversal the coordinates of the point are compared to the coordinates of the separating plane. This comparison determines which child should be processed recursively until a leaf node is reached. If the leaf cell is not empty, then objects overlapping with the cell are intersected with the ray, and the intersection closest to the origin is retained. The closest intersection is tested to see whether or not it is inside the cell (since an object may overlap more than one cells, it can also happen that the intersection is in another cell). If the intersection is in the current cell, then the needed intersection has been found, and the algorithm can be terminated. If the cell is empty, or no intersection is found in the cell, then the algorithm should proceed with the next cell. To identify the next cell, the ray is intersected with the current cell identifying the ray parameter of the exit point. Then the ray parameter is increased "a little" to make sure that the increased ray parameter corresponds to a point in the next cell. The algorithm keeps repeating these steps as it processes the cells of the tree.

This method has the disadvantage that the cell search always starts at the root, which results in the repetitive traversals of the same nodes of the tree.

This disadvantage can be eliminated by putting the cells to be visited into a stack, and backtracking only to the point where a new branch should be followed. When the ray arrives at a node having two children, the algorithm decides the order of processing the two child nodes. Child nodes are classified as "near" and "far" depending on whether or not the child cell is on the same side of the separating plane as the origin of the ray. If the ray intersects only the "near" child, then the algorithm processes only that subtree which originates at this child. If the ray intersects both

children, then the algorithm pushes the "far" node onto the stack and starts processing the "near" node. If no intersection exists in the "near" node, then the stack is popped to obtain the next node to be processed.

The notations of the ray tracing algorithm based on kd-tree traversal are shown by Figure 22.35. The algorithm is the following:

```
RAY-FIRST-INTERSECTION-WITH-KD-TREE(root, s, v)
 1  (t_in, t_out) ← RAY-AABB-INTERSECTION(s, v, root)
                                          ▷ Intersection with the scene AABB.
 2  if no intersection
 3     then return "no intersection"
 4  PUSH(root, t_in, t_out)
 5  while the stack is not empty                              ▷ Visit all nodes.
 6        do POP(cell, t_in, t_out)
 7           while cell is not a leaf
 8                 do coordinate ← orientation of the separating plane of the cell
 9                    d ← cell.right.min[coordinate] − s[coordinate]
10                    t ← d/v[coordinate] ▷ Ray parameter of the separating plane.
11                    if d > 0      ▷ Is s on the left side of the separating plane?
12                       then (near, far) ← (cell.left, cell.right)          ▷ Left.
13                       else (near, far) ← (cell.right, cell.left)          ▷ Right.
14                    if t > t_out or t < 0
15                       then cell ← near ▷ The ray intersects only the near cell.
16                       else if t < t_in
17                                then cell ← far ▷ The ray intersects only the far cell.
18                                else PUSH(far, t, t_out) ▷ The ray intersects both cells.
19                                     cell ← near       ▷ First near is intersected.
20                                     t_out ← t ▷ The ray exists at t from the near cell.
                                                          ▷ If the current cell is a leaf.
21           t ← t_out                          ▷ Maximum ray parameter in this cell.
22           for each object o of cell
23                do t_o ←RAY-SURFACE-INTERSECTION(s, v)
                                          ▷ Negative if no intersection exists.
24                   if t_in ≤ t_o < t ▷ Is the new intersection closer to the ray origin?
25                   then t ← t_o
                                ▷ The ray parameter of the closest intersection so far.
26                        o_visible ← o
                                     ▷ The object intersected closest to the ray origin.
27           if t < t_out                       ▷ Has been intersection at all in the cell?
28              then x ← s + v · t                                ▷ The intersection point.
29                   return t, x, o_visible                ▷ Intersection has been found.
30  return "no intersection"                                    ▷ No intersection.
```

Similarly to the octree algorithm, the likelihood of successful intersections can be increased by continuing the tree building process until all empty spaces are cut

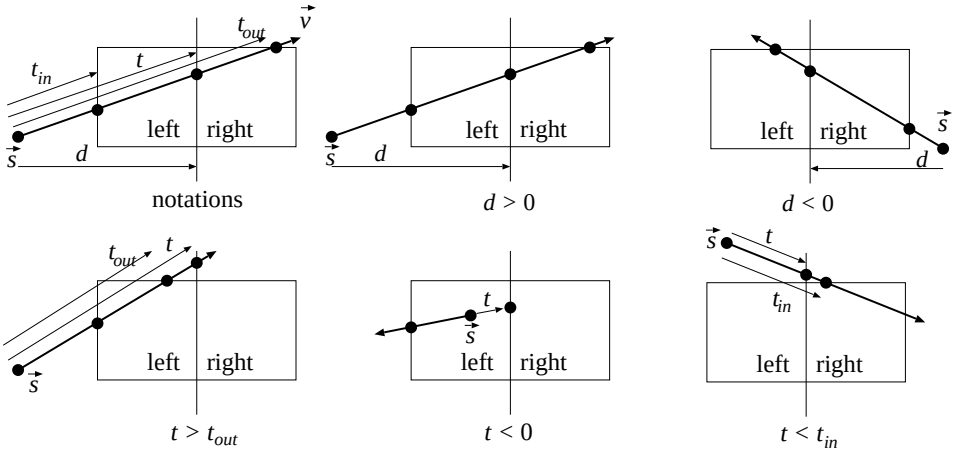


Figure 22.35. Notations and cases of algorithm RAY-FIRST-INTERSECTION-WITH-KD-TREE. t_{in}, t_{out}, and t are the ray parameters of the entry, exit, and the separating plane, respectively. d is the signed distance between the ray origin and the separating plane.

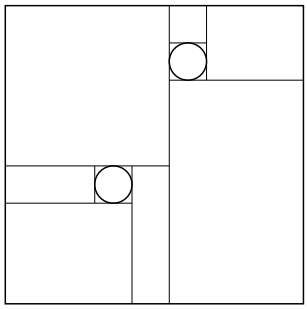


Figure 22.36. Kd-tree based space partitioning with empty space cutting.

(Figure 22.36).

Our probabilistic world model contains spheres of same radius r, thus the non-empty cells are cubes of edge size $c = 2r$. Unlike in uniform grids or octrees, the separating planes of kd-trees are not independent of the objects. Kd-tree splitting planes are rather tangents of the objects. This means that we do not have to be concerned with partially overlapping spheres since a sphere is completely contained by a cell in a kd-tree. The probability of the successful intersection is obtained applying Theorem 22.17. In the current case, the containing convex solid is a cube of edge size $2r$, the contained solid is a sphere of radius r, thus the intersection probability is:

$$s = \frac{4r^2\pi}{6a^2} = \frac{\pi}{6} .$$

The expected number of intersection tests is then:

$$\mathrm{E}[N_I] = \frac{6}{\pi} \approx 1.91 .$$

We can conclude that the kd-tree algorithm requires the smallest number of ray-surface intersection tests according to the probabilistic model.

Exercises

22.6-1 Prove that the expected number of intersection tests is constant in all those ray tracing algorithms which process objects in the order of their distance from the ray origin.

22.6-2 Propose a ray intersection algorithm for subdivision surfaces.

22.6-3 Develop a ray intersection method for B-spline surfaces.

22.6-4 Develop a ray intersection algorithm for CSG models assuming that the ray–primitive intersection tests are already available.

22.6-5 Propose a ray intersection algorithm for transformed objects assuming that the algorithm computing the intersection with the non-transformed objects is available (hints: transform the ray).

22.7. Incremental rendering

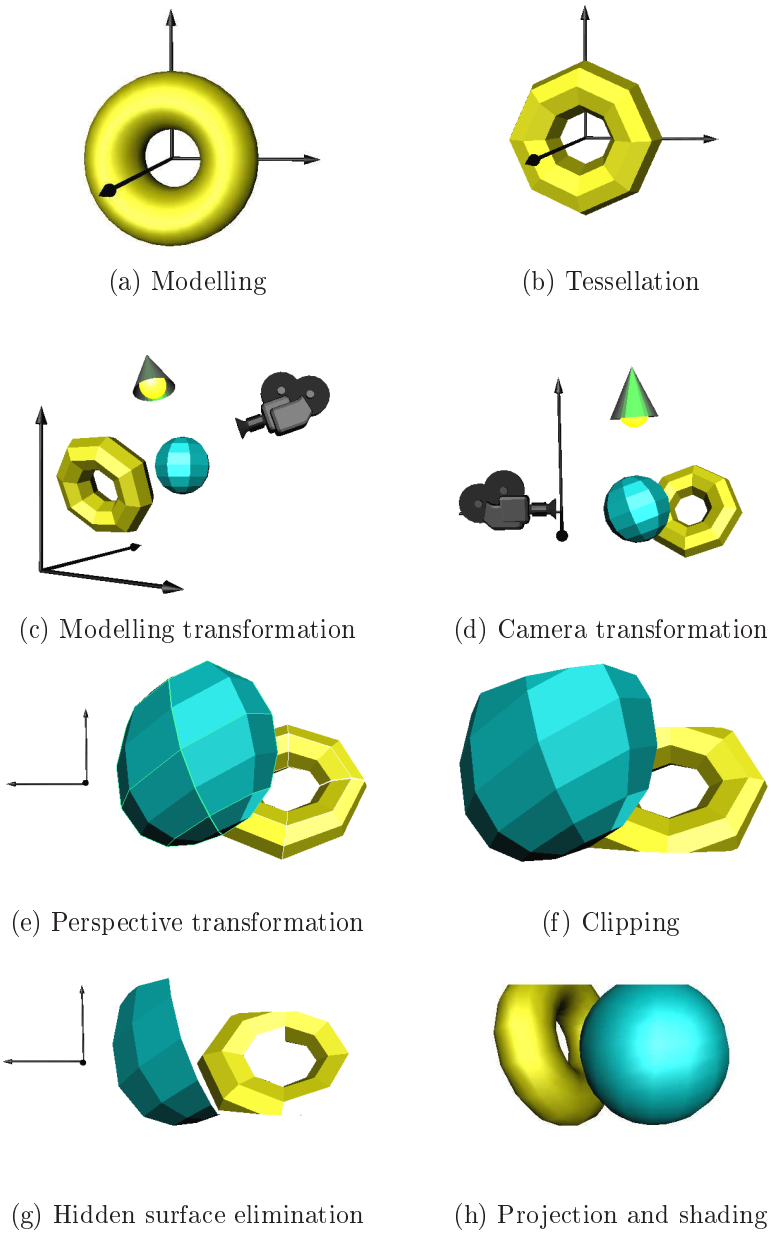
Rendering requires the identification of those surface points that are visible through the pixels of the virtual camera. Ray tracing solves this visibility problem for each pixel independently, thus it does not reuse visibility information gathered at other pixels. The algorithms of this section, however, exploit such information using the following simple techniques:

1. They simultaneously attack the visibility problem for all pixels, and handle larger parts of the scene at once.
2. Where feasible, they exploit the ***incremental principle*** which is based on the recognition that the visibility problem becomes simpler to solve if the solution at the neighbouring pixel is taken into account.
3. They solve each task in that coordinate system which makes the solution easier. The scene is transformed from one coordinate system to the other by homogeneous linear transformations.
4. They minimize unnecessary computations, therefore remove those objects by *clipping* in an early stage of rendering which cannot be projected onto the window of the camera.

Homogeneous linear transformations and clipping may change the type of the surface except for points, line segments and polygons [4]. Therefore, before rendering is started, each shape is approximated by points, line segments, and meshes (Subsection 22.3).

Steps of incremental rendering are shown in Figure 22.37. Objects are defined in their reference state, approximated by meshes, and are transformed to the virtual world. The time dependence of this transformation is responsible for object animation. The image is taken from the camera about the virtual world, which requires

[4] Although Bézier and B-Spline curves and surfaces are invariant to affine transformations, and NURBS is invariant even to homogeneous linear transformations, but clipping changes these object types as well.



(a) Modelling (b) Tessellation

(c) Modelling transformation (d) Camera transformation

(e) Perspective transformation (f) Clipping

(g) Hidden surface elimination (h) Projection and shading

Figure 22.37. Steps of incremental rendering. **(a)** Modelling defines objects in their reference state. **(b)** Shapes are tessellated to prepare for further processing. **(c)** Modelling transformation places the object in the world coordinate system. **(d)** Camera transformation translates and rotates the scene to get the eye to be at the origin and to look parallel with axis $-z$. **(e)** Perspective transformation converts projection lines meeting at the origin to parallel lines, that is, it maps the eye position onto an ideal point. **(f)** Clipping removes those shapes and shape parts, which cannot be projected onto the window. **(g)** Hidden surface elimination removes those surface parts that are occluded by other shapes. **(h)** Finally, the visible polygons are projected and their projections are filled with their visible colours.

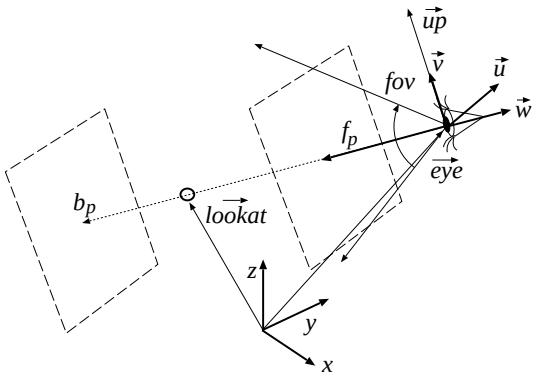


Figure 22.38. Parameters of the virtual camera: eye position $\vec{eye}$, target $\vec{lookat}$, and vertical direction $\vec{up}$, from which camera basis vectors $\vec{u}, \vec{v}, \vec{w}$ are obtained, front f_p and back b_p clipping planes, and vertical field of view *fov* (the horizontal field of view is computed from aspect ratio *aspect*).

the identification of those surface points that are visible from the camera, and their projection onto the window plane. The visibility and projection problems could be solved in the virtual world as happens in ray tracing, but this would require the intersection calculations of general lines and polygons. Visibility and projection algorithms can be simplified if the scene is transformed to a coordinate system, where the X, Y coordinates of a point equal to the coordinates of that pixel onto which this point is projected, and the Z coordinate can be used to decide which point is closer if more than one surfaces are projected onto the same pixel. Such coordinate system is called the ***screen coordinate system***. In screen coordinates the units of axes X and Y are equal to the pixel size. Since it is usually not worth computing the image on higher accuracy than the pixel size, coordinates X, Y are integers. Because of performance reasons, coordinate Z is also often integer. Screen coordinates are denoted by capital letters.

The transformation taking to the screen coordinate system is defined by a sequence of transformations, and the elements of this sequence are discussed separately. However, this transformation is executed as a single multiplication with a 4×4 transformation matrix obtained as the product of elementary transformation matrices.

22.7.1. Camera transformation

Rendering is expected to generate an image from a camera defined by ***eye position*** ($\vec{eye}$) (the focal point of the camera), looking target ($\vec{lookat}$) where the camera looks at, and by vertical direction $\vec{up}$ (Figure 22.38).

Camera parameter *fov* defines the vertical field of view, *aspect* is the ratio of the width and the height of the window, f_p and b_p are the distances of the front and back clipping planes from the eye, respectively. These clipping planes allow to remove those objects that are behind, too close to, or too far from the eye.

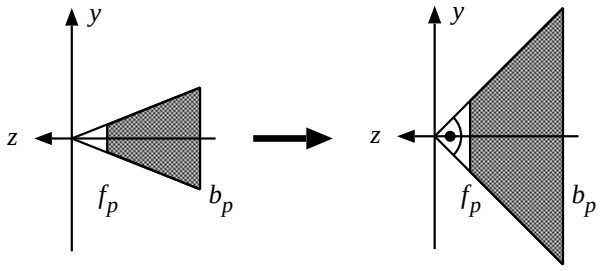


Figure 22.39. The normalizing transformation sets the field of view to 90 degrees.

We assign a coordinate system, i.e. three orthogonal unit basis vectors to the camera. Horizontal basis vector $\vec{u} = (u_x, u_y, u_z)$, vertical basis vector $\vec{v} = (v_x, v_y, v_z)$, and basis vector $\vec{w} = (w_x, w_y, w_z)$ pointing to the looking direction are obtained as follows:

$$\vec{w} = \frac{e\vec{y}e - l\vec{o}okat}{|e\vec{y}e - l\vec{o}okat|}, \qquad \vec{u} = \frac{\vec{up} \times \vec{w}}{|\vec{up} \times \vec{w}|}, \qquad \vec{v} = \vec{w} \times \vec{u} \; .$$

The ***camera transformation*** translates and rotates the space of the virtual world in order to get the camera to move to the origin, to look at direction axis $-z$, and to have vertical direction parallel to axis y, that is, this transformation maps unit vectors $\vec{u}, \vec{v}, \vec{w}$ to the basis vectors of the coordinate system. Transformation matrix $\mathbf{T}_{camera}$ can be expressed as the product of a matrix translating the eye to the origin and a matrix rotating basis vectors $\vec{u}, \vec{v}, \vec{w}$ of the camera to the basis vectors of the coordinate system:

$$[x', y', z', 1] = [x, y, z, 1] \cdot \mathbf{T}_{camera} = [x, y, z, 1] \cdot \mathbf{T}_{translation} \cdot \mathbf{T}_{rotation} \; , \qquad (22.31)$$

where

$$\mathbf{T}_{translation} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -eye_x & -eye_y & -eye_z & 1 \end{bmatrix}, \quad \mathbf{T}_{rotation} = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

Let us note that the columns of the rotation matrix are vectors $\vec{u}, \vec{v}, \vec{w}$. Since these vectors are orthogonal, it is easy to see that this rotation maps them to coordinate axes x, y, z. For example, the rotation of vector $\vec{u}$ is:

$$[u_x, u_y, u_z, 1] \cdot \mathbf{T}_{rotation} = [\vec{u} \cdot \vec{u}, \vec{u} \cdot \vec{v}, \vec{u} \cdot \vec{w}, 1] = [1, 0, 0, 1] \; .$$

22.7.2. Normalizing transformation

In the next step the viewing pyramid containing those points which can be projected onto the window is normalized making the field of view equal to 90 degrees (Figure 22.39).

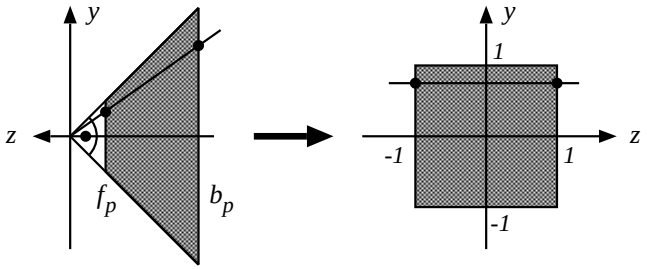


Figure 22.40. The perspective transformation maps the finite frustum of pyramid defined by the front and back clipping planes, and the edges of the window onto an axis aligned, origin centred cube of edge size 2.

Normalization is a simple scaling transformation:

$$\mathbf{T}_{norm} = \begin{bmatrix} 1/(\tan(fov/2) \cdot aspect) & 0 & 0 & 0 \\ 0 & 1/\tan(fov/2) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The main reason of this transformation is to simplify the formulae of the next transformation step, called perspective transformation.

22.7.3. Perspective transformation

The perspective transformation distorts the virtual world to allow the replacement of the perspective projection by parallel projection during rendering.

After the normalizing transformation, the potentially visible points are inside a symmetrical finite frustum of pyramid of 90 degree apex angle (Figure 22.39). The perspective transformation maps this frustum onto a cube, converting projection lines crossing the origin to lines parallel to axis z (Figure 22.40).

Perspective transformation is expected to map point to point, line to line, but to map the eye position to infinity. It means that perspective transformation cannot be a linear transformation of Cartesian coordinates. Fortunately, homogeneous linear transforms also map point to point, line to line, and are able to handle points at infinity with finite coordinates. Let us thus try to find the perspective transformation in the form of a homogeneous linear transformation defined by a 4×4 matrix:

$$\mathbf{T}_{persp} = \begin{bmatrix} t_{11} & t_{12} & t_{13} & t_{14} \\ t_{21} & t_{22} & t_{23} & t_{24} \\ t_{31} & t_{32} & t_{33} & t_{34} \\ t_{41} & t_{42} & t_{43} & t_{44} \end{bmatrix}.$$

Figure 22.40 shows a line (projection ray) and its transform. Let m_x and m_y be the x/z and the y/z slopes of the line, respectively. This line is defined by equation $[-m_x \cdot z, -m_y \cdot z, z]$ in the normalized camera space. The perspective transformation maps this line to a "horizontal" line crossing point $[m_x, m_y, 0]$ and being parallel

to axis z. Let us examine the intersection points of this line with the front and back clipping planes, that is, let us substitute $(-f_p)$ and $(-b_p)$ into parameter z of the line equation. The transformation should map these points to $[m_x, m_y, -1]$ and $[m_x, m_y, 1]$, respectively.

The perspective transformation of the point on the first clipping plane is:

$$[m_x \cdot f_p, m_y \cdot f_p, -f_p, 1] \cdot \mathbf{T}_{persp} = [m_x, m_y, -1, 1] \cdot \lambda \ ,$$

where λ is an arbitrary, non-zero scalar since the point defined by homogeneous coordinates does not change if the homogeneous coordinates are simultaneously multiplied by a non-zero scalar. Setting λ to f_p, we get:

$$[m_x \cdot f_p, m_y \cdot f_p, -f_p, 1] \cdot \mathbf{T}_{persp} = [m_x \cdot f_p, m_y \cdot f_p, -f_p, f_p] \ . \tag{22.32}$$

Note that the first coordinate of the transformed point equals to the first coordinate of the original point on the clipping plane for arbitrary m_x, m_y, and f_p values. This is possible only if the first column of matrix $\mathbf{T}_{persp}$ is $[1, 0, 0, 0]^T$. Using the same argument for the second coordinate, we can conclude that the second column of the matrix is $[0, 1, 0, 0]^T$. Furthermore, in equation (22.32) the third and the fourth homogeneous coordinates of the transformed point are not affected by the first and the second coordinates of the original point, requiring $t_{13} = t_{14} = t_{23} = t_{24} = 0$. The conditions on the third and the fourth homogeneous coordinates can be formalized by the following equations:

$$-f_p \cdot t_{33} + t_{43} = -f_p, \qquad -f_p \cdot t_{34} + t_{44} = f_p \ .$$

Applying the same procedure for the intersection point of the projection line and the back clipping plane, we can obtain other two equations:

$$-b_p \cdot t_{33} + t_{43} = b_p, \qquad -b_p \cdot t_{34} + t_{44} = b_p \ .$$

Solving this system of linear equations, the matrix of the perspective transformation can be expressed as:

$$\mathbf{T}_{persp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -(f_p + b_p)/(b_p - f_p) & -1 \\ 0 & 0 & -2 \cdot f_p \cdot b_p/(b_p - f_p) & 0 \end{bmatrix} \ .$$

Since perspective transformation is not affine, the fourth homogeneous coordinate of the transformed point is usually not 1. If we wish to express the coordinates of the transformed point in Cartesian coordinates, the first three homogeneous coordinates should be divided by the fourth coordinate. Homogeneous linear transforms map line segment to line segment and triangle to triangle, but it may happen that the resulting line segment or triangle contains ideal points (Subsection 22.5.2). The intuition behind the homogeneous division is a traveling from the projective space to the Euclidean space, which converts a line segment containing an ideal point to two half lines. If just the two endpoints of the line segment is transformed, then it is not unambiguous whether the two transformed points need to be connected by

a line segment or the complement of this line segment should be considered as the result of the transformation. This ambiguity is called the ***wrap around problem***.

The wrap around problem does not occur if we can somehow make sure that the original shape does not contain points that might be mapped onto ideal points. Examining the matrix of the perspective transformation we can conclude that the fourth homogeneous coordinate of the transformed point will be equal to the $-z$ coordinate of the original point. Ideal points having zero fourth homogeneous coordinate ($h = 0$) may thus be obtained transforming the points of plane $z = 0$, i.e. the plane crossing the origin and parallel to the window. However, if the shapes are clipped onto a first clipping plane being in front of the eye, then these points are removed. Thus the solution of the wrap around problem is the execution of the clipping step before the homogeneous division.

22.7.4. Clipping in homogeneous coordinates

The purpose of ***clipping*** is to remove all shapes that either cannot be projected onto the window or are not between the front and back clipping planes. To solve the ***wrap around problem***, clipping should be executed before the homogeneous division. The clipping boundaries in homogeneous coordinates can be obtained by transforming the screen coordinate AABB back to homogeneous coordinates. In screen coordinates, i.e. after homogeneous division, the points to be preserved by clipping meet the following inequalities:

$$-1 \le X = X_h/h \le 1, \qquad -1 \le Y = Y_h/h \le 1, \qquad -1 \le Z = Z_h/h \le 1\ . \tag{22.33}$$

On the other hand, points that are in front of the eye after camera transformation have negative z coordinates, and the perspective transformation makes the fourth homogeneous coordinate h equal to $-z$ in normalized camera space. Thus the fourth homogeneous coordinate of points in front of the eye is always positive. Let us thus add condition $h > 0$ to the set of conditions of inequalities (22.33). If h is positive, then inequalities (22.33) can be multiplied by h, resulting in the definition of the clipping region in homogeneous coordinates:

$$-h \le X_h \le h, \qquad -h \le Y_h \le h, \qquad -h \le Z_h \le h\ . \tag{22.34}$$

Points can be clipped easily, since we should only test whether or not the conditions of inequalities (22.34) are met. Clipping line segments and polygons, on the other hand, requires the computation of the intersection points with the faces of the clipping boundary, and only those parts should be preserved which meet inequalities (22.34).

Clipping algorithms using Cartesian coordinates were discussed in Subsection 22.4.3. Those methods can also be applied in homogeneous coordinates with two exceptions. Firstly, for homogeneous coordinates, inequalities (22.34) define whether a point is in or out. Secondly, intersections should be computed using the homogeneous coordinate equations of the line segments and the planes.

Let us consider a line segment with endpoints $[X_h^1, Y_h^1, Z_h^1, h^1]$ and $[X_h^2, Y_h^2, Z_h^2, h^2]$. This line segment can be an independent shape or an edge of a

polygon. Here we discuss the clipping on half space of equation $X_h \leq h$ (clipping methods on other half spaces are very similar). Three cases need to be distinquished:

1. If both endpoints of the line segment are inside, that is $X_h^1 \leq h^1$ and $X_h^2 \leq h^2$, then the complete line segment is in, thus is preserved.
2. If both endpoints are outside, that is $X_h^1 > h^1$ and $X_h^2 > h^2$, then all points of the line segment are out, thus it is completely eliminated by clipping.
3. If one endpoint is outside, while the other is in, then the intersection of the line segment and the clipping plane should be obtained. Then the endpoint being out is replaced by the intersection point. Since the points of a line segment satisfy equation (22.19), while the points of the clipping plane satisfy equation $X_h = h$, parameter t_i of the intersection point is computed as:

$$X_h(t_i) = h(t_i) \quad \Longrightarrow \quad X_h^1 \cdot (1 - t_i) + X_h^2 \cdot t_i = h^1 \cdot (1 - t_i) + h^2 \cdot t_i \quad \Longrightarrow$$
$$\Longrightarrow t_i = \frac{X_h^1 - h^1}{X_h^1 - X_h^2 + h^2 - h^1} \,.$$

Substituting parameter t_i into the equation of the line segment, homogeneous coordinates $[X_h^i, Y_h^i, Z_h^i, h^i]$ of the intersection point are obtained.

Clipping may introduce new vertices. When the vertices have some additional features, for example, the surface colour or normal vector at these vertices, then these additional features should be calculated for the new vertices as well. We can use linear interpolation. If the values of a feature at the two endpoints are I^1 and I^2, then the feature value at new vertex $[X_h(t_i), Y_h(t_i), Z_h(t_i), h(t_i)]$ generated by clipping is $I^1 \cdot (1 - t_i) + I^2 \cdot t_i$.

22.7.5. Viewport transformation

Having executed the perspective transformation, the Cartesian coordinates of the visible points are in $[-1, 1]$. These normalized device coordinates should be further scaled and translated according to the resolution of the screen and the position of the viewport where the image is expected. Denoting the left-bottom corner pixel of the screen viewport by (X_{min}, Y_{min}), the right-top corner by (X_{max}, Y_{max}), and Z coordinates expressing the distance from the eye are expected in (Z_{min}, Z_{max}), the matrix of the viewport transformation is:

$$\mathbf{T}_{viewport} = \begin{bmatrix} (X_{max} - X_{min})/2 & 0 & 0 & 0 \\ 0 & (Y_{max} - Y_{min})/2 & 0 & 0 \\ 0 & 0 & (Z_{max} - Z_{min})/2 & 0 \\ (X_{max} + X_{min})/2 & (Y_{max} + Y_{min})/2 & (Z_{max} + Z_{min})/2 & 1 \end{bmatrix} .$$

Coordinate systems after the perspective transformation are ***left handed***, unlike the coordinate systems of the virtual world and the camera, which are ***right handed***. Left handed coordinate systems seem to be unusual, but they meet our natural expectation that the screen coordinate X grows from left to right, the Y coordinate from bottom to top and, the Z coordinate grows in the direction of the camera target.

22.7.6. Rasterization algorithms

After clipping, homogeneous division, and viewport transformation, shapes are in the screen coordinate system where a point of coordinates (X, Y, Z) can be assigned to a pixel by extracting the first two Cartesian coordinates (X, Y).

Rasterization works in the screen coordinate system and identifies those pixels which have to be coloured to approximate the projected shape. Since even simple shapes can cover many pixels, rasterization algorithms should be very fast, and should be appropriate for hardware implementation.

Line drawing

Let the endpoints of a line segment be (X_1, Y_1) and (X_2, Y_2) in screen coordinates. Let us further assume that while we are going from the first endpoint towards the second, both coordinates are growing, and X is the faster changing coordinate, that is,

$$\Delta X = X_2 - X_1 \geq \Delta Y = Y_2 - Y_1 \geq 0 \ .$$

In this case the line segment is moderately ascending. We discuss only this case, other cases can be handled by exchanging the X, Y coordinates and replacing additions by substractions.

