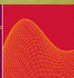
Quantum Computing · Neural Networks · Evolutionary Computing · DNA Computing



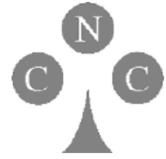
NATURAL COMPUTING SERIES

Hans-Joachim Böckenhauer · Dirk Bongartz

Algorithmic Aspects of Bioinformatics

Springer

Natural Computing Series



Series Editors: G. Rozenberg
Th. Bäck A.E. Eiben J.N. Kok H.P. Spaink

Leiden Center for Natural Computing

Advisory Board: S. Amari G. Brassard K.A. De Jong
C.C.A.M. Gielen T. Head L. Kari L. Landweber T. Martinetz
Z. Michalewicz M.C. Mozer E. Oja G. Păun J. Reif H. Rubin
A. Salomaa M. Schoenauer H.-P. Schwefel C. Torras
D. Whitley E. Winfree J.M. Zurada

Hans-Joachim Böckenhauer
Dirk Bongartz

Algorithmic Aspects of Bioinformatics

With 118 Figures and 9 Tables

Authors

Dr. Hans-Joachim Böckenhauer
ETH Zurich
Information Technology and Education
CAB F11
Universitätsstr. 6
8092 Zurich
Switzerland
hjb@inf.ethz.ch

Dr. Dirk Bongartz
Computer Science I
RWTH Aachen
Germany
bongartz@cs.rwth-aachen.de

Series Editors

G. Rozenberg (Managing Editor)
rozenber@liacs.nl

Th. Bäck, J.N. Kok, H.P. Spaink
Leiden Center for Natural Computing
Leiden University
Niels Bohrweg 1
2333 CA Leiden, The Netherlands

A.E. Eiben
Vrije Universiteit Amsterdam
The Netherlands

Library of Congress Control Number: 2007924247

ACM Computing Classification (1998): F.2.2, G.2.1, G.2.2, J.3

ISSN 1619-7127
ISBN 978-3-540-71912-0 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable for prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2007

Originally published in the German language by B.G. Teubner Verlag as
"Hans-Joachim Böckenhauer und Dirk Bongartz: Algorithmische Grundlagen der Bioinformatik".
© B.G. Teubner Verlag | GWV Fachverlage GmbH, Wiesbaden 2003

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover Design: KünkelLopka, Werbeagentur, Heidelberg
Typesetting: by the Authors
Production: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig

Printed on acid-free paper 45/3100/YL 5 4 3 2 1 0

Preface

The discovery of the double-helix structure of DNA by Watson and Crick more than fifty years ago was the starting point of a new era in molecular biology. Since then, our knowledge of biological structures and processes has grown tremendously. But many of these advances would have been unthinkable without using computational methods. Computer science plays a leading role in the emerging interdisciplinary field of bioinformatics. Only the interplay between biological methods and concepts from informatics has enabled us to successfully maintain projects such as the Human Genome Project. But completing this project also initiated further challenges for bioinformatics. The main goal now is to analyze and make use of the collected data. Moreover, new areas are explored, for instance, the prediction of the spatial structure of proteins. These results will be useful for designing new drugs and improved medical therapies; in this context, computer science faces bold challenges. However, progress in molecular biology also influences the design and development of computer science methods and concepts, as in the exciting research field of molecular computing.

This book introduces some of the fundamental problems from the field of bioinformatics; it discusses the models used to formally describe the problems, and it analyzes the algorithmic approaches used to attack and eventually solve them. This book can be regarded as a textbook, describing the topics in detail and presenting the formal models in a mathematically stringent yet intuitive way. Thus, it is well suited as an introduction into the field of bioinformatics for students or for preparing introductory lectures. The spectrum of topics includes classical subjects such as string algorithms, fundamental approaches for sequencing DNA and for analyzing the sequencing data, but also more recent subjects such as structure prediction for biomolecules and haplotyping models. We have tried to present some fundamental ideas and approaches for each of the topics in as much detail as possible, and also to give some insight into current research.