Line drawing algorithms are expected to find pixels that approximate a line in a way that there are no holes and the approximation is not fatter than necessary. In case of moderately ascending line segments this means that in each pixel column exactly one pixel should be filled with the colour of the line. This coloured pixel is the one closest to the line in this column. Using the following equation of the line

$$y = m \cdot X + b, \qquad \text{where} \quad m = \frac{Y_2 - Y_1}{X_2 - X_1}, \quad \text{and} \quad b = Y_1 - X_1 \cdot \frac{Y_2 - Y_1}{X_2 - X_1} \ , \tag{22.35}$$

in pixel column of coordinate X, the pixel closest to the line has Y coordinate that is equal to the rounding of $m \cdot x + b$. Unfortunately, the determination of Y requires a floating point multiplication, addition, and a rounding operation, which are too slow.

In order to speed up line drawing, we apply a fundamental trick of computer graphics, the ***incremental principle***. The incremental principle is based on the recognition that it is usually simpler to evaluate a function $y(X + 1)$ using value $y(X)$ than computing it from X. Since during line drawing the columns are visited one by one, when column $(X + 1)$ is processed, value $y(X)$ is already available. In case of a line segment we can write:

$$y(X + 1) = m \cdot (X + 1) + b = m \cdot X + b + m = y(X) + m \ .$$

Note that the evaluation of this formula requires just a single floating point addition (m is less than 1). This fact is exploited in ***digital differential analyzator algorithms*** (DDA-algorithms). The ***DDA line drawing algorithm*** is then:

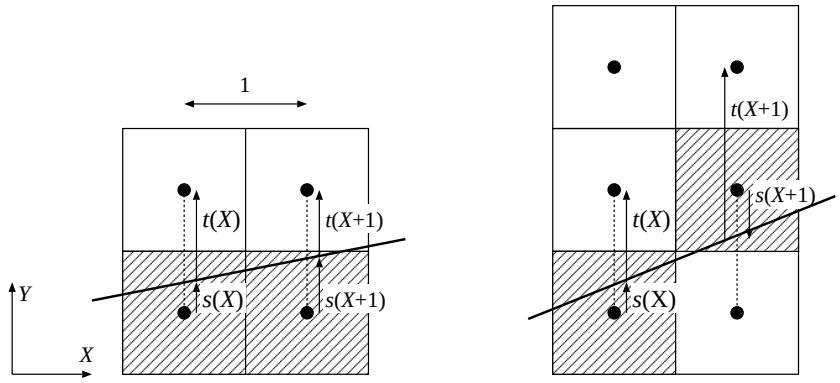


Figure 22.41. Notations of the Bresenham algorithm: s is the signed distance between the closest pixel centre and the line segment along axis Y, which is positive if the line segment is above the pixel centre. t is the distance along axis Y between the pixel centre just above the closest pixel and the line segment.

DDA-Line-Drawing$(X_1, Y_1, X_2, Y_2, \mathit{colour})$

```
1  m ← (Y2 − Y1)/(X2 − X1)
2  y ← Y1
3  for X ← X1 to X2
4      do Y ← Round(y)
5         Pixel-Write(X, Y, colour)
6         y ← y + m
```

Further speedups can be obtained using ***fixed point number representation***. This means that the product of the number and 2^T is stored in an integer variable, where T is the number of fractional bits. The number of fractional bits should be set to exclude cases when the rounding errors accumulate to an incorrect result during long iteration sequences. If the longest line segment covers L columns, then the minimum number of fractional bits guaranteeing that the accumulated error is less than 1 is $\log_2 L$. Thanks to clipping only lines fitting to the screen are rasterized, thus L is equal to the maximum screen resolution.

The performance and simplicity of the DDA line drawing algorithm can still be improved. On the one hand, the software implementation of the DDA algorithm requires shift operations to realize truncation and rounding operations. On the other hand – once for every line segment – the computation of slope m involves a division which is computationally expensive. Both problems are solved in the ***Bresenham line drawing algorithm.***

Let us denote the vertical, signed distance of the line segment and the closest pixel centre by s, and the vertical distance of the line segment and the pixel centre just above the closest pixel by t (Figure 22.41). As the algorithm steps onto the next pixel column, values s and t change and should be recomputed. While the new s and t values satisfy inequality $s < t$, that is, while the lower pixel is still closer

to the line segment, the shaded pixel of the next column is in the same row as in the previous column. Introducing error variable $e = s - t$, the row of the shaded pixel remains the same until this error variable is negative ($e < 0$). As the pixel column is incremented, variables s, t, e are updated using the incremental formulae ($\Delta X = X_2 - X_1$, $\Delta Y = Y_2 - Y_1$):

$$s(X+1) = s(X) + \frac{\Delta Y}{\Delta X}, \quad t(X+1) = t(X) - \frac{\Delta Y}{\Delta X} \quad \Longrightarrow \quad e(X+1) = e(X) + 2\frac{\Delta Y}{\Delta X} .$$

These formulae are valid if the closest pixel in column $(X + 1)$ is in the same row as in column X. If stepping to the next column, the upper pixel gets closer to the line segment (error variable e becomes positive), then variables s, t, e should be recomputed for the new closest row and for the pixel just above it. The formulae describing this case are as follows:

$$s(X+1) = s(X) + \frac{\Delta Y}{\Delta X} - 1, \quad t(X+1) = t(X) - \frac{\Delta Y}{\Delta X} + 1 \Longrightarrow$$

$$\Longrightarrow e(X+1) = e(X) + 2\left(\frac{\Delta Y}{\Delta X} - 1\right) .$$

Note that s is a signed distance which is negative if the line segment is below the closest pixel centre, and positive otherwise. We can assume that the line starts at a pixel centre, thus the initial values of the control variables are:

$$s(X_1) = 0, \qquad t(X_1) = 1 \quad \Longrightarrow \quad e(X_1) = s(X_1) - t(X_1) = -1 .$$

This algorithm keeps updating error variable e and steps onto the next pixel row when the error variable becomes positive. In this case, the error variable is decreased to have a negative value again. The update of the error variable requires a non-integer addition and the computation of its increment involves a division, similarly to the DDA algorithm. It seems that this approach is not better than the DDA.

Let us note, however, that the sign changes of the error variable can also be recognized if we examine the product of the error variable and a positive number. Multiplying the error variable by ΔX we obtain ***decision variable*** $E = e \cdot \Delta X$. In case of moderately ascending lines the decision and error variables change their sign simultaneously. The incremental update formulae of the decision variable can be obtained by multiplying the update formulae of error variable by ΔX:

$$E(X+1) = \begin{cases} E(X) + 2\Delta Y, \text{ if } Y \text{ is not incremented} , \\ \\ E(X) + 2(\Delta Y - \Delta X), \text{ if } Y \text{ needs to be incremented} . \end{cases}$$

The initial value of the decision variable is $E(X_1) = e(X_1) \cdot \Delta X = -\Delta X$.

The decision variable starts at an integer value and is incremented by integers in each step, thus it remains to be an integer and does not require fractional numbers at all. The computation of the increments need only integer additions or subtractions and multiplications by 2.

The complete ***Bresenham line drawing algorithm*** is:

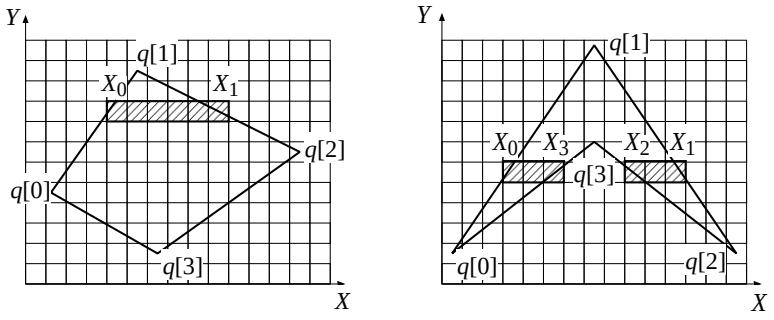


Figure 22.42. Polygon fill. Pixels inside the polygon are identified scan line by scan line.

BRESENHAM-LINE-DRAWING$(X_1, Y_1, X_2, Y_2, \mathit{colour})$

```
1  ΔX ← X2 − X1
2  ΔY ← Y2 − Y1
3  (dE+, dE−) ← (2(ΔY − ΔX), 2ΔY)
4  E ← −ΔX
5  Y ← Y1
6  for X ← X1 to X2
7      do if E ≤ 0
8            then E ← E + dE−          ▷ The line stays in the current pixel row.
9            else E ← E + dE+          ▷ The line steps onto the next pixel row.
10                 Y ← Y + 1
11         PIXEL-WRITE(X, Y, colour)
```

The fundamental idea of the Bresenham algorithm was the replacement of the fractional error variable by an integer decision variable in a way that the conditions used by the algorithm remained equivalent. This approach is also called the ***method of invariants***, which is useful in many rasterization algorithms.

Polygon fill

The input of an algorithm filling single connected polygons is the array of vertices $\vec{q}[0], \ldots, \vec{q}[m-1]$ (this array is usually the output of the polygon clipping algorithm). Edge e of the polygon connects vertices $\vec{q}[e]$ and $\vec{q}[e+1]$. The last vertex needs not be treated in a special way if the first vertex is put again after the last vertex in the array. Multiply connected polygons are defined by more than one closed polylines, thus are specified by more than one vertex arrays.

The filling is executed by processing a horizontal pixel row called ***scan line*** at a time. For a single scan line, the pixels belonging to the interior of the polygon can be found by the following steps. First the intersections of the polygon edges and the scan line are calculated. Then the intersection points are sorted in the ascending order of their X coordinates. Finally, pixels between the first and the second intersection points, and between the third and the fourth intersection points,

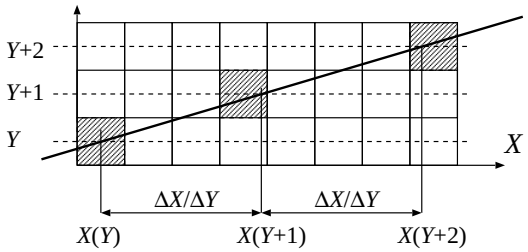


Figure 22.43. Incremental computation of the intersections between the scan lines and the edges. Coordinate X always increases with the reciprocal of the slope of the line.

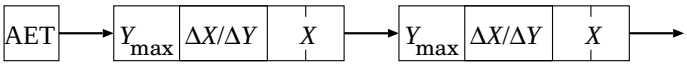


Figure 22.44. The structure of the active edge table.

or generally between the $(2i+1)$th and the $(2i+2)$th intersection points are set to the colour of the polygon (Figure 22.42). This algorithm fills those pixels which can be reached from infinity by crossing the polygon boundary odd number of times.

The computation of the intersections between scan lines and polygon edges can be speeded up using the following observations:

1. An edge and a scan line can have intersection only if coordinate Y of the scan line is between the minimum and maximum Y coordinates of the edge. Such edges are the ***active edges***. When implementing this idea, an ***active edge table*** (***AET*** for short) is needed which stores the currently active edges.

2. The computation of the intersection point of a line segment and the scan line requires floating point multiplication, division, and addition, thus it is time consuming. Applying the ***incremental principle***, however, we can also obtain the intersection point of the edge and a scan line from the intersection point with the previous scan line using a single, fixed-point addition (Figure 22.43).

When the incremental principle is exploited, we realize that coordinate X of the intersection with an edge always increases by the same amount when scan line Y is incremented. If the edge endpoint having the larger Y coordinate is (X_{max}, Y_{max}) and the endpoint having the smaller Y coordinate is (X_{min}, Y_{min}), then the increment of the X coordinate of the intersection is $\Delta X/\Delta Y$, where $\Delta X = X_{max} - X_{min}$ and $\Delta Y = Y_{max} - Y_{min}$. This increment is usually not an integer, hence increment $\Delta X/\Delta Y$ and intersection coordinate X should be stored in non-integer, preferably fixed-point variables. An active edge is thus represented by a fixed-point increment $\Delta X/\Delta Y$, the fixed-point coordinate value of intersection X, and the maximum vertical coordinate of the edge (Y_{max}). The maximum vertical coordinate is needed to recognize when the edge becomes inactive.

Scan lines are processed one after the other. First, the algorithm determines which edges become active for this scan line, that is, which edges have minimum Y coordinate being equal to the scan line coordinate. These edges are inserted into

the active edge table. The active edge table is also traversed and those edges whose maximum Y coordinate equals to the scan line coordinate are removed (note that this way the lower end of an edge is supposed to belong to the edge, but the upper edge is not). Then the active edge table is sorted according to the X coordinates of the edges, and the pixels between each pair of edges are filled. Finally, the X coordinates of the intersections in the edges of the active edge table are prepared for the next scan line by incrementing them by the reciprocal of the slope $\Delta X/\Delta Y$.

POLYGON-FILL(*polygon*, *colour*)

```
1  for Y ← 0 to Y_max
2      do for each edge of polygon          ▷ Put activated edges into the AET.
3             do if edge.ymin = Y
4                    then PUT-AET(edge)
5         for each edge of the AET   ▷ Remove deactivated edges from the AET.
6             do if edge.ymax ≤ Y
7                    then DELETE-FROM-AET(edge)
8         SORT-AET                                  ▷ Sort according to X.
9         for each pair of edges (edge1, edge2) of the AET
10            do for X ← ROUND(edge1.x) to ROUND(edge2.x)
11                   do PIXEL-WRITE(X, Y, colour)
11        for each edge in the AET                  ▷ Incremental principle.
12            do edge.x ← edge.x + edge.ΔX/ΔY
```

The algorithm works scan line by scan line and first puts the activated edges ($edge.ymin = Y$) to the active edge table. The active edge table is maintained by three operations. Operation PUT-AET(*edge*) computes variables ($Y_{max}, \Delta X/\Delta Y, X$) of an edge and inserts this structure into the table. Operation DELETE-FROM-AET removes an item from the table when the edge is not active any more ($edge.ymax \leq Y$). Operation SORT-AET sorts the table in the ascending order of the X value of the items. Having sorted the lists, every two consecutive items form a pair, and the pixels between the endpoints of each of these pairs are filled. Finally, the X coordinates of the items are updated according to the incremental principle.

22.7.7. Incremental visibility algorithms

The three-dimensional ***visibility problem*** is solved in the screen coordinate system. We can assume that the surfaces are given as triangle meshes.

Z-buffer algorithm

The ***z-buffer algorithm*** finds that surface for each pixel, where the Z coordinate of the visible point is minimal. For each pixel we allocate a memory to store the minimum Z coordinate of those surfaces which have been processed so far. This memory is called the ***z-buffer*** or the ***depth-buffer.***

When a triangle of the surface is rendered, all those pixels are identified which fall into the interior of the projection of the triangle by a triangle filling algorithm. As

the filling algorithm processes a pixel, the Z coordinate of the triangle point visible in this pixel is obtained. If this Z value is larger than the value already stored in the z-buffer, then there exists an already processed triangle that is closer than the current triangle in this given pixel. Thus the current triangle is obscured in this pixel and its colour should not be written into the raster memory. However, if the new Z value is smaller than the value stored in the z-buffer, then the current triangle is the closest so far, and its colour and Z coordinate should be written into the pixel and the z-buffer, respectively.

The z-buffer algorithm is then:

```
Z-BUFFER()
1  for each pixel p                                          ▷ Clear screen.
2      do PIXEL-WRITE(p, background-colour)
3          z-buffer[p] ← maximum value after clipping
4  for each triangle o                                       ▷ Rendering.
5      do for each pixel p of triangle o
6          do Z ← coordinate Z of that point o which projects onto pixel p
7              if Z < z-buffer[p]
8                  then PIXEL-WRITE(p, colour of triangle o in this point)
9                       z-buffer[p] ← Z
```

When the triangle is filled, the general polygon filling algorithm of the previous section could be used. However, it is worth exploiting the special features of the triangle. Let us sort the triangle vertices according to their Y coordinates and assign index 1 to the vertex of the smallest Y coordinate and index 3 to the vertex of the largest Y coordinate. The third vertex gets index 2. Then let us cut the triangle into two pieces with scan line Y_2. After cutting we obtain a "lower" triangle and an "upper" triangle. Let us realize that in such triangles the first (left) and the second (right) intersections of the scan lines are always on the same edges, thus the administration of the polygon filling algorithm can be significantly simplified. In fact, the active edge table management is not needed anymore, only the incremental intersection calculation should be implemented. The classification of left and right intersections depend on whether (X_2, Y_2) is on the right or on the left side of the oriented line segment from (X_1, Y_1) to (X_3, Y_3). If (X_2, Y_2) is on the left side, the projected triangle is called ***left oriented,*** and ***right oriented*** otherwise.

When the details of the algorithm is introduced, we assume that the already re-indexed triangle vertices are

$$\vec{r}_1 = [X_1, Y_1, Z_1], \qquad \vec{r}_2 = [X_2, Y_2, Z_2], \qquad \vec{r}_3 = [X_3, Y_3, Z_3].$$

The rasterization algorithm is expected to fill the projection of this triangle and also to compute the Z coordinate of the triangle in every pixel (Figure 22.45).

The Z coordinate of the triangle point visible in pixel X, Y is computed using the equation of the plane of the triangle (equation (22.1)):

$$n_X \cdot X + n_Y \cdot Y + n_Z \cdot Z + d = 0\ , \tag{22.36}$$

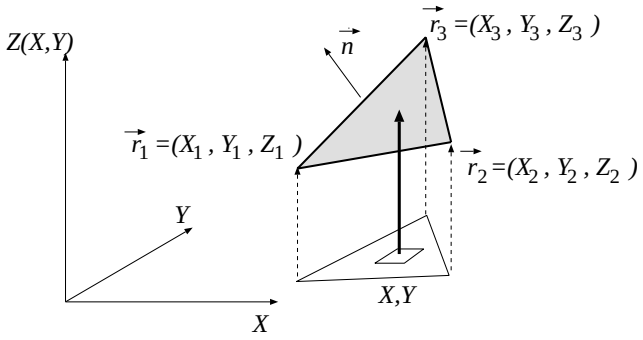


Figure 22.45. A triangle in the screen coordinate system. Pixels inside the projection of the triangle on plane XY need to be found. The Z coordinates of the triangle in these pixels are computed using the equation of the plane of the triangle.

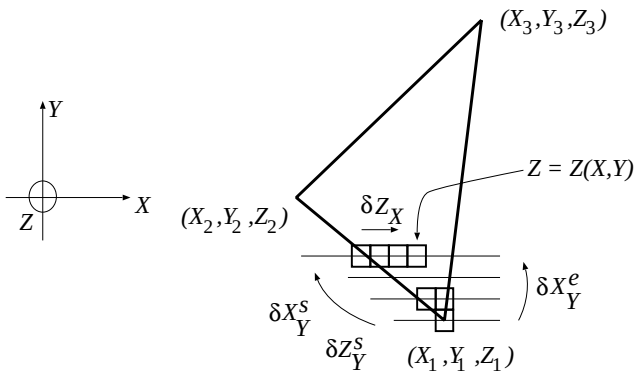


Figure 22.46. Incremental Z coordinate computation for a left oriented triangle.

where $\vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1)$ and $d = -\vec{n} \cdot \vec{r}_1$. Whether the triangle is left oriented or right oriented depends on the sign of the Z coordinate of the normal vector of the plane. If n_Z is negative, then the triangle is left oriented. If it is negative, then the triangle is right oriented. Finally, when n_Z is zero, then the projection maps the triangle onto a line segment, which can be ignored during filling.

Using the equation of the plane, function $Z(X, Y)$ expressing the Z coordinate corresponding to pixel X, Y is:

$$Z(X,Y) = -\frac{n_X \cdot X + n_Y \cdot Y + d}{n_Z} \ . \tag{22.37}$$

According to the incremental principle, the evaluation the Z coordinate can take advantage of the value of the previous pixel:

$$Z(X+1,Y) = Z(X,Y) - \frac{n_X}{n_Z} = Z(X,Y) + \delta Z_X \ . \tag{22.38}$$

Since increment δZ_X is constant for the whole triangle, it needs to be computed

only once. Thus the calculation of the Z coordinate in a scan line requires just a single addition per pixel. The Z coordinate values along the edges can also be obtained incrementally from the respective values at the previous scan line (Figure 22.46). The complete incremental algorithm which renders a lower left oriented triangle is as follows (the other cases are very similar):

Z-BUFFER-LOWER-TRIANGLE$(X_1, Y_1, Z_1, X_2, Y_2, Z_2, X_3, Y_3, Z_3, \mathit{colour})$

1 $\vec{n} \leftarrow ((X_2, Y_2, Z_2) - (X_1, Y_1, Z_1)) \times ((X_3, Y_3, Z_3) - (X_1, Y_1, Z_1))$ ▷ Normal vector.
2 $\delta Z_X \leftarrow -n_X / n_Z$ ▷ Z increment.
3 $(\delta X_Y^s, \delta Z_Y^s, \delta X_Y^e) \leftarrow ((X_2 - X_1)/(Y_2 - Y_1), (Z_2 - Z_1)/(Y_2 - Y_1), (X_3 - X_1)/(Y_3 - Y_1))$
4 $(X_{left}, X_{right}, Z_{left}) \leftarrow (X_1, X_1, Z_1)$
5 **for** $Y \leftarrow Y_1$ **to** Y_2
6 **do** $Z \leftarrow Z_{left}$
7 **for** $X \leftarrow$ ROUND(X_{left}) **to** ROUND(X_{right}) ▷ One scan line.
8 **do if** $Z < \mathit{z\text{-}buffer}[X, Y]$ ▷ Visibility test.
9 **then** PIXEL-WRITE(X, Y, colour)
10 $\mathit{z\text{-}buffer}[X, Y] \leftarrow Z$
11 $Z \leftarrow Z + \delta Z_X$
12 $(X_{left}, X_{right}, Z_{left}) \leftarrow (X_{left} + \delta X_Y^s, X_{right} + \delta X_Y^e, Z_{left} + \delta Z_Y^s)$
▷Next scan line.

This algorithm simultaneously identifies the pixels to be filled and computes the Z coordinates with linear interpolation. Linear interpolation requires just a single addition when a pixel is processed. This idea can also be used for other features as well. For example, if the colour of the triangle vertices are available, the colour of the internal points can be set to provide smooth transitions applying linear interpolation. Note also that the addition to compute the feature value can also be implemented by a special purpose hardware. Graphics cards have a great number of such interpolation units.

Warnock algorithm

If a pixel of the image corresponds to a given object, then its neighbours usually correspond to the same object, that is, visible parts of objects appear as connected territories on the screen. This is a consequence of object coherence and is called ***image coherence***.

If the situation is so fortunate—from a labor saving point of view—that a polygon in the object scene obscures all the others and its projection onto the image plane covers the image window completely, then we have to do no more than simply fill the image with the colour of the polygon. If no polygon edge falls into the window, then either there is no visible polygon, or some polygon covers it completely (Figure 22.47). The window is filled with the background colour in the first case, and with the colour of the closest polygon in the second case. If at least one polygon edge falls into the window, then the solution is not so simple. In this case, using a divide-and-conquer approach, the window is subdivided into four quarters, and each subwindow is searched recursively for a simple solution.

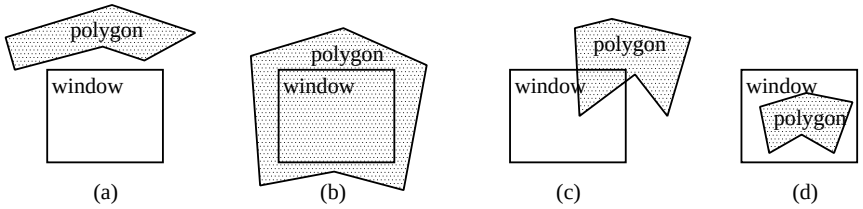


Figure 22.47. Polygon-window relations:: **(a)** distinct; **(b)** surrounding ; **(c)** intersecting; **(d)** contained.

The basic form of the algorithm called ***Warnock-algorithm*** rendering a rectangular window with screen coordinates X_1, Y_1 (lower left corner) and X_2, Y_2 (upper right corner) is this:

WARNOCK(X_1, Y_1, X_2, Y_2)

```
1  if X1 ≠ X2 or Y1 ≠ Y2                          ▷ Is the window larger than a pixel?
2    then if at least one edge projects onto the window
3           then                                  ▷ Non-trivial case: Subdivision and recursion.
4                WARNOCK(X1, Y1, (X1 + X2)/2, (Y1 + Y2)/2)
5                WARNOCK(X1, (Y1 + Y2)/2, (X1 + X2)/2, Y2)
6                WARNOCK((X1 + X2)/2, Y1, X2, (Y1 + Y2)/2)
7                WARNOCK((X1 + X2)/2, (Y1 + Y2)/2, X2, Y2)
8           else        ▷ Trivial case: window (X1, Y1, X2, Y2) is homogeneous.
9                polygon ← the polygon visible in pixel ((X1 + X2)/2, (Y1 + Y2)/2)
10               if no visible polygon
11                 then fill rectangle (X1, Y1, X2, Y2) with the background colour
12                 else fill rectangle (X1, Y1, X2, Y2) with the colour of polygon
```

Note that the algorithm can handle non-intersecting polygons only. The algorithm can be accelerated by filtering out those distinct polygons which can definitely not be seen in a given subwindow at a given step. Furthermore, if a surrounding polygon appears at a given stage, then all the others behind it can be discarded, that is all those which fall onto the opposite side of it from the eye. Finally, if there is only one contained or intersecting polygon, then the window does not have to be subdivided further, but the polygon (or rather the clipped part of it) is simply drawn. The price of saving further recurrence is the use of a scan-conversion algorithm to fill the polygon.

Painter's algorithm

If we simply scan convert polygons into pixels and draw the pixels onto the screen without any examination of distances from the eye, then each pixel will contain the colour of the last polygon falling onto that pixel. If the polygons were ordered by their distance from the eye, and we took the farthest one first and the closest one last, then the final picture would be correct. Closer polygons would obscure farther

ones — just as if they were painted an opaque colour. This method is known as the ***painter's algorithm***.

The only problem is that the order of the polygons necessary for performing the painter's algorithm is not always simple to compute. We say that a polygon P *does not obscure* another polygon Q if none of the points of Q is obscured by P. To have this relation, one of the following conditions should hold

1. Polygons P and Q do not overlap in Z range, and the minimum Z coordinate of polygon P is greater than the maximum Z coordinate of polygon Q.
2. The bounding rectangle of P on the XY plane does not overlap with that of Q.
3. Each vertex of P is farther from the viewpoint than the plane containing Q.
4. Each vertex of Q is closer to the viewpoint than the plane containing P.
5. The projections of P and Q do not overlap on the XY plane.

All these conditions are sufficient. The difficulty of their test increases, thus it is worth testing the conditions in the above order until one of them proves to be true. The first step is the calculation of an ***initial depth order***. This is done by sorting the polygons according to their maximal Z value into a list. Let us first take the polygon P which is the last item on the resulting list. If the Z range of P does not overlap with any of the preceding polygons, then P is correctly positioned, and the polygon preceding P can be taken instead of P for a similar examination. Otherwise P overlaps a set $\{Q_1, \ldots, Q_m\}$ of polygons. The next step is to try to check whether P *does not* obscure any of the polygons in $\{Q_1, \ldots, Q_m\}$, that is, that P is at its right position despite the overlapping. If it turns out that P obscures Q for a polygon in the set $\{Q_1, \ldots, Q_m\}$, then Q has to be moved behind P in the list, and the algorithm continues stepping back to Q. Unfortunately, this algorithm can run into an infinite loop in case of cyclic overlapping. Cycles can be resolved by cutting. In order to accomplish this, whenever a polygon is moved to another position in the list, we mark it. If a marked polygon Q is about to be moved again, then — assuming that Q is a part of a cycle — Q is cut into two pieces Q_1, Q_2 by the plane of P, so that Q_1 does not obscure P and P does not obscure Q_2, and only Q_1 is moved behind P.

BSP-tree

Binary space partitioning divides first the space into two halfspaces, the second plane divides the first halfspace, the third plane divides the second halfspace, further planes split the resulting volumes, etc. The subdivision can well be represented by a binary tree, the so-called ***BSP-tree*** illustrated in Figure 22.48. The kd-tree discussed in Subsection 22.6.2 is also a special version of BSP-trees where the splitting planes are parallel with the coordinate planes. The BSP-tree of this subsection, however, uses general planes.

The first splitting plane is associated with the root node of the BSP-tree, the second and third planes are associated with the two children of the root, etc. For our application, not so much the planes, but rather the polygons defining them, will be assigned to the nodes of the tree, and the set of polygons contained by the volume is also necessarily associated with each node. Each leaf node will then contain either

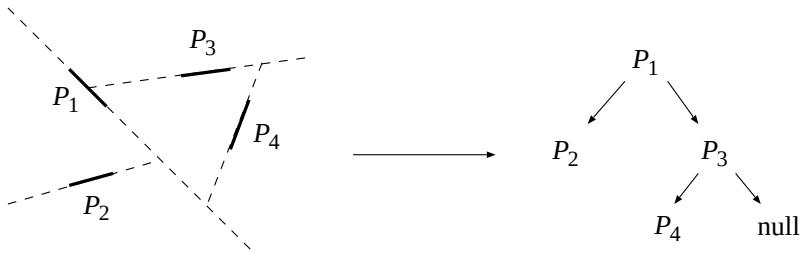


Figure 22.48. A BSP-tree. The space is subdivided by the planes of the contained polygons.

no polygon or one polygon in the associated set.

The BSP-Tree-Construction algorithm for creating the BSP-tree for a set S of polygons uses the following notations. A node of the binary tree is denoted by *node*, the polygon associated with the node by *node.polygon*, and the two child nodes by *node.left* and *node.right*, respectively. Let us consider a splitting plane of normal $\vec{n}$ and place vector $\vec{r}_0$. Point $\vec{r}$ belongs to the positive (right) subspace of this plane if the sign of scalar product $\vec{n} \cdot (\vec{r} - \vec{r}_0)$ is positive, otherwise it is in the negative (left) subspace. The BSP construction algorithm is:

BSP-Tree-Construction(S)

```
1  Create a new node
2  if S is empty or contains just a single polygon
3     then node.polygon ← S
4          node.left ← null
5          node.right ← null
6     else node.polygon ← one polygon from list S
7          Remove polygon node.polygon from list S
8          S+ ← polygons of S which overlap with the positive subspace
                        of node.polygon
9          S− ← polygons of S which overlap with the negative subspace
                        of node.polygon
10         node.right ← BSP-Tree-Construction(S+)
11         node.left ← BSP-Tree-Construction(S−)
12 return node
```

The size of the BSP-tree, i.e. the number of polygons stored in it, is on the one hand highly dependent on the nature of the object scene, and on the other hand on the "choice strategy" used when one polygon from list S is selected.

Having constructed the BSP-tree the visibility problem can be solved by traversing the tree in the order that if a polygon obscures another than it is processed later. During such a traversal, we determine whether the eye is at the left or right subspace at each node, and continue the traversal in the child *not* containing the eye. Having processed the child not containing the eye, the polygon of the node is

drawn and finally the child containing the eye is traversed recursively.

Exercises

22.7-1 Implement the complete Bresenham algorithm that can handle not only moderately ascending but arbitrary line segments.
22.7-2 The presented polygon filling algorithm tests each edges at a scan line whether it becomes active here. Modify the algorithm in a way that such tests are not executed at each scan line, but only once.
22.7-3 Implement the complete z-buffer algorithm that renders left/righ oriented, upper/lower triangles.
22.7-4 Improve the presented Warnock-algorithm and eliminate further recursions when only one edge is projected onto the subwindow.
22.7-5 Apply the BSP-tree for discrete time collision detection.
22.7-6 Apply the BSP-tree as a space partitioning structure for ray tracing.

Problems

22-1 Ray tracing renderer
Implement a rendering system applying the ray tracing algorithm. Objects are defined by triangle meshes and quadratic surfaces, and are associated with diffuse reflectivities. The virtual world also contains point light sources. The visible colour of a point is proportional to the diffuse reflectivity, the intensity of the light source, the cosine of the angle between the surface normal and the illumination direction (Lambert's law), and inversely proportional with the distance of the point and the light source. To detect whether or not a light source is not occluded from a point, use the ray tracing algorithm as well.

22-2 Continuous time collision detection with ray tracing
Using ray tracing develop a continuous time collision detection algorithm which computes the time of collision between a moving and rotating polyhedron and a still half space. Approximate the motion of a polygon vertex by a uniform, constant velocity motion in small intervals dt.

22-3 Incremental rendering system
Implement a three-dimensional renderer based on incremental rendering. The modelling and camera transforms can be set by the user. The objects are given as triangle meshes, where each vertex has colour information as well. Having transformed and clipped the objects, the z-buffer algorithm should be used for hidden surface removal. The colour at the internal points is obtained by linear interpolation from the vertex colours.

Chapter notes

The elements of Euclidean, analytic and projective geometry are discussed in the books of Maxwell [236, 237] and Coxeter [76]. The application of projective geometry

in computer graphics is presented in Herman's dissertation [163] and Krammer's paper [203]. Curve and surface modelling is the main focus of computer aided geometric design (CAD, CAGD), which is discussed by Gerald Farin [103], and Rogers and Adams [288]. Geometric models can also be obtained measuring real objects, as proposed by *reverse engineering methods* [336]. Implicit surfaces can be studied by reading Bloomenthal's work [46]. Solid modelling with implicit equations is also booming thanks to the emergence of *functional representation methods (F-Rep)*, which are surveyed at http://cis.k.hosei.ac.jp/~F-rep. Blobs have been first proposed by Blinn [45]. Later the exponential influence function has been replaced by polynomials [348], which are more appropriate when roots have to be found in ray tracing.

Geometric algorithms give solutions to geometric problems such as the creation of convex hulls, clipping, containment test, tessellation, point location, etc. This field is discussed in the books of Preparata and Shamos [277] and of Marc de Berg [81, 82]. The triangulation of general polygons is still a difficult topic despite to a lot of research efforts. Practical triangulation algorithms run in $O(n \lg n)$ [82, 297, 354], but Chazelle [62] proposed an optimal algorithm having linear time complexity. The presented proof of the *two ears theorem* has originally been given by Joseph O'Rourke [259]. Subdivision surfaces have been proposed and discussed by Catmull and Clark [59], Warren and Weimer [338], and by Brian Sharp [300, 299]. The butterfly subdivision approach has been published by Dyn et al. [95]. The *Sutherland–Hodgeman polygon clipping* algorithm is taken from [308].

Collision detection is one of the most critical problems in computer games since it prevents objects to fly through walls and it is used to decide whether a bullet hits an enemy or not. Collision detection algorithms are reviewed by Jiménez, Thomas and Torras [181].

Glassner's book [132] presents many aspects of ray tracing algorithms. The *3D DDA algorithm* has been proposed by Fujimoto et al. [122]. Many papers examined the complexity of ray tracing algorithms. It has been proven that for N objects, ray tracing can be solved in $O(\lg N)$ time [81, 311], but this is theoretical rather than practical result since it requires $\Omega(N^4)$ memory and preprocessing time, which is practically unacceptable. In practice, the discussed heuristic schemes are preferred, which are better than the naive approach only in the average case. Heuristic methods have been analyzed by probabilistic tools by Márton [311], who also proposed the probabilistic scene model used in this chapter as well. We can read about heuristic algorithms, especially about the efficient implementation of the kd-tree based ray tracing in Havran's dissertation [157]. A particularly efficient solution is given in Szécsi's paper [310].

The probabilistic tools, such as the Poisson point process can be found in the books of Karlin and Taylor [187] and Lamperti [208]. The cited fundamental law of integral geometry can be found in the book of Santaló [294].

The geoinformatics application of quadtrees and octrees are also discussed in chapter 16 of this book.

The algorithms of incremental image synthesis are discussed in many computer graphics textbooks [114]. Visibility algorithms have been compared in [309, 312]. The *painter's algorithm* has been proposed by Newell et al. [254]. Fuchs examined the construction of minimal depth BSP-trees [121]. The source of the Bresenham

algorithm is [49].

Graphics cards implement the algorithms of incremental image synthesis, including transformations, clipping, z-buffer algorithm, which are accessible through graphics libraries *(OpenGL, DirectX)*. Current graphics hardware includes two programmable processors, which enables the user to modify the basic rendering pipeline. Furthermore, this flexibility allows non graphics problems to be solved on the graphics hardware. The reason of using the graphics hardware for non graphics problems is that graphics cards have much higher computational power than CPUs. We can read about such algorithms in the ShaderX or in the GPU Gems [106] series or visiting the http://www.gpgpu.org web page.

23. Human-Computer Interaction

In the internet—within http://www.hcibib.org/—the following definition is found: "Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them... Some of its special concerns are:

- the joint performance of tasks by humans and machines;
- the structure of communication between human and machine;
- human capabilities to use machines (including the learnability of interfaces);
- algorithms and programming of the interface itself;
- engineering concerns that arise in designing and building interfaces;
- the process of specification, design, and implementation of interfaces.

... Human-computer interaction thus has science, engineering, and design aspects."

Many of these topics do only marginally concern algorithms in the classical sense. Therefore, in this chapter we concentrate on human-computer scenario for problem solving where the machines are forced to do lots of computation, and the humans have a role as intelligent controllers and directors.

23.1. Multiple-choice systems

Humans are able to think, to feel, and to sense—and they adopt quickly to a new situation. We can also compute, but not too well. In contrast, computers are giants in computing—they crunch bits and bytes like maniacs. However, they cannot do anything else but computing—especially they are not very flexible. Combining the different gifts and strengths of humans and machines in appropriate ways may lead to impressive performances.

One suitable approach for such team work is *"Multiple-Choice Optimisation"*. In a "Multiple-Choice System" the computer gives a clear handful of candidate solutions, two or three or five... Then a human expert makes the final choice amongst these alternatives. One key advantage of a proper multiple-choice approach is that the human is not drown by deluges of data.

Multiple-Choice Systems may be especially helpful in realtime scenarios of the following type: In principle there is enough time to compute a perfect solution. But certain parameters of the problem are unknown or fuzzy. They concretise only in a very late moment, when there is no more time for elaborate computations. Now assume that the decision maker has used multiple-choice algorithms to generate some good candidate solutions in beforehand. When the exact problem data show up he may select an appropriate one of these alternatives in realtime.

An example from vehicle routing is given. A truck driver has to go from A to Z. Before the trip he uses PC software to find two or three different good routes and prints them out. During the trip radio gives information on current traffic jams or weather problems. In such moments the printed alternatives help the driver to switch routes in realtime.

However, it is not at all easy to have the computer finding good small samples of solutions. Naively, the most natural way seems to be the application of k-best algorithms: Given a (discrete) optimisation problem with some objective function, the k best solutions are computed for a prescribed integer k. However, such k-best solutions tend to be *micro mutations* of each other instead of *true alternatives*.

Figure 23.1 exhibits a typical pattern: In a grid graph of dimension 100×100 the goal was to find short paths from the lower left to the upper right corner. The edge lengths are random numbers, not indicated in the diagram. The 1000 (!) shortest paths were computed, and their union is shown in the figure. The similarity amongst all these paths is striking. Watching the picture from a somewhat larger distance will even give the impression of only a single path, drawn with a bushy pencil. (The computation of alternative short paths will also be the most prominent example case in Section 23.2)

Often the term "multiple-choice" is used in the context of "multiple-choice tests". This means something completely different. The difference between multiple-choice optimisation and multiple-choice tests lies in the type and quality of the candidate solutions:

- In multiple-choice tests always at least one of the answers is "correct", whereas others may be right or wrong. Beforehand an authority (the test designer) has prepared the question together with the candidate answers and the decision which of them are correct ones.
- In the optimisation situation nothing is clear: Perhaps all of the candidate solutions are ok, but it may also be that they all are not appropriate. And there is typically no master who tells the human whether his choice is good or not. Because of this uncertainty many humans really need some initiation time to accept their role within a multiple-choice system.

23.1.1. Examples of multiple-choice systems

(1) **Short Paths**
Starting in the early 1990's, PC programs for vehicle routing have become more and more popular. In 1997 the Dutch software company AND was first to sell such a program which did not only compute the "best" (= shortest or quickest) route but

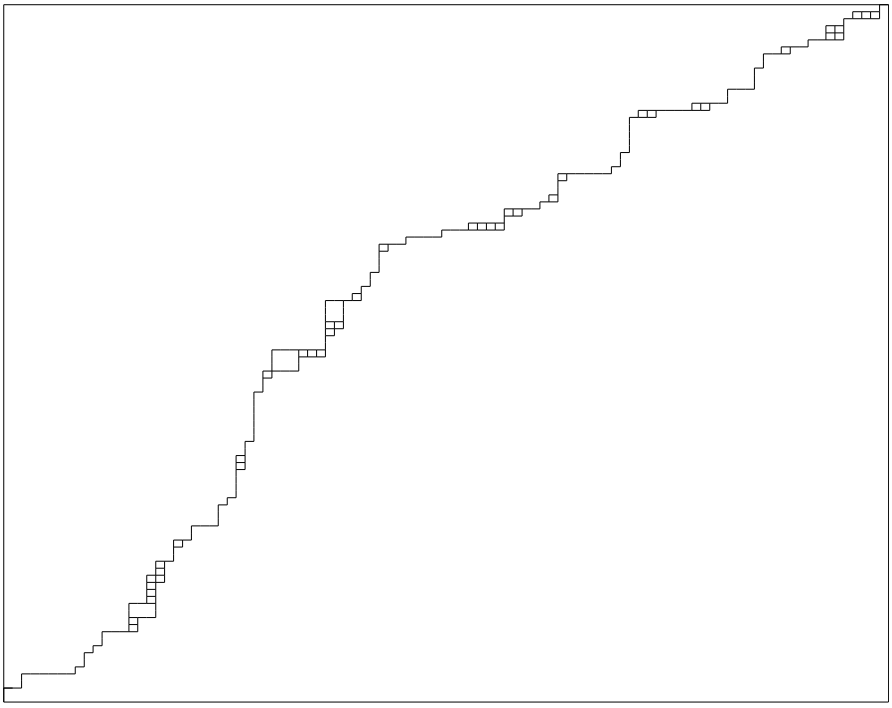


Figure 23.1. 1000 shortest paths in a 100×100 grid-graph, printed in overlap.

also one or two alternatives. The user had the choice to request all these candidate solutions simultaneously or one after the other. The user was also allowed to determine some parameters for route visualisation, namely different colours and thickness for best, second, third choice. Related is work by F. Berger. She developed a method to identify linear structures (like roads, rails, rivers, ...) in grey level satellite images. Typically, candidate structures are not unique, and the algorithm of Berger makes several alternative proposals. The Berger method is based on algorithms for generating short alternative paths.

(2) **Travelling Salesperson Problem and the Drilling of Holes in Circuit Boards**

In the Travelling Salesperson Problem (TSP) N locations are given and their mutual distances. The task is to find a shortest round trip through all N points. TSP is NP-complete. One important application in electronic industry is the drilling of holes in circuit boards. Here the locations are the points where the drill has to make the holes; the goal is to minimise the time needed by the drill. In practice, however, it turns out that the length of the drilling tour is not the only criterion for success: Depending on the drilling tour there occur small or more severe tensions in the circuit board. Especially different tours may give different levels of tension. Unfortunately, the degrees of tension can not easily be computed in beforehand. So it makes sense

to compute a few alternative short drilling tours and select that one which is best with respect to the minimisation of tension.

(3) **Internet Search Engines**
In most cases an internet search engine will find tons of hits, but of course a normal human user is not able nor willing to look through all of them. So, one of the key tasks for a search engine designer is to find good shortlisting mechanisms. As a rule of thumb, the first ten hits in the output should be both most relevant and sufficiently spread. In this field and also in e-commerce Multiple-Choice Systems are often called "*Recommender Systems*".

(4) **Trajectories for Interplanetary Space Missions**
Space missions to distant planets, planetoids, and comets are high-tech adventures. Two key aspects are budget restrictions and the requirement that the probes need extremely high speeds to reach their destinations in time. "Gravity assist" maneuvers help to speed up missiles by narrow flybys at intermediate planets, thus saving fuel and time. In recent years trajectories with gravity assists have become more and more complex, typically involving whole sequences of several flybys. Prominent examples are the mission Cassini to planet Saturn with flyby sequence Venus-Venus-Earth-Jupiter, the mission Rosetta to Comet "67P/Churyumov-Gerasimenko" with flyby sequence Earth-Mars-Earth-Earth, and the Messenger-mission to Mercury with flyby sequence Earth-Venus-Venus-Mercury-Mercury. The current art of trajectory computing allows to finetune a principal route. However, first of all such principal routes have been designed by human engineers with their fantasy and creativity. Computer-generation of (alternative) principal flyby tours is still in its infancies.

(5) **Chess with Computer Assistance**
Commercial chess computers came up in the late 1970's. Their playing strength increases steadily, and nowadays the best PC programs play on one level with the best human players. However, teams with both human and computer members are stronger than humans alone or computers alone. One of these authors (Althöfer) made many chess experiments with Multiple-Choice Systems: In a setting called "*3-Hirn*" ("Triple Brain" in English, but the German term 3-Hirn has been adopted internationally) two different chess programs are running, typically on two independent PC's. Each one proposes a single candidate move, and a human player has the final choice amongst these (at most) two move candidates. In several experiments 3-Hirn showed amazing performance. The final data point was a match in 1997: two computer programs with Elo rating below 2550 each and a human amateur player (Elo 1900) beat the German No. 1 player (GM Yussupov, Elo 2640) by 5-3 in tournament play, thus achieving an event performance of higher than Elo 2700. After this event top human professionals were no longer willing to fight against 3-Hirn teams. The strength of 3-Hirn is to a large extent explained by the combination of two "orthogonal" chess strengths: chess computers propose only moves which are tactically sound and the human player contributes his strength in long-range planning.

Today, all top human chess professionals prepare intensively for their tournament games with the help of chess programs by analysing openings and games in multiple-choice mode. Even more extreme is the situation in correspondence chess, where players are officially allowed to use computer help within their games.

(6) **Travel and Holiday Information**
When someone plans a journey or a holiday, he typically compares different routes or offers, either at the railway station or in a travel agency or from home via internet. Customers typically do not inspect thousands of offers, but only a smaller or larger handful. In real life lots of (normal and strange) strategies can be found how companies, hotels, or airlines try to place their products amongst the top choices. For instance, it is common (bad) policy by many airlines to announce unrealistic short flight times. The only intention is to become top-placed in software (for travel agencies) which sorts all flights from A to B by ascending flight times. In many cases it is not an easy task for the customer to realize such tricks for successful "performance" in shortlisting processes.

(7) **RNA-Foldings**
Computation of RNA-foldings is one of the central topics in computational biology. The most prominent algorithms for this are based on dynamic programming. There exist online repositories, where people get alternative solutions in realtime.

Exercises

23.1-1 Collect practice in operating a multiple-choice system by computer-aided play of the patience game FreeCell. Download the tool BigBlackCell (BBC) from http://www.minet.uni-jena.de/~BigBlackCell/ and make yourself acquainted with the program. After some practising a normal user with the help of BBC should be able to solve in the average more than 60 FreeCell instances per hour.

23.2. Generating multiple candidate solutions

23.2.1. Generating candidate solutions with heuristics and repeated heuristics

Many optimisation problems are really hard, for instance the NP-complete ones. Exact (but slow) branch and bound procedures and unreliable (but quick) heuristics are two standard ways to find exact or approximate solutions. When the task is to generate several alternative solutions it is possible to make a virtue of necessity: there are normally many more good solutions than perfect ones – and different heuristics or heuristics with random elements will not always return the same good solution.

So, a simple strategy is to apply one or several heuristics repeatedly to the same problem, and to record the solutions generated during this process. Either, exactly as many solutions as needed are generated. Or a larger preliminary set of solutions is produced, giving the chance for improvement by shortlisting. Natural shortlisting criteria are quality and spread. Concerning spread, distance measures on the set of admissible solutions may be a helpful tool in conjunction with clustering algorithms.

Repeated runs of a single heuristic
The normal situation is that a heuristic contains randomness to a certain extent. Then no additional efforts are necessary: the heuristic is simply executed in independent runs, until enough different good solutions have been generated. Here we use

the Travelling Salesperson Problem (TSP) for N points as an example to demonstrate the approaches. For exchange heuristics and insertion heuristics on the TSP we give one example each, highlighting the probabilistic elements.

In the TSP with symmetric distances $d(i,j)$ between the points local search with 2-exchanges is a standard exchange heuristic. In the following pseudo-code $T(p)$ denote the p-th component of vector T.

LOCAL-SEARCH-WITH-2-EXCHANGES-FOR-TSP(N,d)

1 Generate a random starting tour $T=(i_1,i_2,\ldots,i_N)$.
2 **while** there exist indices p,q with $1\le p<q\le N$ and $q\ge p+2$, and
$d(T(p),T(p+1))+d(T(q),T(q+1))>$
$d(T(p),T(q))+d(T(p+1),T(q+1))$
▷ For the special case $q=N$ we take $q+1=1$.
3 **do** $T\leftarrow(i_1,\ldots,i_p,i_q,i_{q-1},\ldots,i_{p+1},i_{q+1},\ldots,i_N)$
4 compute the length l of tour T
5 **return** T,l

Random elements in this heuristic are the starting tour and the order in which edge pairs are checked in step 2. Different settings will lead to different local minima. In large problems, for instance with 1,000 random points in the unit square with Euclidean distance it is quite normal when 100 independent runs of the 2-exchange heuristic lead to almost 100 different local minima.

The next pseudo-code describes a standard insertion heuristic.

INSERTION-HEURISTIC-FOR-TSP(N,d)

1 generate a random permutation $(i_1,i_2,\ldots,i_N)$ from the elements of $\{1,2,\ldots,N\}$
2 $T\leftarrow(i_1,i_2)$
3 **for** $t\leftarrow 2$ **to** $N-1$
4 **do** find the minimum of $d(T(r),i_{t+1})+d(i_{t+1},T(r+1))-d(T(r),T(r+1))$
for $r\in\{1,\ldots,t\}$
▷ Here again $r+1=1$ for $r=t$.
let the minimum be at $r=s$
5 $T\leftarrow(T(1),\ldots,T(s),i_{t+1},T(s+1),\ldots,T(t))$
6 compute the length l of tour T
7 **return** T,l

So the elements are inserted one by one, always at the place where insertion results at minimal new length.

The random element is the permutation of the N points. Like for the 2-exchanges, different settings will typically lead to different local minima. Sometimes an additional chance for random choice occurs when for some step t the optimal insertion place is not unique.

Many modern heuristics are based on analogies to nature. In such cases the user has even more choices: In Simulated Annealing several (good) intermediate solutions from each single run may be taken; or from each single run of a Genetic Algorithm

several solutions may be taken, either representing different generations or multiple solutions of some selected generation.

A special technique for repeated exchange heuristics is based on the perturbation of local optima: First make a run to find a local optimum. Then randomise this first optimum by a sequence of random local changes. From the resulting solution start local search anew to find a second local optimum. Randomise this again and so on. The degree of randomisation steers how different the local optima in the sequence will become.

Even in case of a deterministic heuristic there may be chances to collect more than only one candidate solution: In tiebreak situations different choices may lead to different outcomes, or the heuristic may be executed with different precisions (=number of decimals) or with different rounding rules. In Subsection 23.2.4 penalty methods are described, with artificial modification of problem parameters (for instance increased edge lengths) in repeated runs. In anytime algorithms —like iterative deepening in game tree search—also intermediate solutions (for preliminary search depths) may be used as alternative candidates.

Collecting candidate solutions from different heuristic programs
When several heuristics for the same problem are available, each one of them may contribute one or several candidate solutions. The 3-Hirn setting, as described in item (5) of Subsection 23.1.1, is an extreme example of a multiple-choice system with more than one computer program: the two programs should be independent of each other, and they are running on distinct computers. (Tournament chess is played under strict time limits at a rate of three minutes per move. Wasting computational resources by having two programs run on a single machine in multi-tasking mode costs 60 to 80 rating points [161]). The chess programs used in 3-Hirn are standard of-the-shelf products, not specifically designed for use in a multiple-choice setting.

Every real world software has errors. Multiple-choice systems with independent programs have a clear advantage with respect to catastrophic failures. When two independent programs are run, each with the same probability p for catastrophic errors, then the probability for a simultaneous failure reduces to p^2. A human controller in a multiple-choice system will typically recognise when candidate solutions have catastrophic failures. So the "mixed" case (one normal and one catastrophic solution) with probability $2p(1-p)$ will not result in a catastrophe. Another advantage is that the programs do not need to have special k-best or k-choice mechanisms. Coinciding computer proposals may be taken as an indication that this solution is just really good.

However, multiple-choice systems with independent programs may also have weak spots:

- When the programs are of clearly different strength, this may bring the human selector in moral conflicts when he prefers a solution from the less qualified program.
- In multistep actions the proposals of different programs may be incompatible.
- For a human it costs extra time and mental energy to operate more than one program simultaneously.

- Not seldom – depending on programs and operating systems – a PC will run unstably in multi-tasking mode.

And of course it is not always guaranteed that the programs are really independent. For instance, in the late 1990's dozens of vehicle routing programs were available in Germany, all with different names and interfaces. However, they all were based on only four independent program kernels and data bases.

23.2.2. Penalty method with exact algorithms

A more controlled way to find different candidate solutions is given by the **penalty method**. The main idea of this method is illustrated on the route planning example. Starting with an optimal (or good) route R_1 we are looking for an alternative solution R_2 which fulfills the following two criteria as much as possible.

(i) R_2 should be good with respect to the objective function. Otherwise it is not worthwhile to choose it. In our example we have the length (or needed time) of the route as first objective.

(ii) R_2 should have not too much in common with the original solution. Otherwise it is not a *true* alternative. In case of so called *micro mutations* the risk is high that all these similar candidates have the same weak spots. In our example a "measure for similarity" may be the length of the parts shared by R_1 and R_2.

This means R_2 should have a short length but it should also have only little in common with R_1. Therefore we use a combination of the two objective functions – the length of the route and the length of the road sections shared by R_1 and R_2. This can be done by punishing the sections used by R_1 and solving the shortest path problem with this modified lengths to get the solution R_2.

By the size of the penalties different weightings of criteria (i) and (ii) can be modelled.

A most natural approach is to use **relative penalty factors**. This means that the length of each section belonging to R_1 is multiplied by a factor $1 + \varepsilon$.

PENALTY-METHOD-WITH-RELATIVE-PENALTY-FACTORS(G, s, t, ε)

1 find the shortest path R_1 from node s to node t in the weighted graph $G = (V, E, w)$
2 **for** all $e \in E$
3 **do if** e belongs to R_1
4 **then** $\widehat{w}(e) \leftarrow w(e) \cdot (1 + \varepsilon)$
5 **else** $\widehat{w}(e) \leftarrow w(e)$
6 find the the shortest path R_2 from node s to node t in the modified graph $\widehat{G} = (V, E, \widehat{w})$
7 compute its unmodified length $w(R_2)$
8 **return** (R_1, R_2) and $(w(R_1), w(R_2))$

Consider the following example.

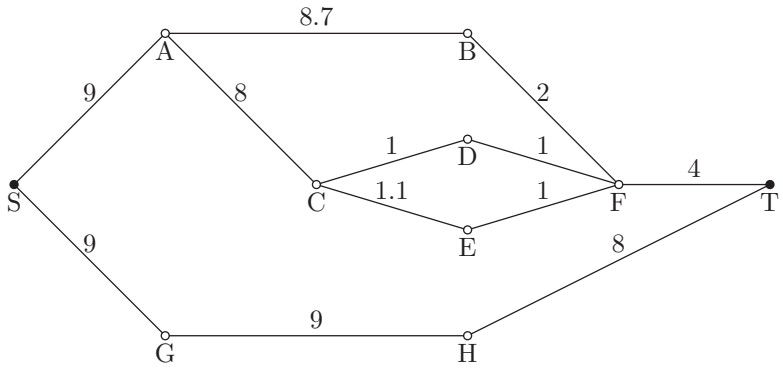


Figure 23.2. The graph for Examples 23.1, 23.2 and 23.6.

Example 23.1 Given is a graph $G = (V, E)$ with weighted edge lengths. In Figure 23.2 the numbers denote the length of the according edges. The shortest path from S to T is P_D via S - A - C - D - F - T with length 23. Multiplying all edge lengths of P_D by 1.1 and solving the obtained shortest path problem gives the alternative solution P_B via S - A - B - F - T with modified length 25.0 and normal length 23.7. The shared parts of P_D and P_B are S-A and F-T with total length 13.

The size of ε has to be appropriate for the situation. In the commercial vehicle routing program [16] all sections of a shortest (or fastest) route were multiplied by 1.2, i.e., $\varepsilon = 0.2$. Then the alternative route was computed. In [40] recognition of linear structures (streets, rivers, airport lanes) in satellite images was done by shortest path methods. Here $\varepsilon = 1.0$ turned out to be an appropriate choice for getting interesting alternative candidates.

Instead of relative penalty factors **additive penalties** might be used. That means we add a constant term ε to all edges we want to punish. The only modification of the algorithm above is in step 4.

4^* **then** $\widehat{w}(e) \leftarrow w(e) + \varepsilon$

Example 23.2 Given is the graph $G = (V, E)$ from Example 23.1 (see Figure 23.2). The shortest path from S to T is still P_D via S - A - C - D - F - T with length 23. Adding 0.1 to all edges of P_D and solving the resulting shortest path problem gives the alternative solution P_E via S - A - C - E - F - T with modified length 23.4 and normal length 23.1. P_D and P_E have three edges in common.

In principle this approach with additive penalties is not worse in comparison with multiplicative penalties. However, the method with multiplicative penalties has the advantage to be immune against artificial splits of edges.

For a generalisation of the penalty method from routing problems the following definition of optimisation problems is helpful.

Definition 23.1 *Let E be an arbitrary finite set and S a set of subsets of E. E is called the* ***base set*** *and the elements of S are* ***feasible subsets*** *of E. Let $w : E \to \mathbb{R}$ be a real valued weight function on E. For every $B \in S$ we set $w(B) = \sum_{e \in B} w(e)$.*

The optimisation problem $\min_{B \in S} w(B)$ is a ***Sum Type Optimisation Problem*** *or in short "$\sum$**-type problem**".*

Remarks:

1. The elements $B \in S$ are also called ***feasible solutions***.
2. By substitution of w by $-w$ every maximisation problem can be formulated as a minimisation problem. Therefore we will also call maximisation problems $\sum$-type problems.

Examples of $\sum$-type problems

- Shortest Path Problem
- Assignment Problem
- Travelling Salesperson Problem (TSP)
- Knapsack Problem
- Sequence Alignment Problem

Example 23.3 Consider the Knapsack Problem. Given a set of items $I = \{I_1, I_2, \ldots, I_n\}$, a weight function $w : I \to \mathbb{R}^+$, a value function $v : I \to \mathbb{R}^+$, and a knapsack capacity C. What is the most valuable collection of items whose weight sum does not exceed the knapsack capacity?

Choosing I as base set and S as the family of all subsets whose weight sum is smaller or equal to C gives a representation as a $\sum$-type problem: maximise $v(B)$ over all $B \in S$.