Without help from others, it would have been much harder, if not impossible, to write this book. So we would like to thank everyone who helped us in one way or another.

Above all, we would like to thank Juraj Hromkovič for encouraging us to write this textbook, for supporting us with many helpful suggestions and comments, and, last but not least, for carefully proofreading previous versions of the manuscript. We are grateful to Heidi Imhoff for her comments and advice concerning the biological basics, and to Mark Cieliebak for discussions and comments on restriction site mapping. Our cordial thanks to our colleagues Joachim Kupke, Sebastian Seibert, Walter Unger, and Viktor Keil for helpful advice and for the discussion on numerous topics, as well as for their support in the technical realization of this book. Moreover, we thank all who helped us to improve on the previous (German) version of the book by sending us their comments and pointing out some mistakes. In particular, we would like to thank Katharina Balzer, Frank Kurth, and Nan Mungard. A special thanks goes to the team at Springer, particularly to Ronan Nugent, for his support. Finally, we would like to express our thanks to our families and friends, for their enduring support and motivation during our work on this book. Cordial thanks to all of you!

Zürich and Mönchengladbach, January 2007

Hans-Joachim Böckenhauer
Dirk Bongartz

Contents

1 Introduction 1

Part I Introduction and Basic Algorithms

2 Basics of Molecular Biology 7
2.1 Proteins 7
2.2 Nucleic Acids 9
2.3 Hereditary Information and Protein Biosynthesis 12
2.4 Experimental Techniques 15
2.4.1 Basic Terms and Methods 15
2.4.2 Duplication of DNA 15
2.4.3 Gel Electrophoresis and Direct Sequencing 16
2.4.4 DNA Chips 19
2.5 Bibliographic Notes 20

3 Basic Concepts: Strings, Graphs, and Algorithms 23
3.1 Strings 23
3.2 Graphs 25
3.3 Algorithms and Complexity 28
3.4 Bibliographic Notes 35

4 String Algorithms 37
4.1 The String Matching Problem 37
4.2 String Matching Automata 39
4.3 The Boyer–Moore Algorithm 44
4.4 Suffix Trees 50
4.5 Further Applications of Suffix Trees 58
4.5.1 Generalized Suffix Trees and the Substring Problem . . . 58
4.5.2 Longest Common Substrings 61
4.5.3 Efficient Computation of Overlaps 63

4.5.4 Repeats in Strings ... 66
4.6 Suffix Arrays ... 68
4.7 Summary ... 77
4.8 Bibliographic Notes ... 78

5 Alignment Methods ... 81
5.1 Alignment of Two Strings ... 82
5.1.1 Basic Definitions ... 82
5.1.2 Global Alignment ... 84
5.1.3 Local and Semiglobal Alignment ... 89
5.1.4 Generalized Scoring Functions ... 94
5.2 Heuristic Methods for Database Search ... 97
5.2.1 The FASTA Heuristic ... 98
5.2.2 The BLAST Heuristic ... 99
5.3 Multiple Alignments ... 101
5.3.1 Definition and Scoring of Multiple Alignments ... 101
5.3.2 Exact Computation of Multiple Alignments ... 104
5.3.3 Combining Pairwise Alignments ... 109
5.4 Summary ... 114
5.5 Bibliographic Notes ... 114

Part II DNA Sequencing

6 Introduction and Overview ... 119

7 Physical Mapping ... 123
7.1 Restriction Site Mapping ... 123
7.1.1 The Double Digest Approach ... 124
7.1.2 The Partial Digest Approach ... 131
7.1.3 Comparison of Methods for Restriction Site Mapping ... 141
7.2 Hybridization Mapping ... 143
7.2.1 Mapping with Unique Probes ... 146
7.2.2 Mapping with Unique Probes and Errors ... 157
7.2.3 Mapping with Non-unique Probes ... 165
7.3 Summary ... 166
7.4 Bibliographic Notes ... 168

8 DNA Sequencing ... 171
8.1 Shotgun Sequencing ... 171
8.1.1 Crucial Points to Be Considered in a Suitable Model ... 174
8.1.2 The Shortest Common Superstring Problem ... 176
8.1.3 Refined Models for Fragment Assembly ... 196
8.2 Sequencing by Hybridization ... 201
8.3 Summary ... 207