Abstract formulation of the penalty method for $\sum$-type problems

Definition 23.2 *Let E be an arbitrary set and S the set of feasible subsets of E. Let $w : E \to \mathbb{R}$ be a real-valued and $p : E \to \mathbb{R}^{\geq 0}$ a non-negative real-valued function on E.*

For every $\varepsilon > 0$, let B_ε be one of the optimal solutions of the problem

$$\min_{B \in S} f_\varepsilon(B) \ ,$$

$$\text{with} \quad f_\varepsilon(B) := w(B) + \varepsilon \cdot p(B) \ .$$

With an algorithm which is able to solve the unpunished problem $\min_{B \in S} w(B)$ we can also find the solutions B_ε. We just have to modify the function w by replacing $w(e)$ by $w(e) + \varepsilon \cdot p(e)$ for all $e \in E$. B_ε is called an ***$\varepsilon-$penalty solution*** *or an* ***$\varepsilon-$alternative****.*

Additionally we define the solution B_∞ of the problem

$$\operatorname{lex}\min_{B\in S}(p(B), w(B)) \quad \textit{(minimisation with respect to the lexicographical order)},$$

which has a minimal value $p(B)$ and among all such solutions a minimal value $w(B)$.

Remark. If both w and p are positive real-valued functions, there is a symmetry in the optimal solutions: B^* is an $\varepsilon-$penalty solution $(0 < \varepsilon < \infty)$ for the function pair (w, p), if and only if B^* is a $\frac{1}{\varepsilon}-$penalty solution for the pair (p, w).

To preserve this symmetry it makes sense to define B_0 as an optimal solution of the problem

$$\operatorname{lex}\min_{B\in S}(w(B), p(B)) \ .$$

That means B_0 is not only an optimal solution for the objective function w, but among all such solutions it has also a minimal p-value.

Example 23.4 We formulate the concrete Example 23.1 from page 1102 in this abstract $\sum$-type formulation. We know the shortest path P_D from S to T and search for a "good" alternative solution. The penalty function p is defined by

$$p(e) = \begin{cases} w(e) & \text{if } e \text{ is an edge of the shortest path } P_D \ , \\ 0 & \text{else} \ . \end{cases}$$

Finding penalty solutions for all parameters $\varepsilon \geq 0$

Often it is a priori not clear which choice of the penalty parameter ε produces good and interesting alternative solutions. With a "divide-and-conquer" algorithm one is able to find *all* solutions which can be produced by *any* parameter ε.

For finite sets S we give an efficient algorithm which generates a "small" set $\mathcal{B} \subseteq S$ of solutions with the following properties.

- For each element $B \in \mathcal{B}$ there exists an $\varepsilon \in \mathbb{R}^+ \cup \{\infty\}$ such that B is an optimal solution for the penalty parameter ε.
- For each $\varepsilon \in \mathbb{R}^+ \cup \{\infty\}$ there exists an element $B \in \mathcal{B}$ such that B is optimal for the penalty parameter ε.
- $\mathcal{B}$ has a minimal number of elements among all systems of sets which have the two properties above.

We call a solution B which is optimal for at least one penalty parameter **penalty-optimal**. The following algorithm 1105 finds a set of penalty-optimal solutions which covers all $\varepsilon \in \mathbb{R}^+ \cup \{\infty\}$.

For easier identification we arrange the elements of the set $\mathcal{B}$ in a fixed order $(B_{\varepsilon(1)}, B_{\varepsilon(2)}, \ldots, B_{\varepsilon(k)})$, with $0 = \varepsilon(1) < \varepsilon(2) < \cdots < \varepsilon(k) = \infty$.
The algorithm has to check that for $\varepsilon(i) < \varepsilon(i{+}1)$ there is no ε with $\varepsilon(i) < \varepsilon < \varepsilon(i{+}1)$ such that for this penalty parameter ε neither $B_{\varepsilon(i)}$ nor $B_{\varepsilon(i+1)}$ is optimal. Otherwise it has to identify such an ε and an ε-penalty solution B_ε. In step 11 of the pseudo code below the variable Border(i) is set to 1 if it turns out that such an intermediate ε does not exist.

We present the pseudo code and give some remarks.

Algorithm for finding a sequence $\mathcal{B}$ of penalty-optimal solutions covering all $\varepsilon \geq 0$ for the problem

$$\min_{B \in S} f_\varepsilon(B)$$
$$\text{with } f_\varepsilon(B) = w(B) + \varepsilon \cdot p(B) \quad .$$

DIVIDE-AND-COVER(w, p)

```
1  compute B_0, which minimises w(B) and has a p(B)-value as small as possible.
2  compute B_∞, which minimises p(B) and has a w(B)-value as small as possible.
3  if p(B_0) = p(B_∞)
4     then B ← {B_0}; E ← (0); Border← ∅
           (B_0 minimises the functions w and p and is optimal for all ε.)
5     else k ← 2; E = (ε(1), ε(2)) ← (0, ∞); Border(1) ← 0; B ← (B_0, B_∞).
6          while There is an i ∈ {1, 2, ..., k − 1} with Border(i) = 0.
7             do ε̄ ← (w(B_ε(i+1)) − w(B_ε(i))) / (p(B_ε(i)) − p(B_ε(i+1)))
8                Find an optimal solution B_ε̄ for the parameter ε̄.
9                if f_ε̄(B_ε̄) = f_ε̄(B_ε(i)) = f_ε̄(B_ε(i+1))
10                  then Border(i) ← 1
11                  else B ← (B_ε(1), ..., B_ε(i), B_ε̄, B_ε(i+1), ..., B_ε(k))
12                       E ← (ε(1), ..., ε(i), ε̄, ε(i + 1), ..., ε(k))
13                       Border← (Border(1), ..., Border(i), 0,Border(i + 1),...,
                                  Border(k − 1))
14                       k ← k + 1
15 return B, E, Border
```

At the end $\mathcal{B}$ is a sequence of different penalty-optimal solutions and the vector $\mathcal{E}$ includes consecutive epsilons.

This algorithm is based on the following properties:

(1) If B is an ε-optimal solution then there exists an interval $I_B = [\varepsilon_l, \varepsilon_h]$, $\varepsilon_l, \varepsilon_h \in \mathbb{R} \cup \{\infty\}$, such that B is optimal for all penalty parameters $\varepsilon \in I_B$ and for no other parameters.

(2) For two different solutions B and B' with nonempty optimality intervals I_B and $I_{B'}$, only three cases are possible.

- $I_B = I_{B'}$. This happens iff $w(B) = w(B')$ and $p(B) = p(B')$.
- I_B and $I_{B'}$ are disjoint.
- $I_B \cap I_{B'} = \{\bar{\varepsilon}\}$, this means the intersection contains only a single epsilon. This happens if I_B and $I_{B'}$ are neighbouring intervals.

By finiteness of E there are only finitely many feasible solutions $B \in S$. So, there can be only finitely many optimality intervals. Hence, (1) and (2) show that the interval $[0, \infty]$ can be decomposed into a set of intervals $\{[0 = \varepsilon_1, \varepsilon_2], [\varepsilon_2, \varepsilon_3], \ldots, [\varepsilon_k, \varepsilon_{k+1} = \infty]\}$. For each interval we have a different solution B which is optimal for all ε in this interval. We call such a solution an **interval representative**.

(3) The aim of the algorithm is to find the *borders* of such optimality intervals and for each interval a representing solution. In every iteration step an interval representative of a new interval or a new border between two different intervals will be found (in steps 7–13). When there are k optimality intervals with $k \geq 2$ it is sufficient to solve $2k-1$ problems of the type $\min_{B \in S} w(B) + \varepsilon \cdot p(B)$ to detect all of them and to find representing solutions.

Unimodality property of the ε-alternatives

When only one ε-alternative shall be computed the question comes up which penalty parameter should be used to produce a "best possible" alternative solution. If the penalty parameter is too small the optimal and the alternative solution are too similar and offer no real choice. If the parameter is too large the alternative solution becomes too poor. The best choice is to take some "medium" ε.

We illustrate this effect in the following route planning example.

Example 23.5 Assume that we have to plan a route from a given starting point to a given target. We know the standard travel times of all road sections and are allowed to plan for *two* different routes. In last minute we learn about the real travel times and can choose the fastest of our two candidate routes.

Let the first route be the route with the smallest standard travel time and the second one a route found by the penalty method. Question: Which penalty parameter should we use to minimise the real travel time of the fastest route?

Concretely, consider randomly generated instances of the shortest path problem in a weighted directed grid graph $G = (V, E, w)$ of dimension 25×25. The weights of the arcs are independently uniformly distributed in the unit interval $[0, 1]$. We compute P_0, a path from the lower left corner to the upper right with minimal weight. Afterwards we punish the edges of path P_0 by multiplying by $1 + \varepsilon$ and calculate a whole set of ε-penalty solutions $P_{\varepsilon_1}, P_{\varepsilon_2}, \ldots, P_{\varepsilon_{30}}$ for $\varepsilon = 0.025,\ 0.050,\ \ldots,\ 0.750$. We get 30 solution pairs $\{S_0, S_{\varepsilon_1}\}, \{S_0, S_{\varepsilon_2}\}, \ldots, \{S_0, S_{\varepsilon_{30}}\}$ and can compare these.

The weight $w(e)$ of an arc e is the *standard travel time without time lags*, i.e. the minimal needed travel time on a free road without any traffic jam. The *real* travel time $\hat{w}(e)$ of this arc may differ from $w(e)$ as follows:

$$\hat{w}(e) = \begin{cases} \lambda_c(e) \cdot\ w(e) & : \text{ with probability } p \\ \qquad\quad w(e) & : \text{ with probability } 1-p \end{cases}$$

independently for all edges e. Here the $\lambda_c(e)$ are independent random numbers, uniformly distributed in the interval $[1, c]$. The parameter $0 \leq p \leq 1$ is called **failure probability** and the parameter $c \geq 1$ is called **failure width**.

For every pair $\{S_0, S_{\varepsilon_i}\}$ we calculate the minimum of the two function values $\hat{w}(S_0)$ and $\hat{w}(S_{\varepsilon_i})$. To get a direct impression of the benefit of having two solutions instead of one we scale with respect to the real value of the optimal solution S_0.

$$\phi_{\varepsilon_i} = \frac{\min\{\hat{w}(S_0), \hat{w}(S_{\varepsilon_i})\}}{\hat{w}(S_0)} \quad \text{for } i = 1, \ldots, 30\ .$$

We computed the values ϕ_{ε_i} for 100,000 randomly generated 25×25 grid graphs with failure probability $p = 0.1$ and failure width $c = 8$. Figure 23.3 shows the averages $\overline{\phi}_{\varepsilon_i}$ for $\varepsilon_1 = 0.025,\ \varepsilon_2 = 0.050,\ \ldots,\ \varepsilon_{30} = 0.750$.

As seen in Figure 23.3, the expected quality ϕ_ε of the solution pairs is unimodal in

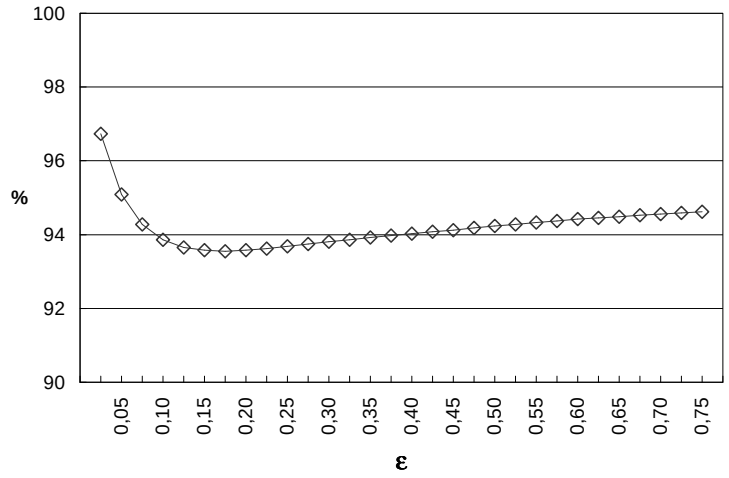


Figure 23.3. $\overline{\phi}_{\varepsilon_i}$ for $\varepsilon_1 = 0.025$, $\varepsilon_2 = 0.050$, ..., $\varepsilon_{30} = 0.750$ on 25×25 grids.

ε. That mean that ϕ_ε first decreases and then increases for growing ε. In this example $\varepsilon^* \approx 0.175$ is the optimal penalty parameter.

In further experiments it was observed that the optimal parameter ε^* is decreasing in the problem size (e.g. $\varepsilon^* \approx 0.6$ for shortest paths in 5×5-grids, $\varepsilon^* \approx 0.175$ for 25×25 and $\varepsilon^* \approx 0.065$ for 100×100 grid graphs).

Monotonicity properties of the penalty solutions

Independently whether all ε-penalty solutions (like on page 1104) are generated or only a single one (like on page 1106), the following structural properties are provable: With increasing penalty factor ε we get solutions B_ε where

- the penalty part p of the objective function is fitted monotonically better (the solution contains less punished parts),
- the original objective function w is getting monotonically worse, in compensation for the improvement with respect to the penalty part.

These facts are formalised in the following theorem.

Theorem 23.3 *Let $w : E \to \mathbb{R}$ be a real-valued function and $p : E \to \mathbb{R}^+$ a positive real-valued function on E. Let B_ε be defined for $\varepsilon \in \mathbb{R}^+$ according to Definition 23.2. The following four statements hold:*
(i) $p(B_\varepsilon)$ is weakly monotonically decreasing in ε.
(ii) $w(B_\varepsilon)$ is weakly monotonically increasing in ε.
(iii) The difference $w(B_\varepsilon) - p(B_\varepsilon)$ is weakly monotonically increasing in ε.
(iv) $w(B_\varepsilon) + \varepsilon \cdot p(B_\varepsilon)$ is weakly monotonically increasing in ε.

Proof Let δ and ε be two arbitrary nonnegative real numbers with $0 \leq \delta < \varepsilon$.

Because of the definition of B_δ and B_ε the following inequalities hold.

(*i*) In case $\varepsilon < \infty$ we have

$$w(B_\varepsilon) + \varepsilon \cdot p(B_\varepsilon) \le w(B_\delta) + \varepsilon \cdot p(B_\delta) \ , \tag{23.1}$$

$$w(B_\varepsilon) + \delta \cdot p(B_\varepsilon) \ge w(B_\delta) + \delta \cdot p(B_\delta) \ . \tag{23.2}$$

Subtracting (23.2) from (23.1) we get

$$(\varepsilon - \delta) \cdot p(B_\varepsilon) \le (\varepsilon - \delta) \cdot p(B_\delta) \qquad | : (\varepsilon - \delta) > 0$$

$$\Leftrightarrow \quad p(B_\varepsilon) \le p(B_\delta) \ . \tag{23.3}$$

In case $\varepsilon = \infty$ inequality (23.3) follows directly from the definition of B_∞ .

(*ii*) Subtracting (23.3) multiplied with δ from (23.2) we get

$$w(B_\varepsilon) \ge w(B_\delta). \tag{23.4}$$

(*iii*) Subtracting (23.3) from (23.4) we get

$$w(B_\varepsilon) - p(B_\varepsilon) \ge w(B_\delta) - p(B_\delta).$$

(*iv*) With (23.2) and $\varepsilon > \delta \ge 0$ we have

$$w(B_\delta) + \delta \cdot p(B_\delta) \le w(B_\varepsilon) + \delta \cdot p(B_\varepsilon) \le w(B_\varepsilon) + \varepsilon \cdot p(B_\varepsilon)$$

$$\Rightarrow \quad w(B_\varepsilon) + \varepsilon \cdot p(B_\varepsilon) \ge w(B_\delta) + \delta \cdot p(B_\delta) \ .$$

Generating more than one alternative solution for the same penalty parameter ε

If we have a solution S_0 and want to get *more than one* alternative solution we can use the penalty method several times by punishing S_0 with different penalty parameters $\varepsilon_1 < \cdots < \varepsilon_m$, getting alternative solutions $S_{\varepsilon_1}, S_{\varepsilon_2}, \ldots, S_{\varepsilon_m}$. This method has a big disadvantage, because only the shared parts of the main solution S_0 and each alternative solution are controlled by the values ε_i. But there is no direct control of the parts shared by two different alternative solutions. So, S_{ε_i} and S_{ε_j} may be rather similar for some $i \neq j$.

To avoid this effect the penalty method may be used *iteratively* for the same ε.

ITERATIVE-PENALTY-METHOD(w, p, k, ε)

1 solve the original problem $\min w(B)$ and find the optimal solution S_0.
2 define the penalty function as $p_1(B) \leftarrow \varepsilon \cdot w(B \cap S_0)$.
3 solve the modified problem $\min w(B) + \varepsilon \cdot p_1(B)$ and find the solution S_1.
4 **for** $j \leftarrow 2$ **to** k
5 **do** $p_j(B) \leftarrow \varepsilon \cdot w(B \cap S_0) + \varepsilon \cdot w(B \cap S_1) + \cdots + \varepsilon \cdot w(B \cap S_{j-1})$
6 solve the modified problem $\min w(B) + \varepsilon \cdot p_j(B)$ and find the solution S_j.
7 **return** $(S_0, S_1, \ldots, S_k)$

Step 5 may be replaced by the variant 5*

5* **do** $p_j(B) \leftarrow \varepsilon \cdot w(B \cap (S_0 \cup S_1 \cup \cdots \cup S_{j-1}))$

In the first case (5) a part of a solution belonging to r of the j solutions $S_0, S_1, \ldots$ and S_{j-1} is punished by the factor $r \cdot \varepsilon$. In the second case (5*) a part of a solution is punished with multiplicity one if it belongs to at least one of $S_0, S_1, \ldots$ or S_{j-1}.

The differences in performance are marginal. However, in shortest path problems with three solutions S_0, S_1 and S_2 setting (5) seemed to give slightly better results.

Example 23.6 Take the graph $G = (V, E)$ from Figure 23.2. For penalty parameter $\varepsilon = 0.1$ we want to find three solutions. The shortest path from S to T is P_D via S - A - C - D - F - T with length 23. Multiplying all edges of P_D by 1.1 and solving the obtained shortest path problem gives the alternative solution P_B via S - A - B - F - T.

Applying setting (5) we have to multiply the edge lengths of (A, C), (C, D), (D, F), (A, B) and (B, F) by penalty factor 1.1. The edges (S, A) and (F, T) have to be multiplied by factor 1.2 (double penalty). The optimal solution is path P_H via S - G - H - T.

Applying setting (5*) we have to multiply the edge lengths (S, A), (A, C), (C, D), (D, F), (F, T), (A, B) and (B, F) by penalty factor 1.1. The optimal solution of this modified problem is path P_E via S - A - C - E - F - T.

23.2.3. The linear programming - penalty method

It is well known that shortest path problems as well as many other network flow problems can be solved with **Linear Programming**. Linear Programming may also be used to generate alternative solutions. We start with the description of Linear Programming for the basic shortest path problem.

The shortest path problem formulated as a linear program

Consider a directed graph $G = (V, E)$ and a function $w : E \to \mathbb{R}^+$ assigning a length to every arc of the graph. Let s and t be two distinguished nodes of G.

Which is the shortest simple path from s to t in G?

For every arc $e = (i, j) \in E$ we introduce a variable x_e. Here x_e shall be 1 if e is part of the shortest path and x_e shall be 0 otherwise.

With $S(i) = \{j \in V : (i, j) \in E\} \subseteq V$ we denote the set of the successors of node i and with $P(i) = \{j \in V : (j, i) \in E\} \subseteq V$ we denote the set of the predecessors of node i. The linear program $LP_{ShortestPath}$ is formulated as follows:

$$\min \sum_{e \in E} w(e) \cdot x_e$$

$$\text{s.t.} \sum_{j \in S(s)} x_{(s,j)} - \sum_{j \in P(s)} x_{(j,s)} = 1 \quad \text{flow-out condition for the starting node } s$$

$$\sum_{j \in S(t)} x_{(t,j)} - \sum_{j \in P(t)} x_{(j,t)} = -1 \quad \text{flow-in condition for the target node } t$$

$$\sum_{j \in S(i)} x_{(i,j)} - \sum_{j \in P(i)} x_{(j,i)} = 0 \quad \text{for all nodes } i \in V \backslash \{s, t\}$$

KIRCHHOFF conditions for all interior nodes

$$0 \leq x_e \leq 1 \text{ for all } e \in E.$$

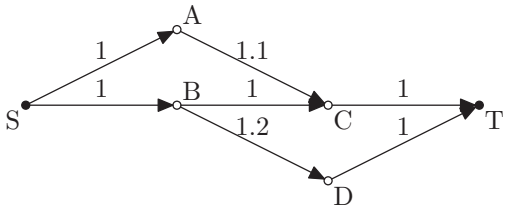


Figure 23.4. Example graph for the LP-penalty method.

By the starting and target conditions node s is a source and node t is a sink. Because of the KIRCHHOFF conditions there are no other sources or sinks. Therefore there must be a "connection" from s to t.

It is not obvious that this connection is a simple path. The variables x_e might have non-integer values or there could be circles anywhere. But there is a basic theorem for network flow problems [7, p. 318] that the linear program $LP_{ShortestPath}$ has an optimal solution where all $x_e > 0$ have the value 1. The according arcs with $x_e = 1$ represent a simple path from s to t.

Example 23.7 Consider the graph in Figure 23.4. The linear program for the shortest path problem in this graph contains six equality constraints (one for each node) and seven pairs of inequality constraints (one pair for each arc).

$$
\begin{aligned}
&\min(x_{SA} + x_{SB} + x_{BC} + x_{CT} + x_{DT}) \cdot 1 + x_{AC} \cdot 1.1 + x_{BD} \cdot 1.2 \\
&\text{s.t. } x_{SA} + x_{SB} = 1, \\
&\quad x_{CT} + x_{DT} = 1\ , \\
&\quad x_{SA} - x_{AC} = 0, \\
&\quad x_{SB} - x_{BC} - x_{BD} = 0\ , \\
&\quad x_{AC} + x_{BC} - x_{CT} = 0, \\
&\quad x_{BD} - x_{DT} = 0\ , \\
&\quad 0 \le x_{SA}, x_{SB}, x_{AC}, x_{BC}, x_{BD}, x_{CT}, x_{DT} \le 1\ .
\end{aligned}
$$

The optimal solution has $x_{SB} = x_{BC} = x_{CT} = 1$.

A linear program which gives two alternative paths from s to t

Here we give an LP-representation for the task to find *two* alternative routes from s to t.

For every arc $e = (i, j) \in E$ we introduce two variables x_e and y_e. If the arc e is used in both routes, then both x_e and y_e shall have the value 1. If e is a part of only one route, x_e shall be 1 and y_e shall be 0. Otherwise x_e and y_e shall both be 0. $\varepsilon > 0$ is a penalty parameter to punish arcs used by both routes.

With this in mind we can formulate the linear program $LP_{2-ShortPaths}$

$$\min f(x,y) := \sum_{e\in E} w(e)\cdot x_e + (1+\varepsilon)\cdot w(e)\cdot y_e$$

$$\text{s.t.} \sum_{j\in S(s)} x_{(s,j)} + y_{(s,j)} - \sum_{j\in P(s)} x_{(j,s)} + y_{(j,s)} = 2 \quad \text{condition for the starting node } s$$

$$\sum_{j\in S(t)} x_{(t,j)} + y_{(t,j)} - \sum_{j\in P(t)} x_{(j,t)} + y_{(j,t)} = -2 \quad \text{condition for the target node } t$$

$$\sum_{j\in S(i)} x_{(i,j)} + y_{(i,j)} - \sum_{j\in P(i)} x_{(j,i)} + y_{(j,i)} = 0 \quad \text{KIRCHHOFF conditions}$$

$$\text{for all } i \in V\backslash\{s,t\}$$

$$0 \le x_e, y_e \le 1 \text{ for all } e \in E\ .$$

Example 23.8 Consider again the graph in Figure 23.4. The linear program for the 2-alternative-paths problem in this graph contains six equality constraints (one for each node) and $2\cdot 7 = 14$ pairs of inequality constraints.

$$\begin{aligned}
\min\quad & (x_{SA}+x_{SB}+x_{BC}+x_{CT}+x_{DT})\cdot 1 + x_{AC}\cdot 1.1 + x_{BD}\cdot 1.2 \\
& + (y_{SA}+y_{SB}+y_{BC}+y_{CT}+y_{DT})\cdot 1 + y_{AC}\cdot 1.1 + y_{BD}\cdot 1.2 \ \cdot(1+\varepsilon) \\
\text{s.t. } & x_{SA}+y_{SA}+x_{SB}+y_{SB} = 2, \\
& x_{CT}+y_{CT}+x_{DT}+y_{DT} = 2\ , \\
& x_{SA}+y_{SA}-x_{AC}-y_{AC} = 0, \\
& x_{SB}+y_{SB}-x_{BC}-y_{BC}-x_{BD}-y_{BD} = 0\ , \\
& x_{AC}+y_{AC}+x_{BC}+y_{BC}-x_{CT}-y_{CT} = 0, \\
& x_{BD}+y_{BD}-x_{DT}-y_{DT} = 0\ , \\
& 0 \le x_{SA}, x_{SB}, x_{AC}, x_{BC}, x_{BD}, x_{CT}, x_{DT}, y_{SA}, y_{SB}, y_{AC}, y_{BC}, y_{BD}, y_{CT}, y_{DT} \le 1\ .
\end{aligned}$$

This linear program can be interpreted as a minimal cost flow problem.

Where is the connection between the linear program and the problem to find two candidate routes from s to t?

Theorem 23.4 *If the linear program $LP_{2-ShortPaths}$ has an optimal solution then it has also an optimal solution (x,y) with the following properties.*
There are disjoint sets $E_1, E_2, E_3 \subseteq E$ with
(i) $E_1 \cap E_2 = \emptyset$, $E_1 \cap E_3 = \emptyset$ and $E_2 \cap E_3 = \emptyset$,
(ii) $x_e = 1$, $y_e = 0$ for all $e \in E_1 \cup E_2$,
(iii) $x_e = 1$, $y_e = 1$ for all $e \in E_3$,
(iv) $x_e = 0$, $y_e = 0$ for all $e \notin E_1 \cup E_2 \cup E_3$.
(v) $E_1 \cup E_3$ represents a path P_1 from s to t and $E_2 \cup E_3$ represents a path P_2 from s to t. E_3 is the set of arcs used by both paths.

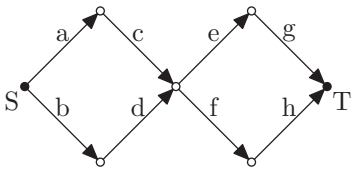


Figure 23.5. An example for a non-unique decomposition in two paths.

(vi) No other pair (Q_1, Q_2) *of paths is better than* (P_1, P_2)*, i.e.,*

$$w(P_1) + w(P_2) + \varepsilon \cdot w(P_1 \cap P_2) \leq w(Q_1) + w(Q_2) + \varepsilon \cdot w(Q_1 \cap Q_2) \ , \quad \textit{for all pairs } (Q_1, Q_2) \ .$$

That means the sum of the lengths of P_1 *and* P_2 *plus a penalty for arcs used twice is minimal.*

We conclude with some remarks.

- For each arc e there are two variables x_e and y_e. This can be interpreted as a street with a **normal lane** and an additional **passing lane**. Using the passing lane is more expensive than using the normal lane. If a solution uses an arc only once, it takes the cheaper normal lane. But if a solution uses an arc twice, it has to take both the normal and the passing lane.

- The decomposition of the solution (x, y) into two paths from the starting node to the target node is in most cases not unique. With the arcs $a, b, \ldots, g, h$ in Figure 23.5 we can build two different pairs of paths from S to T, namely $(a - c - e - g,\ b - d - f - h)$ and $(a - c - f - h,\ b - d - e - g)$. Both pairs are equi-optimal in the sense of Theorem 23.4. So the user has the chance to choose between them according to other criteria.

- The penalty method and the LP-penalty method generally lead to different results. The penalty method computes the best single solution and a suitable alternative. The LP-penalty method computes a pair of good solutions with relatively small overlap. Figure 23.4 shows that this pair not necessarily contains the best single solution. The shortest path from S to T is $P_1 = S$–B–C–T with length 3. For all $\varepsilon > 0.1$ the ε-penalty solution is $P_2 = S$–A–C–T. The path pair (P_1, P_2) has a total lengths of 6.1 and a shared length of 1.0. But for $\varepsilon > 0.2$ the LP-Penalty method produces the pair $(P_2, P_3) = (S$–A–C–T, S–B–D–$T)$ with a total length of 6.3 and a shared length of zero.

- Finding k candidate routes for some larger number $k > 2$ is possible, if we introduce k variables $x_e^0, x_e^1, \ldots, x_e^{k-1}$ for each arc e and set the supply of s and

the demand of t to k. As objective function we can use for instance

$$\min f(x^0, \ldots, x^{k-1}) := \sum_{e \in E} \sum_{j=0}^{k-1} (1 + j \cdot \varepsilon) \cdot w(e) \cdot x_e^j$$

or

$$\min f(x^0, \ldots, x^{k-1}) := \sum_{e \in E} \sum_{j=0}^{k-1} (1 + \varepsilon)^j \cdot w(e) \cdot x_e^j \ .$$

- The LP-penalty method does not only work for shortest path problems. It can be generalised to arbitrary problems solvable by linear programming.
- Furthermore an analogous method – the *Integer Linear Programming Penalty Method* – can be applied to problems solvable by integer linear programming.

23.2.4. Penalty method with heuristics

In Subsection 23.2.2 we discussed the penalty method in combination with exact solving algorithms (e.g. Dijkstra-algorithm or dynamic programming for the shortest path problem). But also in case of heuristics (instead of exact algorithms) the penalty method can be used to find multiple candidate solutions.

Example 23.9 A well known heuristic for the TSP-problem is local search with 2-exchange steps (cp. Subsection 23.2.1).

PENALTY-METHOD-FOR-THE-TSP-PROBLEM-WITH-2-EXCHANGE-HEURISTIC

1 apply the 2-exchange heuristic to the unpunished problem getting a locally (but not necessarily globally) optimal solution T
2 punish the edges belonging to T by multiplying their lengths with $1 + \varepsilon$
3 apply the 2-exchange heuristic to the punished problem getting an alternative solution T_ε
4 compute the unmodified length of T_ε
5 the pair (T, T_ε) is the output

Question: Which penalty parameter $\varepsilon \geq 0$ should be used to minimise the travel time of the fastest route?

An experiment analogous to the one described in Example 23.5 was executed for TSP instances with 25 random cities in the unit square. Figure 23.6 shows the scaled averages for $\varepsilon_0 = 0.000$, $\varepsilon_1 = 0.025$, ..., $\varepsilon_{30} = 0.750$. So, the expected quality ϕ_ε of the solution pairs is (again) unimodal in the penalty factor ε. That means that ϕ_ε first decreases and then increases for growing ε. In this example $\varepsilon^* \approx 0.175$ is the optimal penalty parameter.

In further experiments it was observed that the optimal penalty parameter ε^* is decreasing in the problem size.

Exercises

23.2-1 The following programming exercise on the Travelling Salesperson Problem helps to get a feeling for the great variety of local optima. Generate 200 random

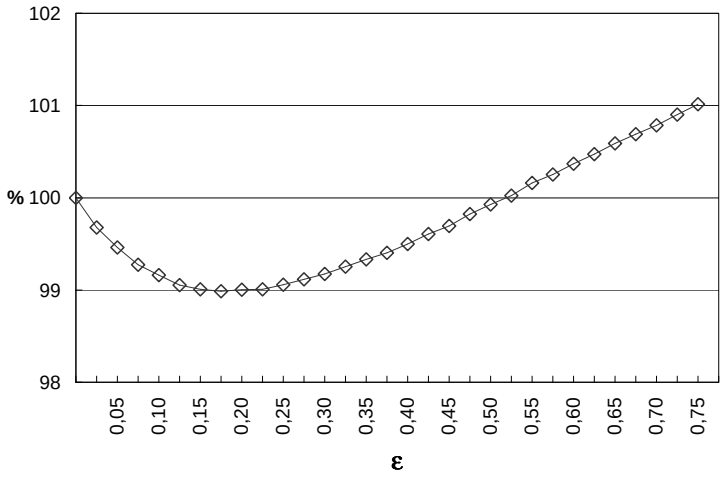


Figure 23.6. $\overline{\phi}_{\varepsilon_i}$ for $\varepsilon_0 = 0$, $\varepsilon_1 = 0.025$, ..., $\varepsilon_{30} = 0.750$ on 25×25 grids.

points in the 2-dimensional unit-square. Compute distances with respect to the Euclidean metric. Make 100 runs of local search with random starting tours and 2-exchanges. Count how many different local minima have been found.

23.2-2 Enter the same key words into different internet search engines. Compare the hit lists and their diversities.

23.2-3 Formulate the Travelling Salesperson Problem as a $\sum$-type problem.

23.2-4 Proof the assertion of the remark on page 1104.

23.2-5 How does the penalty function $p(e)$ look like in case of additive penalties like in Example 23.2?

23.2-6 Prove the properties (1) and (2) on page 1105.

23.2-7 Apply the DIVIDE AND COVER algorithm (page 1105) to the shortest path problem in Figure 23.2 with starting node S and target node T. Set $w(e) =$ length of e for each road section, and $p(e) =$ length of e for the road sections belonging to the shortest path P_D via S - A - C - D - F - T and $p(e) = 0$ for all other sections. So, the penalty value of a whole path is the length of this part shared with P_D.

23.2-8 Find a penalty parameter $\varepsilon > 0$ for Example 23.6 such that the first setting (5) produces three different paths but the second setting (5*) only two different paths for $k = 3$.

23.3. More algorithms for interactive problem solving

There are many other settings where a human controller has access to computer-generated candidate solutions. This section lists four important cases and concludes with a discussion of miscellaneous stuff.

23.3.1. Anytime algorithms

In an anytime-setting the computer starts to work on a problem, and almost from the very first moment on candidate solutions (the best ones found so far) are shown on the monitor. Of course, the early outputs in such a process are often only preliminary and approximate solutions – without guarantee of optimality and far from perfect.

An example: Iterative deepening performs multiple depth-limited searches – gradually increasing the depth limit on each iteration of the search. Assume that the task is to seek good solutions in a large rooted tree $T = (V, E)$. Let $f : V \to \mathbb{R}$ be the function which is to be maximised. Let V_d be the set of all nodes in the tree at distance d from root.

ITERATIVE-DEEPENING-TREE-SEARCH(T, f)

```
1  Opt ← f(root)
2  d ← 1
3  while d < ∞
4        do Determine maximum Max_d of f on V_d
5         if Max_d > Opt
6            then Opt ← Max_d
7         d ← d + 1
```

All the time the currently best solution (Opt) is shown on the monitor. The operator may stop at any moment.

Iterative deepening is not only interesting for HCI but has also many applications in fully automatic computing. A prominent example is game tree search: In tournament chess a program has a fixed amount of time for 40 moves, and iterative deepening is the key instrument to find a balanced distribution of time on the single alpha-beta searches.

Another frequent anytime scenario is repeated application of a heuristic. Let $f : A \to \mathbb{R}$ be some complicated function for which elements with large function values are searched. Let H be a probabilistic heuristic that returns a candidate solution for this maximisation problem (A, f). For instance, H may be local search or some other sort of hill-climbing procedure. H is applied again and again in independent runs, and all the time the best solution found so far is shown.

A third anytime application is in Monte Carlo simulations, for instance in Monte Carlo integration. A static approach would take objective values at a prescribed number of random points (1,000 or so) and give the average value in the output. However, already the intermediate average values (after 1, 2, 3 etc. data points – or after each block of 10 or 50 points) may give early indications in which region the final result might fall and whether it really makes sense to execute all the many runs. Additional display of variances and frequencies of outliers gives further information for the decision when best to stop the Monte Carlo run.

In human-computer systems anytime algorithms help also in the following way: during the ongoing process of computing the human may already evaluate and compare preliminary candidate solutions.

23.3.2. Interactive evolution and generative design

Genetic Algorithms are search algorithms based on the mechanics of natural selection and natural genetics. Instead of single solutions whole populations of solutions are manipulated. Genetic Algorithms are often applied to large and difficult problems where traditional optimisation techniques fall short.

Interactive evolution is an evolutionary algorithm that needs human interaction. In interactive evolution, the user selects one or more individual(s) of the current population which survive(s) and reproduce(s) (with mutations) to constitute the new generation. So, in interactive evolution the user plays the role of an objective function and thus has a rather active role in the search process.

In fields like art, architecture, and photo processing (including the design of phantom photos) *Generative Design* is used as a special form of interactive evolution. In Generative Design *all* solutions of the current generation are shown simultaneously on the screen. Here typically "all" means some small number N between 4 and 20. Think of photo processing as an example, where the user selects modified contrast, brightness, colour intensities, and sharpness. The user inspects the current candidate realizations, and by a single mouse click marks the one which he likes most. All other solutions are deleted, and N mutants of the marked one are generated. The process is repeated (open end) until the user is happy with the outcome. For people without practical experience in generative design it may sound unbelievable, but even from poor starting solutions it takes the process often only a few iterations to come to acceptable outcomes.

23.3.3. Successive fixing

Many problems are high-dimensional, having lots of parameters to adjust. If sets of good solutions in such a problem are generated by repeated probabilistic heuristics, the following interactive multi-stage procedure may be applied: First of all several heuristic solutions are generated and inspected by a human expert. This human especially looks for "typical" pattern in the solutions and "fixes" them. Then more heuristic solutions are generated under the side condition that they all contain the fixed parts. The human inspects again and fixes more parts. The process is repeated until finally everything is fix, resulting in one specific (and hopefully good) solution.

23.3.4. Interactive multicriteria decision making

In multicriteria decision making not only one but two or more objective functions are given. The task is to find admissible solutions which are as good as possible with respect to all these objectives. Typically, the objectives are more or less contradictory, excluding the existence of a unanimous optimum. Helpful is the concept of "efficient solutions", with the following definition: For an efficient solution there exists no other solution which is better with respect to at least one objective and not worse with respect to all the others.

A standard first step in multicriteria decision making is to compute the set of all efficient solutions. In the bicriteria case the "efficient frontier" can be visualized in a 2-dimensional diagram, giving the human controller a good overview of what is

possible.

23.3.5. Miscellaneous

- *Graphical Visualisation of Computer Solutions*
  It is not enough that a computer generates good candidate solutions. The results also have to be visualized in appropriate ways. In case of a single solution important parts and features have to be highlighted. And, even more important, in case of concurring solutions their differences and specialities have to be stressed.

- *Permanent Computer Runs with Short Intermediate Human Control*
  A nickname for this is "1+23h mode", coming from the following picture: Each day the human sits in front of the computer for one hour only. In this hour he looks at the computer results from the previous 23 hours, interacts with the machine and also briefs the computer what to do in the next 23 hours. So, the human invests only a small portion of his time while the computer is running permanently.

  An impressive example comes from correspondence chess. Computer help is officially permitted. Most top players have one or several machines running all around the clock, analysing the most critical positions and lines of play. The human players collect these computer results and analyse only shortly per day.

- *Unexpected Errors and Numerical Instabilities*
  "Every software has errors!" This rule of thumb is often forgotten. People too often simply believe what the monitor or the description of a software product promises. However, running independent programs for the very same task (with a unique optimal solution) will result in different outputs unexpectedly often. Also numerical stability is not for free. Different programs for the same problem may lead to different results, due to rounding errors. Such problems may be recognised by applying independent programs.

  Of course, also hardware has (physical) errors, especially in times of ongoing miniaturisation. So, in crucial situations it is a good strategy to run an identical program on fully independent machines - best of all operated by independent human operators.

Exercises

23.3-1 For a Travelling Salesperson Problem with 200 random points (x_i, y_i) in the unit square $[0,1] \times [0,1]$ and Euclidean distances, generate 100 locally optimal solutions (with 2-exchanges, see Subsection 23.2.1) and count which edges occur how often in these 100 solutions. Define some threshold K (for instance $K = 30$) and fix all edges which are in at least K of the solutions. Generate another 100 local optima, without allowing the fixed edges to be exchanged. Repeat until convergence and compare the final result with typical local optima from the first series.

Chapter notes

In the technical report [292] lots of experiments on the penalty method for various sum type problems, dimensions, failure widths and probabilities are described and analysed. The proof of Theorem 23.3 was originally given in [13]) . In e-commerce multiple-choice systems are often called "Recommender Systems" [284], having in mind customers for whom interesting products have to be listed. Understandably, commercial search engines and e-companies keep their shortlisting strategies secret.

A good class book on Genetic Algorithms is [133]. Interactive Evolution and Generative Design are described in [31]. There is a lot of literature on multicriteria decision making, one of the standard books being [123].

In the book [10] the story of 3-Hirn and its successes in tournament chess is told. The final match between "3-Hirn" and GM Yussupov is described in [11]. [12] gives more general information on improved game play by multiple computer hints. In [14] several good k-best realizations of iterative deepening in game tree search are exhibited and discussed. Screenshots of these realizations can be inspected at http://www.minet.uni-jena.de/www/fakultaet/iam/personen/k-best.html. [161] describes the technical background of advanced programs for playing chess and other games.

There is a nice online repository, run by M. Zuker and D.H. Turner at http://www.bioinfo.rpi.edu/applications/mfold/. The user may enter for instance RNA-strings, and in realtime alternative foldings for these strings are generated. Amongst other data the user may enter parameters for "maximum number of computed foldings" (default = 50) and "percent suboptimality number" (default = 5 %).

Bibliography

[1] S. Abiteboul. Querying semi-structured data. In F. Afrati, P. Kolaitis (editors), *Proceedings of ICDT'97*, Lecture Notes in Computer Science. Volume 1186, pages 1–18, Springer-Verlag, 1997. 930

[2] S. Abiteboul, O. Duschka. Complexity of answering queries using materialized views. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. pages 254–263. ACM-Press, 1998. 930

[3] S. Abiteboul, V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995. 881

[4] L. Addario-Berry, B. Chor, M. Hallett, J. Lagergren, A. Panconesi, T. Wareham. Ancestral maximum likelihood of phylogenetic trees is hard. *Lecture Notes in Bioinformatics*, 2812:202–215, 2003. 1010

[5] R. G. Addie, M. Zukerman, T. Neame. Broadband traffic modeling: Simple solutions to hard problems. *IEEE Communications Magazine*, 36(8):88–95, 1998. 701

[6] A. Aho, C. Beeri, J. D. Ullman. The theory of joins in relational databases. *ACM Transactions on Database Systems*, 4(3):297–314, 1979. 881

[7] R. K. Ahuja, T. L. Magnanti, J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993. 1110

[8] T. Akutsu. Dynamic programming algorithms for RNA secondary prediction with pseudoknots. *Discrete Applied Mathematics*, 104:45–62, 2000. 1010

[9] E. Althaus, A. Caprara, H. Lenhof, K. Reinert. Multiple sequence alignment with arbitrary gap costs: Computing an optimal solution using polyhedral combinatorics. *Bioinformatics*, 18:S4–S16, 2002. 1009

[10] I. Althöfer. *13 Jahre 3-Hirn*. Published by the author, 1998. 1118

[11] I. Althöfer. List-3-Hirn vs. grandmaster Yussupov – report on a very experimental match. *ICCA Journal*, 21:52–60 and 131–134, 1998. 1118

[12] I. Althöfer. Improved game play by multiple computer hints. *Theoretical Computer Science*, 313:315–324, 2004. 1118

[13] I. Althöfer, F. Berger, S. Schwarz. Generating True Alternatives with a Penalty Method. http://www.minet.uni-jena.de/Math-Net/reports/shadows/02-04report.html, 2002. 1118

[14] I. Althöfer, J. de Koning, J. Lieberum, S. Meyer-Kahlen, T. Rolle, J. Sameith. Five visualisations of the k-best mode. *ICCA Journal*, 26:182–189, 2003. 1118

[15] G. M. Amdahl. Validity of the single-processor approach to achieving large-scale computer capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485, 1967. 750

[16] AND Software GmbH. Car routing program *Route Germany*, 1997. 1102

[17] D. Anick, D. Mitra, M. Sondhi. Stochastic theory of a data handling system with multiple sources. *The Bell System Technical Journal*, 61:1871–1894, 1982. 701

[18] M. Arenas, L. Libkin. A normal form for XML documents. In *Proceedings of the 21st Symposium on Principles of Database Systems*, 2002, pages 85–96. 971

[19] V. Arlazanovan. A. Dinic, M. Kronrod, I. Faradzev. On economic construction of the transitive closure of a directed graph. *Doklady Academii Nauk SSSR*, 194:487–488, 1970. 1009

[20] W. Armstrong. Dependency structures of database relationships. In *Proceedings of IFIP Congress*. pages 580–583. North Holland, 1974. 881

[21] J. Aspnes, C. Busch, S. Dolev, F. Panagiota, C. Georgiou, A. Shvartsman, P. Spirakis, R. Wattenhofer. Eight open problems in distributed computing. *Bulletin of European Association of Theoretical Computer Science of EATCS*, 90:109–126, 2006. 642

[22] K. Atteson. The performance of the neighbor-joining method of phylogeny reconstruction. *Algorithmica*, 25(2/3):251–278, 1999. 1012

[23] H. Attiya, C. Dwork, N. A. Lynch, L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41:122–142, 1994. 641

[24] H. Attiya, J. Welch. *Distributed Computing, Fundamentals, Simulations and Advanced Topics.* McGraw-Hill, 1998. 641, 642

[25] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985. 641

[26] D. Bader, B. Moret, M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001. 1007

[27] M. Bader, E. Ohlebusch. Sorting by weighted reversals, transpositions and inverted transpositions. *Lecture Notes in Bioinformatics*, 3909:563–577, 2006. 1006

[28] B. Baker. A tight asymptotic bound for next-fit decreasing bin-packing. *SIAM Journal on Algebraic and Discrete Methods*, 2(2):147–152, 1981. 848

[29] Á. Balogh, A. Iványi. Memory management. In A. Iványi (editor), *Algorithms of Informatics, Volume 2.* pages 799-850. Mondat Kiadó, 2007. 750

[30] J. Banks, J. Carson, B. Nelson. *Discrete-Event Simulation.* Prentice Hall, 1996. 700

[31] W. Banzhaf. Interactive evolution. In T. Back, D. B. Fogel, Z. Michalewicz, T. Baeck Baeck (editors), *Handbook of Evolutionary Computation.* IOP Press, 1997. 1118

[32] C. Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Transactions on Database Systems*, 5:241–259, 1980. 881

[33] C. Beeri, P. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, 4(1):30–59, 1979. 881

[34] C. Beeri, M. Dowd. On the structure of armstrong relations for functional dependencies. *Journal of ACM*, 31(1):30–46, 1984. 881

[35] C. Beeri, R. Fagin, J. Howard. A complete axiomatization for functional and multivalued dependencies. In *ACM SIGMOD Symposium on the Management of Data*, pages 47–61, 1977. 881

[36] R. Bello, K. Dias, A. Downing, J. Freenan, T. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, M. Ziauddin. Materialized views in Oracle. In *Proceedings of Very Large Data Bases'98*, pages 659–664, 1998. 930

[37] S. A. Benner, M. A. Cohen, H. G. H. Gonnet. Empirical and structural models for insertions and deletions in the divergent evolution of proteins. *Journal of Molecular Biology*, 229(4):1065–1082, 1993. 977, 978

[38] J. Beran. *Statistics for Long-Memory Processes.* Monographs on Statistics and Applied Probability. Chapman & Hall, 1986. 701

[39] J. Beran, R. Sherman, M. Taqqu, W. Willinger. Long-range dependence in variable-bit-rate video traffic. *IEEE Transactions on Communications*, 43:1566–1579, 1995. 701

[40] F. Berger. k alternatives instead of k shortest paths. Master's thesis, Friedrich Schiller University, Jena, Faculty of Mathematcs and Informatics, 2000. Master thesis. 1102

[41] P. Berman, J. Garay. Cloture votes: $n/4$-resilient distributed consensus in $t+1$ rounds. *Mathematical Systems Theory*, 26(1):3–19, 1993. 642

[42] K. A. Berman, J. L. Paul. *Sequential and Parallel Algorithms.* PWS Publishing Company, 1996. 749

[43] A. Békéssy, J. Demetrovics. Contribution to the theory of data base relations. *Discrete Mathematics*, 27(1):1–10, 1979. 881

[44] L. A. Bélády, R. Nelson, G. S. Shedler. An anomaly in space-time characteristics of certain programs running in paging machine. *Communications of the ACM*, 12(1):349–353, 1969. 847

[45] J. Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):135–256, 1982. 1092

[46] J. Bloomenthal. *Introduction to Implicit Surfaces.* Morgan Kaufmann Publishers, 1997. 1092

[47] A. Blum, T. Yiang, M. Li, J. Tromp, M. Yannakis. Linear approximation of shortest superstrings. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, 328–336. pages, 1991. 1008

[48] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, 1974. 750

[49] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965. 1093

[50] P. Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM-Press, 1997, pages 117–121. 930

[51] P. Buneman, M. Fernandez, D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *The International Journal on Very Large Data Bases*, 9(1):76–110, 2000. 971

[52] J. E. Burns. A formal model for message passing systems. Technical Report91, Indiana University, 1980. 641

[53] J. E. Burns, N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993. 642

[54] CACI. *COMNET III*. CACI Products Co., 1997. 700

[55] D. Calvanese, D. De Giacomo, M. Lenzerini, M. Varni. Answering regular path queries using views. In *Proceedings of the Sixteenth International Conference on Data Engineering*, pages 190–200. 2000. 930

[56] A. Caprara,. Formulations and complexity of multiple sorting by reversals. In *RECOMB-99*, 84–93. pages, 1999. ACM-Press. 1006

[57] A. Caprara. Sorting permutations by reversals and eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110, 1999. 1006

[58] H. Carillo, D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48:1073–1082, 1988. 985

[59] E. Catmull, J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10:350–355, 1978. 1092

[60] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000. 750

[61] S. Chaudhury, R. Krishnamurty, S. Potomianos, K. Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 190–200. 1995. 930

[62] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6(5):353–363, 1991. 1092

[63] Q. Chen, A. Lim, K. W. Ong. An adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 134–144. 2003. 971

[64] B. Chor, T. Tuller. Maximum likelihood of evolutionary trees: hardness and approximation. *Bioinformatics*, 21:i97–i106, 2005. 1010

[65] D. Christie. Sorting permutations by block-interchanges. *Information Processing Letters*, 60(4):165–169, 1996. 1012

[66] E. F. Codd. A relational model of large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. 881

[67] E. F. Codd. Normalized database structure: A brief tutorial. In *ACM SIGFIDET Workshop on Data Description, Access and Control*, 1971, pages 24–30. 881

[68] E. F. Codd. Further normalization of the data base relational model. In R. Rustin (editor), *Courant Computer Science Symposium 6: Data Base Systems*. Prentice Hall, 1972, pages 33–64. 881

[69] E. F. Codd. Relational completeness of database sublanguages. In R. Rustin (editor), *Courant Computer Science Symposium 6: Data Base Systems*. Prentice Hall, 1972, pages 65–98. 881

[70] E. F. Codd. Recent investigations in relational data base systems. In *Information Processing 74*. North-Holland, 1974, pages 1017–1021. 881

[71] E. Coffman. *Computer and Job Shop Scheduling*. John Wiley & Sons, 1976. 847

[72] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990. 749

[73] 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). 750, 848

[74] D. G. Corneil, C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17(1):51–64, 1970. 971

[75] F. Corpet. Multiple sequence alignment with hierarchical clustering. *Nucleic Acids Research*, 16:10881–10890, 1988. 981

[76] H. S. M. Coxeter. *Projective Geometry.* University of Toronto Press, 1974 (2. edition). 1091

[77] M. Crovella, A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997. 701

[78] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, T. von Eicken. LogP: A practical model of parallel computation. *Communication of the ACM*, 39(11):78–85, 1996. 750

[79] A. Darte, Y. Robert, F. Vivien. *Scheduling and Automatic Parallelization.* Birkhäuser Boston, 2000. 794

[80] M. O. Dayhoff, R. M. Schwartz, B. Orcutt. A model of evolutionary change in proteins. *Atlas of Protein Sequence and Structure*, 5:345–352, 1978. 978

[81] M. de Berg. *Efficient Algorithms for Ray Shooting and Hidden Surface Removal.* PhD thesis, Rijksuniversiteit te Utrecht, 1992. 1092

[82] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 2000. 1092

[83] C. Delobel. Normalization and hierarchical dependencies in the relational data model. *ACM Transactions on Database Systems*, 3(3):201–222, 1978. 881

[84] J. Demetrovics, Gy. O. H. Katona, A. Sali. Minimal representations of branching dependencies. *Discrete Applied Mathematics*, 40:139–153, 1992. 881

[85] J. Demetrovics, Gy. O. H. Katona, A. Sali. Minimal representations of branching dependencies. *Acta Scientiarum Mathematicorum (Szeged)*, 60:213–223, 1995. 881

[86] J. Demetrovics, Gy. O. H. Katona, A. Sali. Design type problems motivated by database theory. *Journal of Statistical Planning and Inference*, 72:149–164, 1998. 881

[87] P. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, 1970. 847

[88] A. Deutsch, M. Fernandez, D. Suciu. Storing semistructured data with stored. In *Proceedings of SIGMOD'99*, pages 431–442, 1999. 930

[89] A. Deutsch, L. Popa, D. Tannen. Physical data independence, constraints and optimization with universal plans. In *Proceedings of VLDB'99*, 459–470. pages, 1999. 930

[90] D. Dolev, R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983. 642

[91] N. Duffield, N. Oconnell. Large deviation and overflow probabilities for the general single-server queue, with applications. *Mathematical Proceedings of the Cambridge Philosophical Society*, 118:363–374, 1995. 701

[92] D. Duffy, A. McIntosh, M. Rosenstein, W. Willinger. Statistical analysis of ccsn/ss7 traffic data from working ccs subnetworks. *IEEE Journal on Selected Areas Communications*, 12:544–551, 1994. 701

[93] O. Duschka, M. Genesereth. Query planning in infomaster. In *Proceedings of ACM Symposium on Applied Computing.* ACM-Press, 1997, pages 109–111. 930

[94] O. Duschka, M. Genesereth. Answering recursive queries using views. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* ACM-Press, pages 109–116, 1997. 930

[95] N. Dyn, J. Gregory, D. Levin. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9:160–169, 1990. 1092

[96] I. Elias, T. Hartman. A 1.375 approximation algorithm for sorting by transpositions. *Lecture Notes in Bioinformatics*, 3692:204–215, 2005. 1006

[97] I. Elias, J. Lagergren, J.Fast neighbor joining. *Lecture Notes in Computer Science*, 3580:1263–1274, 2005. 1012

[98] P. Erdős, M. Steel, L. Székely, T. Warnow. Local quartet splits of a binary tree infer all quartet splits via one dyadic inference rule. *Computers and Artificial Intelligence*, 16(2):217–227, 1997. 1012

[99] A. Erramilli, O. Narayan, W. Willinger. Experimental queueing analysis with long-range dependent packet-traffic. *IEEE/ACM Transactions on Networking*, 4(2):209–223, 1996. 701

[100] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems*, 2:262–278, 1977. 881

[101] R. Fagin. Horn clauses and database dependencies. *Journal of ACM*, 29(4):952–985, 1982. 881

[102] R. Fagin. Armstrong databases. In *Proceedings of IBM Symposium on Mathematical Foundations of Computer Science*, 1982, 24 pages. 881

[103] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Morgan Kaufmann Publishers, 2002 (2. revised edition). 1092

[104] J. Felsenstein. Evolutionary trees from DNA sequences: a maximum likelihood approach. *Journal of Molecular Evolution*, 17:368–376, 1981. 987

[105] D. Feng, R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25:351–360, 1987. 981

[106] R. Fernando. *GPUGems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, 2004. 1093

[107] J.W. Fickett. Fast optimal alignment. *Nucleid Acids Research*, 12:175–180, 1984. 984

[108] M. J. Fischer, N. A. Lynch, M. S. Paterson. Impossibility of distributed consensus with one faulty proces. *Journal of the ACM*, 32(2):374–382, 1985. 642

[109] W. M. Fitch. Toward defining the course of evolution: minimum change for a specified tree topology. *Systematic Zoology*, 20:406–416, 1971. 988

[110] D. Florescu, A. Halevy, A. O. Mendelzon. Database techniques for the world-wide web: a survey. *SIGMOD Record*, 27(3):59–74, 1998. 930

[111] D. Florescu, L. Raschid, P. Valduriez. Answering queries using oql view expressions. In *Workshop on Materialized views, in cooperation with ACM SIGMOD*, 1996, pages 627–638. 930

[112] D. D. Florescu. *Search spaces for object-oriented query optimization*. PhD thesis, University of Paris VI, 1996. 930

[113] M. J. Flynn. Very high-speed computer systems. *Proceedings of the IEEE*, 5(6):1901–1909, 1966. 749

[114] J. D. Fooley, A., S. K. Feiner, J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990. 1092

[115] P. Fornai, A. Iványi. Bélády's anomaly is unbounded. In E. Kovács, Z. Winkler (editors), *5th International Conference on Applied Informatics*. Molnár és társa, 2002, pages 65–72. 847

[116] S. Fortune, J. Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978. 750

[117] I. Foster, C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufman Publisher, 2004 (2. edition). 750

[118] I. Foster, C. Kesselman, J. M. Nick, S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Available at www.globus.org/research/papers/ogsa.pdf, 2002. 750

[119] L. Foulds, R. Graham. The Steiner problem in phylogeny is NP-complete. *Advances in Applied Mathematics*, 3:43–49, 1982. 986

[120] M. Friedman, D. S. Weld. Efficient execution of information gathering plans. In *Proceedings International Joint Conference on Artificial Intelligence*, pages 785–791. 1997. 930

[121] H. Fuchs, Z. M. Kedem, B. F. Naylor. On visible surface generation by a priory tree structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, 1980, pages 124–133. 1092

[122] A. Fujimoto, T. Takayuki, I. Kansey. ARTS: accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986. 1092

[123] T. Gal, T. Stewart, T. Hanne (editors). *Multicriteria Decision Making*. Kluwer Academic Publisher, 1999. 1118

[124] Z. Galil, R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989. 977

[125] J. Gallant, D. Maier, J. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980. 1008

[126] W. Gararch, E. Evan, C. Kruskal. Proofs that yield nothing but their validity or all languages in NP. *Journal of the ACM*, 38(3):691–729, 1991. 750

[127] H. Garcia-Molina, J. Seiferas. Elections in a distributed computing systems. *IEEE Transactions on Computers*, C-31(1):47–59, 1982. 641

[128] M. R. Garey, D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 971

[129] W. H. Gates, C. H. Papadimitriou. Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27:47–57, 1979. 1012

[130] F. Gécseg, I. Peák. *Algebraic Theory of Automata*. Akadémiai Kiadó, 1972. 847

[131] P. B. Gibbons, Y. Matias, V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation. *Theory of Computing Systems*, 32(3):327–359, 1999. 750

[132] A. Glassner. *An Introduction to Ray Tracing*. Academic Press, 1989. 1092

[133] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989. 1118

[134] N. Goldman, J. Thorne, D. Jones. Using evolutionary trees in protein secondary structure prediction and other comparative sequence analyses. *Journal of Molecular Biology*, 263(2):196–208, 1996. 1010

[135] J. Goldstein, P. A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Optimizing queries using materialized views: a practical, scalable solution*, pages 331–342, 2001. 930

[136] H. G. H. Gonnet, M. A. Cohen, S. A. Benner. Exhaustive matching of the entire protein sequence database. *Science*, 256:1443–1445, 1992. 977

[137] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982. 977

[138] G. Gottlob, C. Koch, R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the 22nd Symposium on Principles of Database Systems*, 2003, pages 179–190. 971

[139] G. Grahne, A. Mendelzon. Tableau techniques for querying information sources through global schemas. In *Proceedings of ICDT'99*, Lecture Notes in Computer Science. Vol. 1540, pages 332-347, Springer-Verlag, 1999. 930

[140] A. Grama, A. Gupta, G. Karypis, V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2003 (2. edition). 749

[141] J. Grant, J. Minker. Inferences for numerical dependencies. *Theoretical Computer Science*, 41:271–287, 1985. 881

[142] J. Grant, J. Minker. Normalization and axiomatization for numerical dependencies. *Information and Control*, 65:1–17, 1985. 881

[143] J. N. Gray. Notes on database operating system. In R. Bayer, R. M. Graham, G. Seegmuller (editors), *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science. Volume 60, pages 393–481. Springer-Verlag, 1978. 641

[144] W. Gropp, M. Snir, B. Nitzberg, E. Lusk. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. The MIT Press, 1998. 750

[145] Q-P. Gu, S. Peng, H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoretical Computer Science*, 210(2):327–339, 1999. 1006

[146] R. Gusella. A measurement study of diskless workstation traffic on an ethernet. *IEEE Transactions on Communications*, 38:1557–1568, 1990. 701

[147] D. M. Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55:141–154, 1993. 982

[148] D. M. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997. 981, 985

[149] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of ACM*, 28(1):532–535, 1988. 750

[150] T. Gyires. Simulation of the harmful consequences of self-similar network traffic. *The Journal of Computer Information Systems*, 42(4):94–111, 2002. 701

[151] T. Gyires. Extension of multiprotocol label switching for long-range dependent traffic: QoS routing and performance in IP networks. *Computer Standards and Interfaces*, 27:117–132, 2005. 701

[152] A. Halevy, A. Rajaraman, J. J. Ordille, D. Srivastava. Querying heterogeneous information sources using source descriptions. In *Proceedings of Very Large Data Bases*, 1996, pages 251–262. 930

[153] A. Halevy. Logic based techniques in data integration. In J. Minker (editor), *Logic-based Artificial Intelligence*, 575–595. pages. Kluwer Academic Publishers, 2000. 930

[154] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10:270–294, 2001. 930

[155] A. Halevy, A. Mendelzon, Y. Sagiv, D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM-Press, 1995, pages 95–104. 930

[156] S. Hannenhalli. Polynomial-time algorithm for computing translocation distance between genomes. *Discrete Applied Mathematics*, 71:137–151, 1996. 1012

[157] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, 2001. 1092

[158] M. Höchsman, T. Töller, R. Giegerich, S. Kurtz. Local similarity in RNA secondary structure. In *Proceedings of IEEE Bioinformatics Conference 2003*, 159–168. pages, 2003. 1010

[159] H. Hefles, D. Lucantoni. A markov modulated characterization of packetized voice and data traffic and related statistical multiplexer performance. *IEEE Journal on Selected Areas in Communication*, 4:856–868, 1986. 701

[160] H. He, J. Yang. Multiresolution indexing of XML for frequent queries. In *Proceedings of the 20th International Conference on Data Engineering*, 2004, pp. 683–694. 971

[161] E. A. Heinz. *Algorithmic Enhancements and Experiments at High Search Depths*. Vieweg Verlag, Series on Computational Intelligence, 2000. 1100, 1118

[162] M. R. Henzinger, T. A. Henzinger, P. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1995, pages 453–462. 971

[163] I. Herman. *The Use of Projective Geometry in Computer Graphics*. Springer-Verlag, 1991. 1092

[164] F. Heylighen. Principia Cybernetica Project. http://pespmc1.vub.ac.be/HEYL.html, 2004. 701

[165] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18:341–343, 1975. 983

[166] 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. 847

[167] E. Horowitz, S. Sahni, S. Rajasekaran. *Computer Algorithms*. Computer Science Press, 1998. 749, 750

[168] T. J. P. Hubbard, A. M. Lesk, A. Tramontano. Gathering them into the fold. *Nature Structural Biology*, 4:313, 1996. 981

[169] R. Hughey, A. Krogh. Hidden markov models for sequence analysis: Extension and analysis of the basic method. *CABIOS*, 12(2):95–107, 1996. 1010

[170] J. Hunt, T. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977. 1009

[171] K. Hwang, Z. Xu. *Scalable Parallel Computing*. McGraw-Hill, 1998. 750

[172] Interoute. http://wwww.interoute.com/glossary.html, 2004. 700

[173] A. Iványi. On dumpling-eating giants. In *Finite and Infinite Sets* (Eger, 1981), Colloquia of Mathematical Society János Bolyai,. Volume 37, pages 279–390. North-Holland, 1984. 847

[174] A. Iványi. Performance bounds for simple bin packing algorithms. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 5:77–82, 1984. 848

[175] A. Iványi. Tight worst-case bounds for bin packing algorithms. In *Theory of Algorithms* (Pécs, 1984), Colloquia of Mathematical Society János Bolyai. Volume 44, pages 233–240. North-Holland, 1985. 848

[176] A. Iványi. *Párhuzamos algoritmusok* (*Parallel Algorithms*). ELTE Eötvös Kiadó, 2003. 749

[177] A. Iványi, R. Szmeljánszkij. *Elements of Theoretical Programming* (in Russian). Moscow State University, 1985. 847

[178] R. Jain, S. Routhier. Packet trains: Measurements and a new model for computer network traffic. *IEEE Journal on Selected Areas in Communication*, 4:986–995, 1986. 701

[179] M. R. Jerrum. The complexity of finding minimum-length generator sequences. *Theoretical Computer Science*, 36:265–289, 1986. 1012

[180] T. Jiang, M. Li. Towards a DNA sequencing theory (learning strings). Technical Report91-08, Computer Science Department, McMaster University, 1991. 1008

[181] P. Jiménez, F. Thomas, C. Torras. 3D collision detection: A survey. *Computers and Graphics*, 25(2):269–285, 2001. 1092

[182] D. S. Johnson. *Near-Optimal Bin Packing Algorithms*. PhD thesis, MIT Department of Mathematics, 1973. 848

[183] 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. 848

[184] G. A. Jones, J. Jones. *Information and Coding Theory*. Springer-Verlag, 2000. 700

[185] K. Jones. *Consultant's Guide to COMNET III*. CACI Product Company, 1997. 700

[186] M. Kandemir, J. Ramanujam, A. Choudhary. Compiler algorithms for optimizing locality and parallelism on shared and distributed-memory machines. *Journal of Parallel and Distributed Computing*, 60:924–965, 2000. 750

[187] S. Karlin, M. T. Taylor. *A First Course in Stochastic Processes*. Academic Press, 1975. 1092

[188] R. M. Karp, R. E. Miller, S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967. 794

[189] R. Kaushik, R. Krishnamurthy, J. F. Naughton, R. Ramakrishnan. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 129–140, 2002. 971

[190] R. Kaushik, R. Shenoy, P. F. Bohannon, E. Gudes. On the integration of structure indexes and inverted lists. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pages 779–790. 971

[191] R. R. Kaushik, P. Bohannon, J. F. Naughton, H. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002, pages 133–144. 971

[192] R. R. Kaushik, P. Bohannon, J. F. Naughton, H. Korth, P. Shenoy. Updates for structure indexes. In *Proceedings of Very Large Data Bases*, 2002, pages 239–250. 971

[193] D. Kececioglu, E. Myers. Combinatorial algorithms for DNA sequence assembly. Technical report, Department of Computer Sciences, University of Arizona, Department of Computer Sciences, University of Arizona. 1008

[194] J. Kececioglu, H. Lenhof, K. Mehlhorn, P. Mutzel, K. Reinert, M. Vingron. A polyhedral approach to sequence alignment problems. *Discrete Applied Mathematics*, 104((1-3)):143–186, 2000. 1009

[195] K. Kennedy, R. Allen. *Optimizing Compilers for Modern Architectures*. Morgan Kaufman Publishers, 2001. 750

[196] M. Khosrow-Pour (editor). *Encyclopedia of Information Science and Technology, Vol. 1, Vol. 2, Vol. 3, Vol. 4, Vol. 5*. Idea Group Inc., 2005. 971

[197] S. Kleiman, D. Shah, B. Smaalders. *Programming with Threads*. Prentice Hall, 1996. 750

[198] L. Kleinrock. *Queueing Systems*. John Wiley & Sons, 1975. 700

[199] B. Knudsen, J. Hein. RNA secondary structure prediction using stochastic context free grammars and evolutionary history. *Bioinformatics*, 15(6):446–454, 1999. 993

[200] B. Knudsen, J. Hein. Pfold: RNA secondary structure prediction using stochastic context-free grammars. *Nucleic Acids Researchs*, 31(13):3423–3428, 2003. 993, 1010

[201] D. E. Knuth, J. Morris, V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. 1009

[202] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. Steele Jr., M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994. 750

[203] G. Krammer. Notes on the mathematics of the PHIGS output pipeline. *Computer Graphics Forum*, 8(8):219–226, 1989. 1092

[204] H. T. Kung, C. E. Leiserson. Systolic arrays (for VLSI). In I. S. Duff, G. W. Stewart (editors), *Sparse Matrix Proceedings*. pages 256–282. SIAM, 1978. 794

[205] C. T. Kwok, D. Weld. Planning to gather information. In *Proceedings of AAAI 13th National Conference on Artificial Intelligence*, 1996, pages 32–39. 930

[206] T. Lai, S. Sahni. Anomalies in parallel branch and bound algorithms. *Communications of ACM*, 27(6):594–602, 1984. 847

[207] E. T. Lambrecht, S. Kambhampati, S. Gnanaprakasam. Optimizing recursive information gathering plans. In *Proceedings of 16th International Joint Conference on Artificial Intelligence*, pages 1204–1211. 1999. 930

[208] J. Lamperti. *Stochastic Processes*. Springer-Verlag, 1972. 1092

[209] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 18(8):453–455, 1974. 642

[210] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computers*, 5(1):1–11, 1987. 642

[211] L. Lamport, R. Shostak M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. 642

[212] G. Lancia. Integer programming models for computational biology problems. *Journal of Computer Science and Technology*, 19(1):60–77, 2004. 1009

[213] G. Landau, U. Vishkin. Eficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986. 1009

[214] A. Law, W. Kelton. *Simulation Modeling and Analysis*. 3. edition. McGraw-Hill Higher Education, 1999. 700

[215] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes*. Morgan Kaufman Publishers, 1992. 750

[216] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Algorithms and VSLI*. Morgan Kaufman Publishers, 2001. 750

[217] W. Leland, M. Taqqu, W. Willinger, D. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, 1994. 701

[218] W. Leland, M. Taqqu, W. Willinger, D. Wilson. Statistical analysis and stochastic modeling of self-similar data-traffic. In J. Labetoulle, J. W. Roberts (editors), *The Fundamental Role of Teletraffic in the Evolution of Telecommunications Networks*. Elsevier Science B. V., 1994. 701

[219] W. Leland, M. Taqqu, D. Wilson. On the self-similar nature of ethernet traffic. *Computer Communication Reviews*, 23:183–193, 1993. 700

[220] C. Leopold. *Parallel and Distributed Computing*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, Copyrights 2001. 642, 749, 750

[221] B. Lewis, D. J. Berg. *Multithreaded Programming with Phtreads*. Prentice Hall, 1998. 750

[222] D. Lipman, S. J. Altshuland, J. Kecioglu. A tool for multiple sequence alignment. *Proc. Natl. Academy Science*, 86:4412–4415, 1989. 985

[223] M. Listanti, V. Eramo, R. Sabella. Architectural and technological issues for future optical internet networks. *IEEE Communications Magazine*, 8(9):82–92, 2000. 701

[224] C. Lucchesi. Candidate keys for relations. *Journal of of Computer and System Sciences*, 17(2):270–279, 1978. 881

[225] G. Lunter, I. Miklós, A. Drummond, J. L. Jensen, J. Hein. Bayesian phylogenetic inference under a statistical indel model. *Lecture Notes in Bioinformatics*, 2812:228–244, 2003. 1010

[226] G. Lunter, I. Miklós, Y., J. Hein. An efficient algorithm for statistical multiple alignment on arbitrary phylogenetic trees. *Journal of Computational Biology*, 10(6):869–889, 2003. 1010

[227] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufman Publisher, 2001 (5. edition). 642, 795

[228] N. A. Lynch, M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13(1):17–43, 1981. 641

[229] R. Lyngso, C. N. S. Pedersen. RNA pseudoknot prediction in energy based models. *Journal of Computational Biology*, 7(3/4):409–428, 2000. 1010

[230] R. Lyngso, M. Zuker, C. Pedersen. Fast evaluation of internal loops in RNA secondary structure prediction. *Bioinformatics*, 15(6):440–445, 1999. 1010

[231] D. Maier. Minimum covers in the relational database model. *Journal of the ACM*, 27(4):664–674, 1980. 881

[232] D. Maier, A. O. Mendelzon, Y. Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems*, 4(4):455–469, 1979. 881

[233] B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, 1982. 701

[234] B. Mandelbrot, J. W. Van Ness. Fractional brownian motions, fractional noises and applications. *SIAM Review*, 10:422–437, 1968. 701

[235] R. L. Mattson, J. Gecsei, D. R. Slutz, I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970. 847

[236] E. A. Maxwell. *Methods of Plane Projective Geometry Based on the Use of General Homogenous Coordinates*. Cambridge University Press, 1946. 1091

[237] E. A. Maxwell. *General Homogenous Coordinates in Space of Three Dimensions*. Cambridge University Press, 1951. 1091

[238] McAfee. Sniffer technologies. http://www.nai.com/us/index.asp, 2004. 701

[239] J. S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29:1105–1119, 1990. 1010

[240] B. Melamed. An overview of tes processes and modeling methodology. In L. Donatiello, A. R. Nelson (editors), *Models and Techniques for Performance Evaluation of Computer and Communications Systems*, pages 359–393. Lecture Notes in Computer Science. Springer-Verlag, 1993. 701

[241] I. Meyer, R. Durbin. Comparative ab initio prediction of gene structures using pair HMMs. *Bioinformatics*, 18(10):1309–1318, 2002. 1010

[242] I. M. Meyer, R. Durbin. Gene structure conservation aids similarity based gene prediction. *Nucleic Acids Research*, 32(2):776–783, 2004. 1010

[243] Sz. Mihnovskiy, N. Shor. Estimation of the page fault number in paged memory (in Russian). *Kibernetika (Kiev)*, 1(5):18–20, 1965. 847

[244] W. Miller, E. Myers. A file comparison program. *Software – Practice and Experience*, 15(11):1025–1040, 1985. 1009

[245] W. Miller, E. W. Myers. Sequence comparison with concave weighting functions. *Bulletin of Mathematical Biology*, 50:97–120, 1988. 977

[246] T. Milo, D. Suciu. Index structures for path expressions. In *7th International Conference on Data Bases*, Lecture Notes in Computer Science, Vol. 1540. Springer-Verlag, 1999, pages 277–295. 971

[247] S. Molnár, A. Vidács, A. Nilsson. Bottlenecks on the way towards characterization of network traffic: Estimation and interpretation of the hurst parameter. http://hsnlab.ttt.bme.hu/ molnar (Conference papers), 1997. 701

[248] B. Morgenstern. DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, 15:211–218, 1999. 982

[249] B. Morgenstern, A. Dress, T. Werner. Multiple DNA and protein sequence alignment based on segment-to-segment comparison. *Proc. Natl. Academy Science*, 93:12098–12103, 1996. 982

[250] B. Morgenstern, K. Frech, A. Dress, T. Werner. DIALIGN: Finding local similarities by multiple sequence alignment. *Bioinformatics*, 14:290–294, 1998. 982

[251] S. N. Needleman, C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970. 1009

[252] M. F. Neuts. A versatile markovian point process. *Journal of Applied Probability*, 18:764–779, 1979. 701

[253] M. F. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications*. Marcel Dekker, 1989. 701

[254] M. E. Newell, R. G. Newell, T. L. Sancha. A new approach to the shaded picture problem. In *Proceedings of the ACM National Conference*, 1972, pages 443–450. 1092

[255] R. Nussinov, G. Pieczenk, J. Griggs, D. Kleitman. Algorithms for loop matching. *SIAM Journal of Applied Mathematics*, 35:68–82, 1978. 993

[256] S. Oaks, H. Wong. *Java Threads*. O'Reilly, 1999. 750

[257] OpenMP Website. http://www.openmp.org, 2007. 750

[258] OPNET. Opnet Modeler Documentation. www.opnet.com, 2007. 701

[259] J. O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987. 1092

[260] S. Owicki, D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976. 641

[261] S. Owicki, L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982. 641

[262] L. Pachter, B. Sturmfels (editors). *Algebraic Statistics for Computational Biology*. Cambridge University Press, 2005. 1009

[263] R. Page, E. Holmes. *Molecular Evolution: a Phylogenetic Approach*. Blackwell, 1998. 981

[264] R. Paige, R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987. 971

[265] C. Partridge. The end of simple traffic models. *IEEE Network*, 7(9), 1993. Editor's Note. 701

[266] V. Paxson, S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3:226–244, 1995. 701

[267] M. Pease, R. Shostak L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980. 642

[268] J. S. Pedersen J. Hein. Gene finding with a hidden Markov model of genome structure and evolution. *Bioinformatics*, 19(2):219–227, 2003. 1010

[269] G. Peterson, J. Fischer. Economical solutions for the critical section problem in distributed systems. In *Proceedings of the 9th ACM Symposium on Theory of Computing*. pages 91–97. IEEE Computer Society Press, 1977. 641, 642

[270] S. Petrov. Finite axiomatization of languages for representation of system properties. *Information Sciences*, 47:339–372, 1989. 881

[271] P. A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000. 1007

[272] G. F. Pfister. *In Search of Clusters*. Prentice Hall, 1998 (2. edition). 750

[273] N. Pisanti, M. Sagot. Further thoughts on the syntenic distance between genomes. *Algorithmica*, 34(2):157–180, 2002. 1012

[274] N. Polyzotis, M. N. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proceedings of the 2002 ACM SIGMOD international Conference on Management of Data*, 2002, pages 358–369. 971

[275] R. Pottinger. MinCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2):182–198, 2001. 930

[276] R. Pottinger, A. Halevy. A scalable algorithm for answering queries using views. In *Proceedings of Very Large Data Bases'00*, 2000, pages 484–495. 930

[277] F. P. Preparata, M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985. 1092

[278] T. Pupko, I. Peer, R. Shamir, D. Graur. A fast algorithm for joint reconstruction of ancestral amino acid sequences. *Molecular Biology and Evolution*, 17:890–896, 2000. 1010

[279] X. Qian. Query folding. In *Proceedings of International Conference on Data Engineering*, pages 48–55. 1996. 930

[280] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 208–214. 1984. 794

[281] K. Räihä, E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16:187–198, 1981. 1008

[282] S. K. Rao. Regular iterative algorithms and their implementations on processor arrays. Doktori értekezés, Stanford University, 1985. 794

[283] R. Ravi, J. D. Kececioglu. Approximation algorithms for multiple sequence alignment under a fixed evolutionary tree. *Discrete Applied Mathematics*, 88(1–3):355–366. 982

[284] P. Resnick, H. R. Varian. Recommender Systems. *Communications of the ACM*, 40(3):56–58, 1997. 1118

[285] E. Rivas, S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285(5):2053–2068, 1999. 1010

[286] A. Rényi. *Probability Theory*. Akadémiai Kiadó/North Holland Publ. House, 1970. 700

[287] S. Roch. A short proof that phylogenetic tree reconstruction by maximum likelihood is hard. *EEE Transactions on Computational Biology and Bioinformatics*, 3(1):92–94, 2006. 1010

[288] D. F. Rogers, J. Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill Book Co., 1989. 1092

[289] S. H. Roosta. *Parallel Processing and Parallel Algorithms*. Springer-Verlag, 1999. 847

[290] S. N. Ross. *Simulation*. Academic Press, 2006. 700

[291] A. Sali, Sr., A. Sali. Generalized dependencies in relational databases. *Acta Cybernetica*, 13:431–438, 1998. 881

[292] J. Sameith. On the generation of alternative solutions for discrete optimization problems with uncertain data – an experimental analysis of the penalty method. http://www.minet.uni-jena.de/Math-Net/reports/shadows/04-01report.html, 2004. 1118

[293] D. Sankoff. Minimal mutation trees of sequences. *SIAM Journal of Applied Mathematics*, 28:35–42, 1975. 981, 987

[294] L. A. Santaló. *Integral Geometry and Geometric Probability*. Addison-Wesley, 1976. 1092

[295] N. Santoro. *Design and Analysis of Distributed Algorithms*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons, 2006. 642

[296] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, 1983. 641

[297] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1(1):51–64, 1991. 1092

[298] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal of Applied Mathematics*, 26:787–793, 1974. 974

[299] B. Sharp. Implementing subdivision theory. *Game Developer*, 7(2):40–45, 2000. 1092

[300] B. Sharp. Subdivision Surface theory. *Game Developer*, 7(1):34–42, 2000. 1092

[301] I. Shindyalov, P. Bourne. Protein structure alignment by incremental combinatorial extension (CE) of the optimal path. *Protein Engineering*, 11(9):739–747, 1998. 1010

[302] A. Silberschatz, P. Galvin, G. Gagne. *Applied Operating System Concepts*. John Wiley & Sons, 2000. 750, 847

[303] D. Sima, T. Fountain, P. Kacsuk. *Advanced Computer Architectures: a Design Space Approach*. Addison-Wesley Publishing Company, 1998 (2. edition). 700, 749, 795

[304] T. F. Smith, M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981. 980

[305] J. L. Spouge. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM Journal of Applied Mathematics*, 49:1552–1566, 1989. 984

[306] J. L. Spouge. Fast optimal alignment. *CABIOS*, 7:1–7, 1991. 984, 985

[307] L. Stockmeyer, A. R. Meyer. Word problems requiring exponential time. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*. ACM Press, 1973, pp. 1–9. 971

[308] I. Sutherland, G. Hodgeman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974. 1092

[309] I. E. Sutherland, R. Sproull, R. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974. 1092

[310] L. Szécsi. An effective kd-tree implementation. In J. Lauder (editor), *Graphics Programming Methods*. Charles River Media, 2003, pages 315–326. 1092

[311] L. Szirmay-Kalos, G. Márton. Worst-case versus average-case complexity of ray-shooting. *Computing*, 61(2):103–131, 1998. 1092

[312] L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics*. Akadémiai Kiadó, 1995. 1092

[313] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001. 750

[314] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 2004. 700

[315] A. S. Tanenbaum, M. van Steen. *Distributed Systems. Principles and Paradigms*. Prentice Hall, 2002. 750

[316] A. S. Tanenbaum, A. Woodhull. *Operating Systems. Design and Implementation*. Prentice Hall, 1997. 847

[317] M. Taqqu, W. Teverovsky, W. Willinger. Estimators for long-range dependence: an empirical study. *Fractals*, 3(4):785–788, 1995. 700

[318] J. Tarhio, J. E. Ukkonen A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57:131–145, 1988. 1008

[319] J. Teich, L. Thiele. Control generation in the design of processor arrays. *International Journal of VLSI and Signal Processing*, 3(2):77–92, 1991. 794

[320] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000 (2. edition). 642

[321] B. Thalheim. *Dependencies in Relational Databases*. B. G. Teubner, 1991. 881

[322] J. D. Thompson, D. G. Higgins, T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994. 977, 981, 1009

[323] I. Tinoco, O., Uhlenbeck M. Levine. Estimation of secondary structure in ribonucleic acids. *Nature*, 230:362–367, 1971. 1010

[324] Top 500 Supercomputer Sites. http://www.top500.org, 2007. 749

[325] Trusoft Intl Inc.. *Benoit 1.1*. Trusoft Intl Inc., 2007. 701

[326] O. G. Tsatalos, M. C. Solomon, Y. Ioannidis. The GMAP: a versatile tool for physical data independence. *The VLDB Journal*, 5(2):101–118, 1996. 930

[327] D. M. Tsou, P. C. Fischer. Decomposition of a relation scheme into Boyce–Codd normal form. *SIGACT News*, 14(3):23–29, 1982. 881

[328] A. Tucker. *Handbook of Computer Science*. Chapman & Hall/CRC, 2004. 971

[329] Y. Uemura, A. Hasegawa, Y. Kobayashi, T. Yokomori. Tree adjoining grammars for RNA structure prediction. *Theoretical Computer Science*, 210:277–303, 1999. 1010

[330] E. Ukkonen. On approximate string matching. *Lecture Notes in Computer Science*, 158:487–495, 1984. 984

[331] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985. 984

[332] J. D. Ullman. *Principles of Database and Knowledge Base Systems. Vol. 1*. Computer Science Press, 1989 (2. edition). 881

[333] J. D. Ullman. Information integration using logical views. In *Proceedings of ICDT'97*, Lecture Notes in Computer Science. Volume 1186, pages 19–40. Springer-Verlag, 1997. 930

[334] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. 750

[335] V. Vianu. A Web Odyssey: from Codd to XML. In *Proceedings of the 20th Symposium on Principles of Database Systems*, 2001, pages 1–5. 970

[336] T. Várady, R. R. Martin, J. Cox. Reverse engineering of geometric models - an introduction. *Computer-Aided Design*, 29(4):255–269, 1997. 1092

[337] L. Wang, T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1:337–348, 1994. 981

[338] J. Warren, H. Weimer. *Subdivision Methods for Geometric Design: A Constructive Approach*. Morgan Kaufmann Publishers, 2001. 1092

[339] M. S. Waterman, T. F. Smithand, W. A. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20:367–387, 1976. 976

[340] Web Dictionary of Cybernetics, Systems. http://pcp.wub.ac.be/ASC.html, 2007. 700

[341] J. Weber, E. Myers. Human whole genome shotgun sequencing. *Genome Research*, 7:401–409, 1997. 1008

[342] W. Willinger, V. Paxson. Discussion of "heavy tail modeling and teletraffic data" by S. R. Resnick. *The Annals of Statistics*, 25(5):1805–1869, 1997. 701

[343] W. Willinger, M. Taqqu, R. Sherman, D. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5:71–86, 1997. 701

[344] W. Willinger, D. Wilson, W. Leland, M. Taqqu. On traffic measurements that defy traffic models (and vice versa): self-similar traffic modeling for high-speed networks. *Connections*, 8(11):14–24, 1994. 701

[345] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley Longman, 1996. 750

[346] J. Wu, S.. On cost-optimal merge of two intransitive sorted sequences. *International Journal of Foundations of Computer Science*, 14(1):99–106, 2003. 750

[347] S. Wu, E.. W. Myers, U. Manber, W. Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, 1990. 1009

[348] G. Wyvill, C. McPheeters, B. Wyvill. Data structure for soft objects. *The Visual Computer*, 4(2):227–234, 1986. 1092

[349] H. Z. Yang, P. A. Larson. Query transformation for PSJ-queries. In *Proceedings of Very Large Data Bases'87*, pages 245–254, 1987. 930

[350] J. Yang, K., Karlapalem, Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of Very Large Data Bases'97*, 1997, pages 136–145. 930

[351] K. Yi, H. He, I. Stanoi, J. Yang. Incremental maintenance of XML structural indexes. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pages 491–502. 971

[352] C. Yu, D. Johnson. On the complexity of finding the set of candidate keys for a given set of functional dependencies. In *Information Processing 74*. North-Holland, 1974, pages 580–583. 881

[353] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata. Answering complex SQL queries using automatic summary tables. In *Proceedings of SIGMOD'00*, pages 105–116. 2000. 930

[354] B. Zalik, G. Clapworthy. A universal trapezoidation algorithms for planar polygons. *Computers and Graphics*, 23(3):353–363, 1999. 1092

[355] C. Zaniolo. Analysis and design of relational schemata for database systems. Technical Report UCLA–Eng–7669, Department of Computer Science, University of California at Los Angeles, 1976. 881

[356] C. Zaniolo. A new normal form for the design of relational database schemata. *ACM Transactions on Database Systems*, 7:489–499, 1982. 881

[357] E. Zehendner. *Entwurf systolischer Systeme: Abbildung regulärer Algorithmen auf synchrone Prozessorarrays*. B. G. Teubner Verlagsgesellschaft, 1996. 795

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

A(k)-index, *951*, 964, 968
AABB, *1043*
absolute optimal algorithm, *726*
abstract computer, *see* model of computation
accurate complexity, *726*
action, *see* event
active edge table, *1083*
active mode, 640*pr*
additive metric, *1000*
adjacent processors, *598*
admissible execution, *593*
AET, *1083*
affine function, *977*
affine point, *1046*
affine transformation, *1051*
A(k)-INDEX-EVALUATION, *953*
algorithm
  absolute optimal, 726
  asymptotically optimal, 726
  recursive, 729
  work-efficient, 726
alternatives, true, *1095*
Analysis Tool, 660
analytic geometry, *1013*
ancestor-stable, 962
anomaly, *824*, 857, 865
  deletion, 858
  insertion, 858
  redundancy, 858
  update, 858
anytime algorithm, *1100*, *1115*
arbitrary PRAM, 720
Armstrong-axioms, *850*, 857*ex*, 865
Armstrong-relation, *878*
assignment-free notation, *756*, 767
assignment problem, 1103
asymptotically identical order, 727
asymptotically optimal algorithm, *726*
asymptotically self-similar process, *675*
asynchronous system, *592*, 641
atom
  relational, *884*
attribute, *849*
  external, 880*pr*
  prime, *867*, 880*pr*
average running time, *724*
axiomatisation, *877*

B

backing memory, *813*
backward label sequence, 961
band matrix, *794*
band width, *794*
bandwidth, *648*
bandwith, *710*
bank transactions, 641*pr*
base addressing, *798*
base set, *1103*
basic partition, *940*
basis function, *1019*
basis vector, *1014*
benoit, 701
Bernstein-polinom, *1019*
best case, 724
Best Fit, 838
BEST-FIT, *809*, 812*ex*
Best Fit Decreasing, 839
Bézier curve, *1019*, 1026*ex*
BF, *838*, *see* Best Fit
BFD, *see* Best Fit Decreasing
binary space partitioning tree, *1066*
bioinformatics, 973–1012
bisimilar, *942*, 965
bisimulation, *942*, 965
blob method, *1025*
block, *1015*
Bluetooth, 647
boundary surface, *1015*
bounding volume, *1057*

AABB, *1057*
hierarchikus, *1057*
sphere, *1057*
branch and bound, 1098
branching query, 960, *961*
Bresenham algorithm, *1080*
BRESENHAM-LINE-DRAWING, *1082*
Bresenham line drawing algorithm, *1081*
broadcast network, 646
BSP, *720*
B-spline, *1020*, *1022*
order, *1020*
BSP-tree, *1066*, *1089*
BSP-TREE-CONSTRUCTION, *1090*
bubble sort, 791*fig*, 792
bucket, *917*
Bucket Algorithm, *917*, 924
BULLY, 600, *602*, 641
burstiness, *650*
Burst Interarrival Time, *679*
busy page frame, 814
butterfly subdivision, *1033*
Byzantine failure, *606*, 608

C

CACI, *see* Consolidated Analysis center, Inc.
CALCULATE-RANK, *804*
calculation mode, *781*
camera transformation, *1074*
candidate solutions, 1094–1096, 1098, 1114
Cartesian coordinate system, *1014*
Catmull-Clark subdivision, 1033*fig*
Catmull-Clark subdivision algorithm, *1033*
CCW, *see* counter clock-wise
cell, *753*, 758, 769
boundary, *753*, 770, 774, 775, 783, 791
program, *787*
structure of, 752*fig*, *753*, 762*fig*, 764, 768–770, 785*fig*, 791*fig*
with distributed control, *783*, 785*fig*
with global control, *779*
with local control, 780–783
without control, *779*
CELL-PROGRAM, *790*
cellular automaton, 795
certain answer, 928*pr*
chain, *721*
CHAIN-PREFIX, *747*
cherry motif, *997*
chess, *1097*, 1115
children*children*, 594
Chomsky normal form, 991
clear, 755, *759*, 780
clipping, *1036*, 1041, 1071, *1077*
line segments, *1041*
CLOCK, 821, 823*ex*
clock signal, 752, *757*, 758, 778
clock-wise, *600*
CLOSURE, *852*, 870, 879*ex*
of a set of attributes, 857*ex*
of a set of functional dependencies, *850*, 851, 857*ex*
of set of attributes, *851*, 852
clustering algorithms, 1098
COHEN-SUTHERLAND-LINE-CLIPPING, *1045*
Cohen-Sutherland line clipping algorithm, *1044*
collecting rumors, *624*
collector, 627
collector processor, *627*
collision detection, *1036*, *1053*
columnindex, *722*
column of processors, 722
combined PRAM, 720
COMBINE-MCDS, *927*
common PRAM, 720
communication
external, 758
internal, 758
communication topology
hexagonal, *768*
of systolic array, *767*, 786
orthogonal, *768*
COMNET, 652
complexity
message, 593
time, 593
compound operation, *757*, 778
computational biology, 1098
computer graphics, 1013–1093
condition
liveness, 592
safety , 592
cone, *1016*
configuration, *592*
connection pattern, *753*, 764
consensus problem, *606*, 642
CONSENSUS-WITH-BYZANTINE-FAILURES, *609*
CONSENSUS-WITH-CRASH-FAILURES, *607*, 642
consistent cut, *616*
constructive solid geometry, *1025*
control
local/global, 780, 781*fig*
control signal
propagated, 783
convex combination, *1017*
convex combinations, *1016*
convex hull, *1019*
convex vertex, *1029*
coordinate, *1014*
copy operation, *768*
cost driven method, *1067*
counter clock-wise, *600*
covering argument, *638*
covering of a shared variable, *638*
Cox–deBoor algorithm, *1022*
crash failure, *606*
CRCW, 720, 742
CRCW PRAM, 746*ex*
CREATE-BUCKET, *917*
CREW, 720
CREW PRAM, 728, 732, 733
CREW-Prefix, *729*
crossover point, *727*
cross product, *1014*
cube, *722*
curve, *1016*
cut, *615*
CW, *see* clock-wise
CYK algorithm, 991
cylinder, *1016*

CS

CSG, *see* constructive solid geometry

CSG tree, 1026

D
D(k)-index, *954*, 955
D(k)-Index-Creator, 956
data
  input, *756*, 758
  output, *756*
database architecture
  layer
    logical, *901*
    outer, *902*
    physical, *901*
data dependence, *767*
data dependencies, 711
Data Exchange Chart, *664*
data flow, *767*
  regular, 753
data independence
  logical, *903*
data integration system, *907*
datalog
  non-recursive, *890*
    with negation, *891*
  program, *894*, 920
    precedence graph, *896*
    recursive, *896*
  rule, *893*, 920
data parallelism, *708*
data rate, 770, 774, *792*
data storage, 758
data stream, *752*
  input, 774, 791
  input/output, 774
  length of, 760
data structure index, *771*, 772
data structure vector, *771*
DDA, *see* digital differential analyzator algorithm
DDA-Line-Drawing, *1080*
d-dimensional hypercube, *722*
decision maker, 1095
decision of a processor, *606*
decision variable, *1081*
decomposition
  dependency preserving, *863*
  dependency preserving into 3NF, 870
  lossless join, *859*
    into BCNF, *867*
delay, *758*, 759, 781
dendrograms, *998*
dependence vector, *767*, 788
dependency
  branching, *878*
  equality generating, *871*
  functional, *849*
    equivalent families, *854*
    minimal cover of a family of, *855*
  join, *877*
  multivalued, *871*
  numerical, *879*
  tuple generating, *871*
dependency basis, *873*
Dependency-basis, *874*, 880*pr*
depth-buffer, *1084*
depth first search, 897
depth of the tree, 962
Depth Search First, 595
descendant-stable, 962
Det-Ranking, *734*
DFS, *see* Depth Search First
diagonal, *1028*
diameter, 722
digital differential analyzator algorithm, *1079*
disseminator processor, *627*
distinct shared variable, *638*
distributed algorithms, 591–642
Distributed-Snapshot, 617
divide-and-conquer, 736, 741, 745, 746
divide-and-conquer algorithm, 1104
Divide-and-Cover, *1105*
DNA, 973
domain, *756*, *849*
  dense convex, *764*, 765
  of input/output operations, 775
  parametric, *764*
domain calculus, 861
domain restricted, *889*
dot product, *755*, *1013*
dropped packet rate, 650
DTD, *938*
dual tree, *1030*
dynamic page replacement algorithms, *814*
dynamic partitions, *805*
dynamic programming, 1098
dynamic programming table, *975*
D(k)-Index-Creator, *955*

E, É
ear, *1029*
ear cutting, *1030*
e-commerce, 1097
Economic Power, 841
Edgeaddition-1-Index, *967*
Edgeaddition-FB-Index, *968*
Edge-Plane-Intersection, 1042
edge point, *1033*
efficiency, *726*, 747
Efficient-Maximal-Simulation, *937*
Efficient-PT, *949*
efficient solutions, *1116*
ellipse, *1017*
emulation, 645
end cell, *791*, 792
ending phase, *628*
Ending-Phase, *630*
EP, *see* Economic Power
ε-alternatives, 1103, 1106
  monotonicity properties, *1107*
  unimodality property, *1106*
Equate, *860*
equational calculus, 756
equation of the line, *1048*
  in homogeneous coordinates, *1049*
equation of the tangent plane, *1018*
ERCW, 720
EREW, 719
EREW PRAM, 746*ex*
EREW-Prefix, 728, *730*, 732
event, *592*
exact, *941*
Exact-cover, *899*
execution, *592*

execution sequences, 641
execution time, *825*
external point, *1015*
eye position, *1073*

F
F+B+F+B-index, 963
F+B-index, 963
face point, *1033*
FACT, *894*, *see* Factorial
Factorial, 840
failure, 795
  Byzantine, 606
  crash, 606
fairness, *592*
fast exclusion algorithm, *638*
Fast-Lock, 639
FAST-MUTUAL-EXCLUSION, *639*, 640
FB(f, b, d)-index, 961, 963
FB(f, b, d)-INDEX-CREATOR, *964*
FB-index, 961, 962, 967
FB-INDEX-CREATOR, *963*
FDDI, *see* Fiber Distributed Data Interface
feasible solution, *1103*
feasible subset, *1103*
Felsenstein algorithm, *987*
FF, *837*, *see* First Fit
FFD, 839, *see* First Fit decreasing
Fiber Distributed Data Interface, 647
FIFO, *815*, *see* First In First Out
FIFO-EXECUTES, *816*
final choice, 1094, 1097
First Fit, 837
FIRST-FIT, *807*, 812*ex*
First In First Out, 815
fixed partitions, *798*
fixed point number representation, *1080*
fixpoint, *894*
FLOOD, *596*, 640*pr*
flow direction, *768*, 769, 770
FORM-MCDS, *926*
forward label sequence, 961
FORWARD-LOOKING, *978*
forward looking algorithm, *978*
FORWARD-LOOKING-BINARY-SEARCHING, *979*
Fourier-Motzkin elimination, 789
four-point metric, *1000*
fragmentation the memory, 808
frame, *813*
  busy, 814
frame size, 650
free tuple, *884*, 891
frequent regular queries, 956
FULL-TUPLEWISE-JOIN, *897*
functional representation, 1092

G
gap, *976*
gap penalty, *976*
general, 641*pr*
Generalised Multi-level Access Path, *906*
GENERAL-SELECTION, *743*
generative design, *1116*
generic operator, *755*, 775
  distributed, *769*
genetic algorithm, *1099*, *1116*
Global Positioning System, *647*
global view, *787*
GMAP, *906*
gossip, *624*
GPS, *see* Global Positioning System
grammar, 961
GRAPHADDITION-1-INDEX, *966*
GRAPHADDITION-A(k)-INDEX, *969*
GRAPHADDITION-FB-INDEX, *968*
graphical visualisation, *1117*
graph message, *628*
gravity assist, 1097
guide-tree, *981*

H
Hamming-distance, 722
happens before, *613*, 618
hardware algorithm, *751*
head homomorphism, *925*
head of a list, *733*
helix, *1017*
heuristic, *1098*, 1099, 1113, 1115, 1116
  exchange, *1099*, 1100
  insertion, *1099*
Hidden Markov Model, *989*
High Performance Fortran, 718, 750
Hirschberg algorithm, *983*
HMM, *see* Hidden Markov Model
homogeneous coordinate, *1047*, *1049*
homogeneous linear transformation, *1046*, *1050*
homomorphism theorem, 897
Horn rule, *920*
host computer, 752, *758*, 780, 783, 786, 791
HPF, *see* High Performance Fortran
human-computer interaction, *1094*
Hurst parameter, 679
Hurst-parameter, *675*

I, Í
ideal line, *1046*
ideal plane, *1046*
ideal point, *1046*
identifier, *600*
image, *1013*
image under a query, *885*
immediate consequence, *894*
  operator, *894*
implicit equation, *1015*
IMPROVED-MAXIMAL-SIMULATION, *936*
IMPROVED-SEMI-NAIV-DATALOG, *897*, 900*ex*
incremental principle, *1071*, *1079*, *1083*
index, *940*
INDEX-EVALUATION, *942*
index expression, *787*
index function, *788*
indexing, *938*
indexing of processors, *748*
index of an index, 964
index of the virtual page frame, *814*
index refresh, 964
inference rules, *850*
  complete, *850*
  sound, *850*
infinite domain, *885*

inhomogeneity, *793*
input/output expansion, *775*
input/output scheme, 752*fig*, 753, 759, *770*, 771*fig*, 772, 780*fig*, 791*fig*
  extended, 774, 775, 775, 776*fig*
  superposition of, 773, 774
input of a processor, *606*
input stream, *760*, 779
  length of, 760
inquiring message, *628*
INSERTION-HEURISTIC-FOR-TSP, *1099*
insertion sort, 792
INSIDE, 991
instance, *756*, 759, *849*, 882
instruction affects causally, *613*
integer linear programming, *1113*
INTEGER-SELECTION, *742*
integral geometry, *1067*, 1092
integrity constraint, *849*, 863, 901
interactive evolution, *1116*
interleaving, *776*, 777*fig*
internal point, *1015*
Internet, 646
internetwork, *646*
intersection calculation
  plane, *1055*
  triangle, *1055*
Inverse-rules Algorithm, 919, 924
iso-parametric curve, *1017*
iteration
  variable, *756*, 787
  vector, *756*, 757, 763, 771, 772, 778
iterative deepening, *1100*, *1115*
ITERATIVE-DEEPENING-TREE-SEARCH, *1115*
ITERATIVE-PENALTY-METHOD, *1108*
iterative sequence alignment, *981*
beg-ind, 839
counter, 823
end-ind, 839
free, 838
guarded, 818
ind, 838

J
join
  natural, *886*
JOIN-TEST, *860*, 861*fig*, 870

K
k-best algorithm, *1095*
k-bisimilar, *951*
k-bisimulation, *951*
kd-tree, *1066*
KD-TREE-CONSTRUCTION, *1067*
key, 850, *856*, 867
  primary, *877*
knapsack problem, 1103
knot vector, *1020*

L
label sequence, *938*
LAN, *see* local area network
LARGEST-FIT, *799*, 812*ex*
LARGEST-OR-LONG-WAITING-FIT, *802*, 812*ex*
latency, *649*, *710*
lazy method, 966
leader election, 641
left handed, *1078*
level, 723
LF, *837*, *see* Linear Fit
LFU-EXECUTES, *819*
lieutenant, 641*pr*
LIFO, 823*ex*
LIMITED-BEST-FIT, *809*, 812*ex*
LIMITED-WORST-FIT, *811*, 812*ex*
line, *1017*
  direction vector, *1017*
  equation, *1017*
  place vector, *1017*
LINEAR-CLOSURE, *854*, 864, 865, 869
Linear Fit, 836
linear programming, 1109
linear speedup, *725*
line segment, *1017*
link, *753*, 758, 768
  directed, 758
link capacity, 648
List-Keys, *857*
list ranking problem, *733*
LIST-SCHEDULING, 824
literal, *891*
  negative, *891*
  positive, *891*
liveness condition, *592*
LOAD-LARGEST, *800*
LOAD-LARGEST-OR-LONG-WAITING, *802*
LOAD-LONG-WAITING-OR-NOT-SMALLER, *805*
local area network, 647
local control, *1022*
locality, *710*
local search, 1115
  with 2-exchange steps, *1099*, 1113, 1114
LOCAL-SEARCH-WITH-2-EXCHANGES-FOR-TSP, *1099*
local view, *787*
LOGARITHMIC-MERGE, 735
logical clock, *611*
LOGICAL-CLOCK, 612
logical implication, *850*, 872
logic program, *920*
log-odds, *980*
LogP, *720*
long-range dependency, 650
long-range dependent process, *675*
LONG-WAITING-OR-NOT-FIT-SMALLER, *804*, 812*ex*
lossless join, *859*
lower bound, *724*, *935*
LRU, *816*, *see* Least Recently Used
LRU-EXECUTES, *817*

M
M(k)-index, 958
M(k)-INDEX-CREATOR, *958*
M*(k)-index, 959
M*(k)-INDEX- NAIVE-EVALUATION, *959*
M*(k)-INDEX-PREFILTERED-EVALUATION, 960
main memory, *813*

main query, 961
MAN, *see* metropolitan area network
Management Information Base, 669
marching cubes algorithm, *1035*
matrix
  full, *794*
  unimodular, *765*
matrix product, 752*fig*, *753*, 762, 785*fig*
MATRIX-PRODUCT, *755*
matrix-vector product, *791*
MCD, *925*
MDLC, *see* Model Development Life Cycle
mediator system, *907*
memory, *813*
memory management, 797–848
merge, 735–739, 747*ex*
mesh, 747–749, *1027*
message
  graph, 628
  inquiring, 628
  notifying, 628
  range, 628
message complexity, *593*
messagereply, 628
method of invariants, *1082*
metric
  ultrametric, 998
metropolitan area network, 647
MIB, *see* Management Information Base
micro mutations, *1095*, 1101
Microsoft
  Access, 885
MiniCon Description, 925
MiniCon Description (MCD), *925*
MINIMAL-COVER, *855*, 870, 880*pr*
minimum evolution, *974*
mode
  active, 640
  calculation, 781
  propagation, 781
  relay, 640
model
  continuous, 644
  deterministic, 644
  discrete, 644
  dynamic, 643
  static, 643
  stochastic, 644
Model Development Life Cycle, 668, *669*
modell
  batched Markovian arrival, 676
  fluid-flow, 676
  Markovian arrival process, 676
  Markov-modulated Poisson-process, 676
model of computation, 723
Monte Carlo integration, *1115*
morphing, *1026*
MPMD, *see* Multiple Program Multiple Data
M*(*k*)-INDEX-EVALUATION-TOP-TO-BOTTOM, *959*
multicriteria decision making, 1116
multiple-choice algorithm, 1095
multiple-choice optimisation, *1094*, 1095
multiple-choice system, *1094*, 1095, 1100
  examples, *1095*
multiple-choice test, 1095
Multiple Program Multiple Data, 708
multiply-add, 757

N

NAIV-BCNF, *868*
NAIV-DATALOG, *895*, 922
NAIVE-APPROXIMATION, *952*
NAIVE-EVALUATION, *939*
naive index, *940*
NAIVE-INDEX-EVALUATION, *941*
NAIVE-MAXIMAL-SIMULATION, *935*
NAIVE-PT, *946*
natural join, *858*, 868, *886*
NEIGHBOR-JOINING, 1012
Neighbour-Joining, *1000*, *1003*
network, *592*
network attributes, *648*
network flow problems, 1109
network simulation, 643
Next Fit, 837
NEXT-FIT, *808*, 812*ex*
NF, *837*, *see* Next Fit
NFD, 839, *see* Next Fit Decreasing
NFU, 823*ex*
Node Editor, *659*
normal form, *865*
  BCNF, *866*, 880*pr*
  Boyce–Codd, 866
  Boyce-Codd, *866*
  5NF, *877*
  4NF, 866, *874*
  3NF, 866, *867*, 870, 880*pr*
notifying message, *628*
NP-complete problems, 1096, 1098
nr-datalog¬ program, *892*
NRU, *819*, 823*ex*
NRU-EXECUTES, *820*
NRU-PREPARES, *820*
NRU-SWAP-OUT, *821*
number of page faults, *814*
numerical instabilities, 1117
NURBS, *1023*

O, Ó

object median method, *1066*
oblivious algorithm, *737*
octree, *1065*
ODD-EVEN-MERGE, *736*, 745
ODD-EVEN-SORT, *745*
1-index, *942*, 965
1-INDEX-EVALUATION, *943*
OpenMP, 718
operation
  elementary, *757*
  input/output, *756*
operation mode, *778*, 782
OPNET, 659, 700
OPT-EXECUTES, *818*
optimal alignment, *975*
OPTIMAL-PREFIX, 728, *731*, 733
OPT-SWAP-OUT, *818*
ORDERED-BROADCAST, *621*, 632*ex*
origin, *1014*
orthogonal, *1013*
  vector, *1013*
other, 596

output
delayed, 780*fig*
output normal form, *788*
output of a processor, *606*
OUTSIDE, 991
outside world, 753, *758*, 760

P
packet, *647*
packet interarrival time, *679*
page, *813*
Page Frequency Fault, 822
page replacement algorithm
dynamic, 814
static, 814
page replacement algorithms,
page table, *814*
painter's algorithm, *1089*, 1092
Pairwise Fit, 838
Pairwise Fit Decreasing, 839
PAN, *see* personal area network
parallel, *1014*
line, *1017*
plane, *1016*
vector, *1014*
parallel computations, 702–750
parallelising compilers, 719
parallelism
directly express, 756
massive, 752
task, 708
parallel random access machine, *719*
parametric equation, *1016*
parametric problem, 761
parent, 594
partition, *797*
dynamic, 805
fixed, 798
PDA, *see* personal digital assistant
penalty function
additive penalties, *1102*, 1114
relative penalty factors, *1101*, 1104
penalty method, 1100, *1101*, *1103*
penalty method, iterative, *1108*
PENALTY-METHOD-FOR-THE-TSP-PROBLEM-WITH-2-EXCHANGE-HEURISTIC, *1113*
PENALTY-METHOD-WITH-RELATIVE-PENALTY-FACTORS, *1101*
penalty-optimal, *1104*
penalty parameter, optimal, 1107, 1113
performance measure, *725*
absolute, 724
relative, 724
performance metrics, *648*
period, *784*
permanent virtual circuit (PVC), *670*
personal area network, 646
personal digital assistant, 646
perturbation of local optima, *1100*
PF, *839*, *see* Pairwise Fit
PFD, *see* pairwise Fit Decreasing
PFF, *822*, *see* Page Frequency Fault
phase
ending, 628
regular, 628
pipelining, 752, 760, 761
pixel, *1013*
PLACE, *806*, 812*ex*
pointer jumping, *734*
point-to-point network, 646
Poisson point process, *1061*
polygon, *1027*
POLYGON-FILL, *1084*
polygon fill algorithm, *1082*
polyhedron, *1028*
polyline, *1027*
POLYNOMIAL-BCNF, *869*
port, *753*, 769
input, *753*, 762, 769
output, *753*, 762, 769
Post Correspondence Problem, 928*pr*
PRAM, *see* parallel random access machine
PRAM algorithm, 795
precedence graph, *896*, 923
prefix, 728, *731*
prefix computation, *728*
prefix computation on chain, 747
prefix computation on square, 748
prefix problem, *728*, 747
PREPARATA, *746*
present/absent bit, *814*
PRESERVE, *864*
priority PRAM, 720
problem parameter, *754*, 760, 766, 779, 785
Process Editor, *659*
processor
disseminator, 627
profile, *982*
Project Editor, *659*
projection, *763*, *886*
direction, *763*, 775, 793
matrix, *763*
vector, *763*, 787
projective
geometry, *1046*
line, *1049*
line segment, *1049*
plane, *1049*
space, *1046*
propagation mode, *781*
protocol overhead, 650
PT, *947*
PT-ALGORITHM, *943*
PVC, *see* permanent virtual circuit

Q
QBE, 885
QFD, *840*, *see* Quick Fit Decreasing
qos, *see* quality of service
QP, *see* Quick Power
QSM, *720*
QUADRATIC-SELECT, *740*
QUADRATIC-SORT, *744*
quadric, *1055*
quadtree, 1065*ex*
quality of service, *618*
quantification, *756*
query, *883*
conjunctive, 897
domain restricted, *889*
program, *887*

rule based, *884*
subgoal, *917*
empty, 900*ex*
equivalent, *883*
homomorphism, *898*, 909
language, 882
equivalent, *884*
mapping, *883*
monotone, *885*, 900*ex*
relational algebra, 900*ex*
rewriting, *909*
complete, 908, *909*
conjunctive, *918*
equivalent, *909*, 912
globally minimal, *909*
locally minimal, *909*
maximally contained, *912*
minimal, 908
rule based, 900*ex*
satisfiable, *885*, 900*ex*
subgoal, 924
tableau, *885*, 900*ex*
minimal, *898*
summary, *885*
variables of, *924*
query evaluation, *938*
query language, *938*
relationally complete, *891*
query rewriting, 882–930
Quick Fit Decreasing, 839
Quick Power, 840
Quick-Selection, *741*

R
RAM, 721, *see* random access machine
random access machine, 719
range message, *628*
rasterization, *1079*
ray, *1053*
Ray-First-Intersection, *1054*
Ray-First-Intersection-with-kd-Tree, *1069*
Ray-First-Intersection-with-Octree, *1065*
Ray-First-Intersection-with-Uniform-Grid, *1060*
ray parameter, *1054*
ray tracing, *1053*
rbb, *see* Reliable Basic Broadcast service
rco, *see* Reliable Causal Order
realtime scenario, 1095
recommender systems, *1097*
record, *849*
recurrence equation, *756*
uniform, 794
recursion, 893
recursive algorithm, 729
reference string, *814*, *815*
Refine, *957*
Refine-Index-Node, *957*
register, *769*
regular expression, *939*
regular phase, *628*
Regular-Phase, *629*
relation, 882
extensional, *885*, 894, 902
instance, 882, 883*fig*
intensional, *885*, 887, 894, 902
mutually recursive, *896*
virtual, *907*
relational
schema, *849*
decomposition of, *857*
table, *849*
relational algebra*, *886*
relational data bases,
relational schema, 882
relative number of steps, 725
relative speed, 747, *see* speedup
relay mode, 640*pr*
Reliable Basic Broadcast service, 619
Reliable Causal Order, 619
Reliable-Causally-Ordered-Broadcast, *623*, 632*ex*
Reliable Total Order, 619
relocatable programs, *797*
Remote Monitoring, 669
renaming, *886*
rendering, *1013*
reply message, *628*
reset, 779*fig*
response time, *649*
reverse engineering, 1092
right handed, *1078*
right hand rule, *1014*
right neighbour, *733*
RMON, *see* Remote Monitoring
RNA-foldings, *1098*
Rodrigues formula, *1052*, 1053*ex*
round, 729
round trip, *1096*
route planning, 1101, 1106
Routing protocols, 649
row-major indexing scheme, *748*
row of processors, 722
rssf, *see* Reliable Single-Source FIFO
rto, *see* Reliable Total Order
rule, 884
body, *884*
domain restricted, 891
head, *884*
realisation, *894*
run length, *671*
running time, 727
average, 724
expected, 724
in best case, 724
in worst case, 724
of systolic algorithm, 753, 755, 758, *767*, 792

S
safe, *941*
safety condition, *592*
Satisfiable, *890*
scalar control processor, 758
scan line, *1082*
schema
extensional, *894*
intensional, *894*
mediated, *907*
screen coordinate system, *1073*

search problem, *739*
SECOND-CHANCE, 823*ex*
segment, *813*
Selection, *739*, *886*
  condition, *886*
selection sort, 792
self-similarity, 674
self-similar process, *675*
SEMI-NAIV-DATALOG, *896*, 900*ex*, 922
semistructured databases, 931–971
sender, 618
separating plane, *1066*
sequence alignment, *974*
sequence alignment problem, 1103
sequentialisation, *712*
serialisation, *755*
shadow, *1053*
shape, *1013*
Shortening of lists, 841
shortest path problem, *1095*, 1103, 1109
shortlisting, 1097, 1098
short-range dependent process, *675*
shotgun sequencing, *1007*
side effect, 756
similar nodes, *934*
simple, *1027*
simple expression, *938*
simple graph, *625*
Simple Network Management Protocol, 669
simple polygon, *1027*
Simple Power, 840
simulated annealing, *1099*
simulation, *934*
simultaneity, 757
single connected, *1027*
Single Program Multiple Data, 708
single source consensus, 641*pr*
skew, 760
SL, *see* Shortening of lists
slow down, *780*
Slow-Lock, 639
slow start algorithm, 651
snapshot, 761*fig*, *772*
Sniffer, 668, 701
SNMP, *see* Simple Network Management Protocol
SNR, *see* signal to noise ratio
solid, *1015*
solve the leader election problem, *599*
sorting, 744–746
sorting algorithms, 792, 793
sorting problem, *744*
source description, *907*
source processor, *618*
SP, *see* Simple Power
space, *1013*
space coordinates, *754*, 763, 764
  parametric, 764–766
space-time
  matrix, 765, 777
  transformation, 761–770, *764*, 787
SPANNING-TREE-BROADCAST, *594*
spanning tree has ben constructed, *596*
spatial median method, *1066*
speedup, *725*, 747*ex*
  linear, 725
  sublinear, 725
  superlinear, 725
sphere, *1015*, *1016*
split, *945*
SPLIT-PARTITION, *806*
splitter, *945*
SPMD, *see* Single Program Multiple Data
spurious operation, *775*
SQL, 902
square, *722*
SQUARE-PREFIX, *748*
stable, *944*
*-RUN, *815*
state
  terminated, 593
state flip-flop, *782*
static page replacement algorithms, *814*
stationary
  matrix, *760*, 770
  variable, *760*, 769, 775, 779
strongly connected component, 897
subgoal, *917*
sublinear speedup, *725*
substitution, *897*
substring, *980*
successive fixing, *1116*
sum-of-pairs, *985*
sum type optimisation problem, *1103*
superkey, 850, *856*, 866
superlinear speedup, *725*
superstep, *721*
Sutherland-Hodgeman polygon clipping, 1092
SUTHERLAND-HODGEMAN-POLYGON-CLIPPING, *1042*
Sutherland-Hodgeman polygon clipping algorithm, *1042*
swap-in, 815
swap-out, 815
symbolic evaluation, 764
synchronous, *757*
  network, 795
System-R style optimiser, *914*
systolic, 751, *752*
  algorithm, *752*
  architecture, 758
  system, 751–795, *752*
  timestep, *757*
systolic array, 751–795, *752*
  architecture of, *764*
  border of, *760*, 766, 770
  boundary of, *775*, 791
  degenerated, *791*
  hexagonal, *762*, 766
  linear, *791*
  multi-dimensional, 760
  parameter of, *754*, 766, 779
  programmable, 787
  rectangular, 752*fig*, *753*, 762, 785*fig*
  shape of, 762, 765, 766
  size of, *754*, 765
  structure of, 752*fig*, *753*, 762*fig*, 785*fig*, 787, 791*fig*

T
task, 708, *825*
task parallelism, *708*

terminated algorithm, *593*
terminated state, *593*
termination, 593
tessellation, *1027*
  adaptive, *1031*
thread, 708
threads programming, 718
3D DDA algorithm, 1092
3D line drawing algorithm, *1059*
3-Hirn, *1097*, 1100
throughput, *784*
tiebreak situations, 1100
time complexity, *593*, *594*
timed execution, *593*
timestep, 758
  discrete, *757*
time vector, *763*, 769, 778, 787
topology, *592*
torus, *1015*
Tournament-Tree, *636*
trace-back, *976*
traffic engineering, 649
Transaction Interarrival Time, *679*
transformation, *1045*
  of domain, *787*, 788
  of index set, *788*
Transform-Expand-Sample models, 676
transitive closure, 893
translation, *1013*
Travelling Salesperson Problem (TSP), *1096*, 1099, 1103, 1113, 1117
Traverse, *939*
triangle, *1016*
  left oriented, *1085*
  right oriented, *1085*
triangular matrix
  lower, 793
tri-linear approximation, *1035*
triple brain, *see* 3-Hirn
tuple
  free, *884*
T vertex, *1032*
two ears theorem, *1030*, 1092
2-exchange, 1113, 1114
2-exchange, *1099*
two generals problem, *605*, 641
2-mutual exclusion, 640

U, Ú

ULE, *see* upper and lower estimations
ultrametric, *998*
uniform algorithm, 761
Uniform-Grid-Construction, *1058*
Uniform-Grid-Enclosing-Cell, *1058*
Uniform-Grid-Next-Cell, *1059*
Uniform-Grid-Ray-Parameter-Initialization, *1059*
upper and lower estimations, 841
upper bound, *935*
utilisation, *760*, 776, 778

V

value, 640*pr*
vector, *1013*
  absolute value, *1013*
  addition, *1013*
  cross product, *1014*
  dot product, *1013*
  multiplication with a scalar, *1013*
vectorization, *1027*
vector of the execution times, *825*
vector time stamp, *614*
vehicle routing, 1095, 1102
view, 882, *902*
  inverse, *920*
  materialised, *903*
virtual memory, 813
virtual world, *1013*
visibility problem, *1084*
voxel, *1034*

W

Want, 639
warm-up period, *671*
Warnock, *1088*
Warnock-algorithm, *1088*
Weight-Changer, 954, *955*, 956, 960*ex*
Weight-Increaser, *956*
wide area network, 647
work, *725*, 747
work-efficient algorithm, *726*
Working-Set, 821
work-optimal algorithm, *726*
worst case, 724, 727
Worst-Fit, *811*, 812*ex*
wrap around problem, *1077*
writing array, *814*
WS, *822*, *see* Working Set

X

XML, 905, *932*

Z

z-buffer, *1084*, *1085*
  algorithm, *1084*
Z-Buffer-Lower-Triangle, *1087*
zero-one principle, *737*

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
Abiteboul, Serge, 861, 877, 881, 930, 1119
Adams, J. A., 1129
Addario-Berry, L., 1119
Addie, R. G., 701
Addie, Ronald G., 1119
Afrati, Foto N., 1119
Aho, Alfred V., 875, 881, 1119
Ahuja, Ravindra K., 1110, 1119
Akutsu, T., 1119
Allen, Randy, 750
Althaus, E., 1119
Althöfer, Ingo, 578, 588, 1094, 1097, 1118, 1119
Altshuland, S. J., 1127
Amdahl, Gene M., 709, 750, 1119
Anick, D., 701, 1119
Arenas, Marcelo, 971, 1119
Arlazarov, V. L., 1119
Armstrong, William Ward, 865, 879, 881, 1119
Aspnes, James, 642, 1120
Atteson, K., 1120
Attiya, Hagit, 641, 642, 1120
Awerbuch, Baruch, 641, 1120

B
Back, Thomas, 1120
Bader, D. A., 1120
Bader, M., 1120
Baeck, Thomas, 1120
Baker, B., 1120
Balogh, Ádám, 588, 750, 797, 1120
Banks, Jerry, 700, 1120
Banzhaf, Wolfgang, 1118, 1120
Bayer, R., 1124
Beeri, Catriel, 872, 873, 875, 881, 1119, 1120
Békéssy, András, 856, 881, 1120
Bélády, László A., 824, 847, 1120
Belényesi, Viktor, 578, 588
Bello, Randall G., 930, 1120
Benczúr, A. András, 588
Benner, Steven A., 1120, 1124
Beran, J., 701, 1120
Berg, Daniel J., 750, 1127
Berger, Franziska, 1096, 1102, 1118–1120
Berman, Ken A., 749
Berman, Piotr, 608, 642, 1120
Bernstein, Felix, 1019
Bernstein, P. A., 881, 1120
Bestavros, Azer, 701, 1122
Beyer, W. A., 1131
Bézier, Pierre (1910–1999), 1019, 1026
Blinn, Jim, 1025, 1092, 1120
Bloomenthal, J., 1120
Blum, A., 1121
Bodon, Ferenc, 587
Bohannon, Philip, 971, 1126
Bourne, P. E., 1130
Boyce, Raymond F., 866, 881
Brent, R. P., 750
Bresenham, Jack E., 1080, 1093, 1121
Buneman, Peter, 930, 971, 1121
Burns, James E., 638, 641, 642, 1121
Busch, Costas, 1120

C
Calvanese, Diego, 930, 1121
Caprara, A., 1119, 1121
Carillo, H., 1121
Carson, J., 700, 1120
Catmull, Edwin, 1033, 1092, 1121
Chandra, Rohit, 750, 1121
Chaudhuri, Surajit, 930, 1121
Chazelle, Bernhard, 1092, 1121
Chen, Qun, 971, 1121
Chor, B., 1119, 1121
Choudhary, Alok, 750, 1126
Christie, D. A., 1121
Clapworthy, Gordon, 1132
Clark, James, 1033, 1092, 1121
Cochrane, Roberta, 930, 1132
Codd, Edgar F. (1923–2003), 849, 866, 881, 891, 1121
Coffman, Ed G., Jr., 847, 1121
Cohen, Mark A., 1120, 1124
Cormen, Thomas H., 749, 750, 848, 1121, 1122
Corneil, Derek G., 971, 1122
Corpet, F., 1122

Cox, J., 1131
Cox, M. G., 1022
Coxeter, Harold Scott MacDonald (1907–2003), 1091, 1122
Crovella, Mark E., 701, 1122
Culler, D. E., 750

CS
Csirik, János, 588
Csörnyei, Zoltán, 578, 587

D
Dagum, Leo, 750, 1121
Darte, Alain, 794, 1122
Dayhoff, M. O., 1122
de Berg, Marc, 1092, 1122
deBoor, Carl, 1022
De Giacomo, Giuseppe, 930, 1121
Delobel, C., 881, 1122
Demers, A., 1126
Demetrovics, János, 578, 588, 849, 856, 881, 882, 1120, 1122
Denning, Peter J., 847, 1122
Descartes, René, 1014
Deutsch, Alin, 930, 1122
Dias, Karl, 930, 1120
Dinic, E., 1119
Dolev, Danny, 608, 642, 1122
Dolev, Shlomi, 1120
Doolittle, R. F., 1123
Dowd, M., 881, 1120
Downing, Alan, 930, 1120
Dömösi, Pál, 578, 588
Dress, A., 1128
Drummond, A., 1127
Duff, Iain S., 1126
Duffield, N., 701, 1122
Duffy, D. E., 701, 1122
Durbin, Richard, 1128
Duschka, Oliver M., 930, 1119, 1122
Dwork, Cynthia, 641, 1120
Dyn, Niva, 1092, 1122

E, É
Eddy, Sean, 1129
Einstein, Albert (1879–1955), 757
Elek, István, 587
Elias, I., 1122
Englert, Burkhard, 578, 588, 591
Eramo, Vincenzo, 701, 1127
Erdős Pál (1913–1996), 1122
Erramilli, A., 701, 1123

F
Fagin, R., 872, 881, 1120, 1123
Faradzev, I. A., 1119
Farin, Gerald, 1092, 1123
Farkas, Gábor, 587
Feenan, James J., 930
Feiner, Steven K., 1123
Felsenstein, J., 1123
Feng, D., 1123
Fernandez, Mary, 930, 971, 1121, 1122
Fernando, Randoma, 1093, 1123
Fickett, J. W., 1123
Finnerty, James L., 930
Finnerty, T., 1120
Fischer, Michael J., 608, 610, 636, 641, 1123, 1127, 1129
Fischer, P. C., 881, 1131
Fitch, W. M., 1123
Florescu, Daniela D., 930, 1123
Floyd, S., 701, 1129
Flynn, Michael J., 749, 1123
Fogel, David B., 1120
Foley, James D., 1123
Fornai, Péter, 825, 847, 1123
Fortune, S., 750, 1123
Foster, Ian, 750, 1123
Foulds, L. R., 1123
Fountain, Terence J., 700, 795, 1130
Fourier, Jean Baptiste Joseph (1768–1830), 789
Freenan, J., 1120
French, K., 1128
Fridli, Sándor, 588
Friedman, Marc, 930, 1123
Fuchs, Henri, 1123
Fujimoto, A., 1092, 1123
Fülöp,Zoltán, 588

G
Gács, Péter, 578, 587
Gagne, Greg, 750, 847, 1130
Gál, Anna, 578, 588
Gal, T., 1123
Galántai, Aurél, 578, 587
Galil, Z., 1123
Gallant, J. K., 1123
Galvin, Peter Baer, 750, 847, 1130
Gararch, William, 750, 1123
Garay, Juan A., 608, 642, 1120
Garcia-Molina, Hector, 641, 1124
Garey, Michael R., 971, 1124, 1126
Garofalakis, Minos, 971, 1129
Gates, William Henry, 1124
Gécseg, Ferenc, 847, 1124
Gecsei, Jan, 847, 1128
Genesereth, Michael R., 930, 1122
Georgiou, Chryssis, 1120
Giancarlo, R., 1123
Gibbons, P. B., 750
Gibson, T. J., 1131
Giegerich, R., 1125
Glassner, A. S., 1124
Gnanaprakasam, Senthil, 930, 1127
Goldberg, David E., 1118, 1124
Goldman, N., 1124
Goldstein, Jonathan, 930, 1124
Golub, Evan, 750, 1123
Gonda, János, 578, 588
Gonnet, Haas Gaston Henry, 1120, 1124
Gotlieb, Calvin C., 971, 1122
Gotoh, O., 1124
Gottlob, Georg, 971, 1124
Graham, R. L., 1123
Graham, R. M., 1124
Graham, Ronald Lewis, 1126
Grahne, Gösta, 930, 1124
Grama, Ananth, 1124
Grant, John, 879, 881, 1124
Graur, D., 1129
Gray, James N., 641, 1124

Gregory, J., 1122
Gries, David, 641, 1128
Griggs, J. R., 1128
Gropp, William, 750, 1124
Gu, Q-P., 1124
Gudes, Ehud, 971, 1126
Gusella, Riccardo, 701, 1124
Gusfield, Daniel M., 1124
Gustafson, John L., 709, 750, 1124

GY
Gyires, Tibor, 578, 588, 643, 701, 1124

H
Halevy, Alon Y., 930, 1123–1125, 1129
Hallett, M., 1119
Hanne, T., 1123
Hannenhalli, S., 1125
Hartman, T., 1122
Hasegawa, A., 1131
Havran, Vlastimil, 1092, 1125
He, Hao, 959, 969, 971, 1125, 1132
Hefles, H., 701, 1125
Hein, J., 1126, 1127, 1129
Heinz, Ernst A., 1118, 1125
Henzinger, Monika Rauch, 971, 1125
Henzinger, Thomas A., 971, 1125
Herman, Iván, 1092, 1125
Heylighen, Francis, 701, 1125
Higgins, D. G., 1131
Hirschberg, D. S., 1125
Hodgeman, G. W., 1042, 1092, 1130
Holmes, Eddie C., 1129
Hopcroft, John E., 847, 1125
Horowitz, Ellis, 749, 750, 1125
Horváth, Zoltán, 587
Howard, J. H., 872, 881, 1120
Höchsman, M., 1125
Hörmann, Anikó, 578, 588
Hubbard, T. J. P., 1125
Hughes, John F., 1123
Hughey, R., 1125
Hull, Richard, 861, 877, 881
Hunt, J. W., 1125
Hurst, H. E., 675
Hwang, K., 1125

I, Í
Illés, Tibor, 587
Imreh, Csanád, 578, 587
Ioannidis, Yannis E., 906, 930, 1131
Iványi, Anna, 578, 588
Iványi, Antal, 578, 588, 589, 702, 749, 750, 797, 825, 847, 1120, 1123, 1125
Ivanyos, Gábor, 578, 587, 588

J
Jain, Anil K., 701
Jain R., 1125
Járai, Antal, 578, 587, 588
Jeney, András, 578, 587
Jensen, J. L., 1127
Jerrum, M. R., 1125
Jiang, T., 1121, 1126, 1131
Jiménez, P., 1126
Johnson, D. T., 856, 881, 1132
Johnson, David S., 971, 1126
Jones, D. T., 1124
Jones, Gareth A., 700, 1126
Jones, J. Mary, 700, 1126
Jones, Kit, 700, 1126

K
Kacsuk, Péter, 700, 795, 1130
Kambhampati, Subbarao, 930, 1127
Kandemir, Mahmut Taylan, 750, 1126
Kansei, I., 1123
Karlapalem, Kamalakar, 930, 1132
Karlin, Samuel, 1092, 1126
Karp, Richard M., 750, 794, 1126
Kása, Zoltán, 578, 587
Kátai, Imre, 587
Katona, Gyula O. H., 879, 881, 1122
Katsányi, István, 578
Kaushik, Raghav, 971, 1126
Kececioglu, J. D., 1126, 1127, 1129
Kedem, Z. M., 1123
Kellerman, Carl, 750
Kelton, W. David, 700, 1127
Kennedy, Ken, 750
Kesselman, Carl, 1123
Khosrow-Pour, Mehdi, 971, 1126
Kiss, Attila, 578, 588, 931
Kleiman, S., 750, 1126
Kleinrock, Leonard, 700, 1126
Kleitman, D. J., 1128
Knudsen, B., 1126
Knuth, Donald Ervin, 1126
Kobayashi, Y., 1131
Koch, Christoph, 971, 1124
Koelbel, C. H., 750, 1126
Kohr, Dave, 750, 1121
Kolaitis, Phokion G., 1119
Kopke, Peter W., 971, 1125
Korth, Henry F., 971, 1126
Kovács, Attila, 578, 587
Kovács, Előd, 1123
Kowalski, Dariusz, 578, 591
Krammer, Gergely, 1092, 1126
Krishnamurthy, Rajasekar, 971, 1126
Krishnamurty, Ravi, 930, 1121
Krogh, Anders, 1125
Kronrod, M. A., 1119
Kruskal, Clyde, 750, 1123
Kung, H. T., 794, 1126
Kurtz, S., 1125
Kwok, Cody T., 930, 1126

L
Labetoulle, Jacques, 1127
Lagergren, J., 1119, 1122
Lai, Ten Hwang, 847, 1126
Lakatos, László, 587
Lambert, Johann Heinrich (1728–1777), 1091
Lambrecht, Eric, 930, 1127
Lamperti, J., 1092, 1127
Lamport, Leslie, 606, 608, 635, 641, 642, 1127–1129
Lancia, G., 1127
Landau, G. M., 1127
Lapis, George, 930, 1132
Larson, Per–Åke, 930, 1124, 1132

Lauder, J., 1130
Law, Averill M., 700, 1127
Leighton, F. Tom, 750
Leiserson, Charles E., 749, 750, 794, 848, 1121, 1122, 1126
Leland, W., 700, 701, 1127, 1131
Lenhof, H. P., 1119, 1126
Lenzerini, Maurizio, 930, 1121
Leopold, Claudia, 578, 588, 642, 702, 750
Lesk, A. M., 1125
Levin, D., 1122
Levine, M. D., 1131
Levy, Alon Y., 930
Lewis, Bil, 750, 1127
Li, M., 1121, 1126
Li, Qing, 930, 1132
Libkin, Leonid, 971, 1119
Lim, Andrew, 971, 1121
Lipman, D., 1121, 1127
Listanti, Marco, 701, 1127
Locher, Kornél, 578, 588
Lovász, László, 586
Loveman, D. B., 750, 1126
Lucantoni, D., 1125
Lucchesi, C. L., 881, 1127
Lukács, András, 587
Lunter, G. A., 1127
Lusk, Ewing, 750, 1124
Lynch, Nancy Ann, 608, 610, 638, 641, 642, 1120, 1121, 1123, 1127
Lyngso, R., 1127

M
Magnanti, Thomas L., 1110, 1119
Maier, D., 1123
Maier, David, 880, 881, 1127
Malewicz, Grzegorz, 578, 588, 591
Manber, U., 1132
Mandelbrot, Benoit, 701, 1127
Markov, Andrej Andreyevitsch (1856–1922), 674
Martin, Ralph R., 1131
Márton, Gábor, 1092, 1130
Matias, Y., 750
Mattson, R. L., 847, 1128
Maxwell, E. A., 1128
Maydan, Dror, 750, 1121
Mayer, János, 578, 588
McCaskill, J. S., 1128
McDonald, Jeff, 750, 1121
McIntosh, A. A., 701, 1122
McPheeters, C., 1132
Mehlhorn, K., 1126
Melamed, B., 701, 1128
Mendelzon, Alberto O., 881, 930, 1123–1125, 1127
Menon, Ramesh, 750, 1121
Meskó, Attila, 588
Meyer, Albert R., 941, 971, 1130
Meyer, I. M., 1128
Meyer-Kahlen, Stefan, 1119
Michalewicz, Zbigniew, 1120
Mihnovskiy, S. D., 847, 1128
Miklós, István, 578, 588, 973, 1127
Miller, Raymond E., 794, 1126
Miller, W., 1128, 1132
Milo, Tova, 944, 971, 1128
Minker, J., 1125
Minker, Jack, 879, 881, 1124
Mitra, D., 1119
Molnár, Sándor, 701, 1128
Moret B. M. E., 1120
Morgenstern, B., 1128
Morris, J. H., 1126
Motwani, Rajeev, 847, 1125
Motzkin, Theodore Samuel, 789
Mutzel, P., 1126
Myers, E., 1131
Myers, E. W., 1126, 1128, 1132

N
Nagy, Marianna, 587
Narayan, O., 701, 1123
Naughton, Jeffrey F., 971, 1126
Naylor, B. F., 1123
Neame, T., 701
Neame, Timothy D., 1119
Needleman, S. B., 1128
Nelson, B., 700, 1120
Nelson, R. A., 824, 1120
Neuts, Marcel F., 701, 1128
Newell, M. E., 1128
Newell, R. G., 1128
Nick, Jeffrey M., 1123
Nilsson, Arne A., 701, 1128
Nitzberg, Bill, 750, 1124
Norcott, William D., 930, 1120
Nussinov, R., 1128

O, Ó
O'Rourke, Joseph, 1030, 1092, 1128
Oaks, Scott, 750, 1128
OConnell, N., 701, 1122
Ohlebusch, E., 1120
Olariu, Stephen, 750, 1131
Ong, Kian Win, 971, 1121
Orcutt, B. C., 1122
Ordille, Joann J., 930, 1124
Orlin, James B., 1110, 1119
Orosz, László, 578, 588
Osborne, Sylvia L., 856
Overmars, M., 1122
Owicki, Susan Speer, 641, 1128

P
Pachter, Lior, 1129
Page, Roderic D. M., 1129
Paige, Robert, 945, 971, 1129
Panagiota, Fatourou, 1120
Panconesi, A., 1119
Papadimitrou, Christos H., 1124
Pareto, Vilfredo, 655
Partridge, C., 701, 1129
Pataricza, András, 588
Paterson, M. S., 610, 1123
Patterson, D., 750
Paul, Jerome, 749
Paxson, V., 701, 1129, 1131
Peák, István (1938–1989), 847, 1124
Pease, Marshall, 606, 608, 642, 1127, 1129
Pedersen, C. N. S., 1127
Pedersen, J. S., 1129
Peer, I., 1129
Peng, S., 1124

Peterson, Gary, 636, 641, 1129
Pethő, Attila, 578, 588
Petrov, S. V., 881, 1129
Pfister, Gregory F., 750, 1129
Pichler, Reinhard, 971, 1124
Pieczenk, G., 1128
Pintér, Miklós Péter, 578, 588
Pirahesh, Hamid, 930, 1132
Pisanti, N., 1129
Poisson, Siméon-Denis (1781–1840), 1061, 1092
Polyzotis, Neoklis, 971, 1129
Popa, Lucian, 930, 1122
Potomianos, Spyros, 930, 1121
Pottinger, Rachel, 930, 1129
Pratt, V. R., 1126
Pupko, T., 1129

Q
Qian, Xiaolei, 930, 1129
Quinton, Patrice, 794, 1129

R
Rajaraman, Anand, 930, 1124
Rajasekaran, Sanguthevar, 749, 750, 1125
Ramachandran, V., 750
Ramakrishnan, Raghu, 971, 1126
Ramanujam, J., 750, 1126
Rao, Sailesh K., 794, 1129
Raschid, Louiqa, 930, 1123
Ravi, R., 1129
Recski, András, 578, 588
Reinert, K., 1119, 1126
Rényi, Alfréd (1921–1970), 700, 1129
Resnick, Paul, 1118, 1129
Rih, K. J., 1129
Rivas, E., 1129
Rivest, Ronald Lewis, 749, 750, 848, 1121, 1122
Robert, Yves, 794, 1122
Roberts, J. W., 1127
Roch, S., 1129
Rodrigues, Olinde, 1052, 1053
Rogers, D. F., 1129
Rolle, T., 1119
Rónyai, Lajos, 578, 587, 588
Roosta, Sayed H., 847, 1129
Rosenstein, M., 701, 1122
Ross, Sheldon N., 700, 1130
Rothe, Jörg, 578, 587
Routhier, S., 1125
Rustin, R., 1121

S
Sabella, Roberto, 701, 1127
Sagiv, Yehoshua, 881, 930, 1125, 1127
Sagot, M. F., 1129
Sahay, A., 750
Sahni, Sartaj, 749, 750, 847, 1126
Sali, Attila, 578, 588, 849, 881, 882, 1122, 1130
Sali, Attila, Sr., 1130
Sameith, Jörg, 1118, 1119, 1130
Sancha, T. L., 1128
Sankoff, D., 1130
Santaló, Luis A., 1130
Santoro, Nicola, 642, 1130
Santos, E. E., 750
Schauser, K. E., 750
Schneider, Csaba, 578, 588
Schreiber, Robert S., 750, 1126
Schumacker, R. A., 1130
Schwartz, R. M., 1122
Schwarz, Stefan, 578, 588, 1094, 1118, 1119
Schwarzkopf, O., 1122
Seegmuller, G., 1124
Segall, Adrian, 641, 1130
Seidel, R., 1130
Seiferas, J., 1124
Sellers, P. H., 1130
Shah, D., 750, 1126
Shamir, R., 1129
Sharp, Brian, 1092, 1130
Shedler, G. S., 824, 847, 1120
Shenoy, Pradeep, 971, 1126
Sherman, R., 701, 1120, 1131
Shim, Kyuseok, 930, 1121
Shindyalov, I. N., 1130
Shor, N. Z., 847, 1128
Shostak, Robert, 606, 608, 642, 1127, 1129
Shvartsman, Alexander Allister, 578, 588, 591, 1120
Sidló, Csaba, 587
Silberschatz, Avi, 750, 847, 1130
Sima, Dezső, 578, 700, 795, 1130
Slutz, D. R., 847, 1128
Smaalders, B., 750, 1126
Smith, T. F., 1130
Smithand, T. F., 1131
Snir, Marc, 750, 1124
Solomon, Marvin H., 906, 930, 1131
Sondhi, M. M., 1119
Song, Y. S., 1127
Spirakis, Paul, 1120
Spouge, J. L., 1130
Sproull, R. F., 1130
Srivastava, Divesh, 930, 1124, 1125
Stanoi, Ioana, 969, 971, 1132
Statman, R., 881
Steel, Mike, 1122
Steele Guy L., Jr., 750, 1126
Stein, Clifford, 750, 848, 1122
Stewart, G. W., 1126
Stewart, T. J., 1123
Stockmeyer, Larry, 641, 1120
Stockmeyer, Larry J., 941, 971, 1130
Storer, J. A., 1123
Strassen, Volker, 727
Strong, Ray, 608, 642, 1122
Sturmfels, Bernd, 1129
Subramonian, E., 750
Suciu, Dan, 930, 944, 971, 1121, 1122, 1128
Sudborough, H. I., 1124
Sun, Harry, 930, 1120
Sutherland, Ivan E., 1042, 1092, 1130

SZ
Szántai, Tamás, 578, 588
Szécsi, László, 1092, 1130
Szeidl, László, 587
Székely, László, 1122
Szidarovszky, Ferenc, 578, 587
Szirmay-Kalos, László, 578, 588, 1130
Szirmay-Kalos László, 1013

Szmeljánszkij, Ruszlán, 847, 1125
Sztrik, János, 578
Szymanski, T. G., 1125

T
Takayuki, T., 1123
Tamm, Ulrich, 578, 587
Tanenbaum, Andrew S., 700, 750, 847, 1130
Tannen, Val, 930, 1122
Taqqu, Murad S., 700, 701, 1120, 1127, 1130, 1131
Tarhio, J., 1130
Tarjan, Robert Endre, 945, 971, 1129
Taylor, Brook, 1017
Taylor, M. T., 1092, 1126
Teich, Jürgen, 794, 1131
Tejfel, Máté, 587
Tel, Gerard E., 642, 1131
Telek, Miklós, 587
Terlaky, Tamás, 587
Teverovsky, W., 1130
Thalheim, Bernhardt, 881, 1131
Thiele, Lothar, 794, 1131
Thomas, F., 1126
Thompson, J. D., 1131
Thorne, J. L., 1124
Tinoco, I., 1131
Tompa, Frank Wm., 856
Torras, C., 1126
Töller, T., 1125
Traiger, I. L., 847, 1128
Tramontano, A., 1125
Tromp, J., 1121
Tsatalos, Odysseas G., 906, 930, 1131
Tsou, D. M., 881, 1131
Tucker, Alan B., 971, 1131
Tuecke, Steven, 1123
Tuller, T., 1121
Turner, Douglas H., 1118

U, Ú
Uemura, Y., 1131
Uhlenbeck, O. C., 1131
Ukkonen, 1129
Ukkonen, E., 1130
Ukkonen, Esko, 1131
Ullman, Jeffrey David, 847, 861, 875, 881, 930, 1119, 1125, 1126, 1131
Urata, Monica, 930, 1132

V
Valduriez, Patrick, 930, 1123
Valiant, L. G., 750
van Dam, Andries, 1123
van Kreveld, M., 1122
Van Ness, John W., 701, 1127
van Steen, Maarten, 750, 1130
Várady, T., 1131
Vardi, Moshe Y., 930, 1121
Varga, László, 578
Varian, Hal R., 1118, 1129
Vasarely, Victor, 578
Vianu, Victor, 861, 877, 881, 970, 1119, 1131
Vida, János, 578
Vidács, Attila, 701, 1128
Vingron, M., 1126
Vishkin, U., 1127
Vivien, Frédéric, 794, 1122
Vizvári, Béla, 578, 587, 588
von Eicken, T., 750
Vörös, Veronika, 578, 588

W
Wang, L., 1131
Wareham, T., 1119
Warnock, John, 1088
Warnow, T., 1122
Warren, Joe, 1092, 1131
Waterman, Michael S., 1130, 1131
Wattenhofer, Roger, 1120
Weber, E., 1131
Weimer, Henrik, 1092, 1131
Welch, Jennifer Lundelius, 641, 642, 1120
Weld, Daniel S., 930, 1123, 1126
Wenn, H. Josef, 701
Werner, T., 1128
Willinger, W., 700, 701, 1120, 1122, 1123, 1127, 1130, 1131
Wilson, D., 700, 701, 1127, 1131
Winkler, Zoltán, 1123
Winograd, Shmuel, 794, 1126
Witkowski, Andrew, 930, 1120
Wolfe, Michael J., 750, 1131
Wong, Henry, 750, 1128
Woodhull, Albert S., 750, 847, 1130
Wu, Jie, 750, 1131
Wu, S., 1132
Wunsch, C.D., 1128
Wyllie, J., 750, 1123
Wyvill, Brian, 1132
Wyvill, Geaff, 1132

X
Xu, Z., 1125

Y
Yan, M., 1120
Yang, H. Z., 930, 1132
Yang, Jian, 930, 1132
Yang, Jun, 959, 969, 971, 1125, 1132
Yannakakis, M., 1121
Yi, Ke, 969, 971, 1132
Yokomori, T., 1131
Yu, C. T., 856, 881, 1132
Yussupov, Arthur, 1097, 1118

Z
Zaharioudakis, Markos, 930, 1132
Zalik, Bornt, 1132
Zaniolo, C., 881, 1132
Zehendner, Eberhard, 578, 588, 751, 795, 1132
Ziauddin, Mohamed, 930, 1120
Zosel, Mary E., 750, 1126
Zuker, Michael, 1118, 1127
Zukerman, M., 701
Zukerman, Moshe, 1119

The book *Algorithms of Informatics* consists of two volumes: *Volume 1. Foundations* and *Volume 2. Applications.*

The second volume contains eleven chapters. These chapters are divided into three parts. The chapters of Part 4 are connected with computer networks: *Distributed Algorithms* (written by 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), *Systolic Systems* (Eberhard Zehendner, Friedrich Schiller University).

Part 5 consists of the following chapters: *Memory Management* (Ádám Balog, Antal Iványi, Eötvös Loránd University), *Query in Relational Data Bases* and *Relational Databases* (János Demetrovics, Computer and Automation Research Institute; Attila Sali, Alfréd Rényi Institute of Mathematics), *Semi-structured Data Bases* (Attila Kiss, Eötvös Loránd University). These chapters are devoted to algorithms of data bases.

The chapters of Part 6 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 were validated by Dezső Sima (Budapest Polytechnical Institution), István Majzik, and Tamás Szántai (Budapest University of Technology and Economics), András Benczúr, István Katsányi, Attila Kiss, László Varga, János Vida (Eötvös Loránd University), János Sztrik (University of Debrecen).

The second volume contains verbal description, pseudocode and analysis of over 110 algorithms, and also about 250 figures illustrating how the algorithms work. Each section ends with exercises and each chapter ends with problems. We have included about 190 exercises and over 40 research 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 978-963-87596-2-7