8.4 Bibliographic Notes ... 208

Part III Analyzing Biological Data

9 Finding Signals in DNA Sequences ... 213
9.1 Identical and Similar Substrings ... 213
9.2 Tandem Repeats ... 217
9.3 Frequent and Infrequent Substrings ... 223
9.4 Hidden Markov Models ... 228
9.5 Summary ... 235
9.6 Bibliographic Notes ... 235

10 Genome Rearrangements ... 237
10.1 Modeling ... 237
10.2 Sorting Undirected Permutations ... 239
10.3 Sorting Directed Permutations ... 247
10.4 Computing the Syntenic Distance ... 249
10.5 Summary ... 255
10.6 Bibliographic Notes ... 255

11 Phylogenetic Trees ... 257
11.1 Ultrametric Distances ... 258
11.2 Additive Trees ... 265
11.3 Characters with Binary States ... 268
11.4 The Parsimony Principle and the Quartet Method ... 275
11.5 Summary ... 283
11.6 Bibliographic Notes ... 285

12 Haplotyping ... 287
12.1 Inferring Haplotypes from a Population ... 288
12.2 Haplotyping a Single Individual ... 305
12.3 Summary ... 316
12.4 Bibliographic Notes ... 316

13 Molecular Structures ... 319
13.1 RNA Secondary Structure Prediction ... 320
13.1.1 Minimizing the Free Energy ... 322
13.1.2 Stochastic Context-Free Grammars ... 329
13.2 Structure-Based Comparison of Biomolecules ... 337
13.3 Protein Structure Prediction ... 349
13.3.1 De Novo Structure Prediction — The HP Model ... 352
13.3.2 Protein Threading ... 363
13.4 Summary ... 369
13.4.1 RNA Secondary Structure Prediction ... 369

13.4.2 Structure-Based Comparison of Biomolecules 371
13.4.3 Protein Structure Prediction 371
13.5 Bibliographic Notes 372
13.5.1 RNA Secondary Structure Prediction 372
13.5.2 Structure-Based Comparison of Biomolecules 373
13.5.3 Protein Structure Prediction 374

References 377

Index 389

1

Introduction

Topics like biotechnology, genetics, and bioinformatics frequently find their way into today's headlines. This trend, often seen as the beginning of a new era, was initiated by a seminal discovery more than fifty years ago, the discovery of the DNA double helix structure by Watson and Crick in 1953. The development in molecular biology continued to grow steadily from then, and it became more and more important to the public, for example, with the launch of the Human Genome Project in the 1990s and with the presentation of the cloned sheep "Dolly." In 2000, scientists eventually announced the complete sequencing of the human genome.

Let us take a closer look at the task of DNA sequencing. The goal here is to determine the sequence of nucleotides, which are the elementary building blocks in the human DNA, i.e., in the molecule storing our hereditary information. From the viewpoint of informatics, we are looking for a string made up from letters representing these nucleotides. Methods for reading their sequence have already been known since the 1980s, but the length of DNA molecules that can be sequenced using these methods is severely restricted. Current methods enable us to read about 1 000 consecutive nucleotides. Nevertheless, we are mainly interested in DNA molecules consisting of hundreds of thousands of them. How can we reach our goal of sequencing in these cases? One possible approach is the following: We generate a multitude of copies of the DNA molecule of our interest. We randomly break each of these copies into fragments. With high probability, the resulting fragments from different copies overlap with each other. Ideally, these fragments are sufficiently short for direct sequencing. Having performed these sequencing operations, we are left with many string fragments of which we know that they occur as substrings in the DNA sequence, and that these fragments may overlap with each other. But we have no clue how to combine these fragments to achieve the complete DNA sequence, since their order was lost in this process. Typically, we have to reorder thousands of string fragments, a task we are not capable of doing by hand. This is the point where informatics comes in. First of all, it gives us the appropriate tools for managing the data, i.e., the set of string fragments.

Moreover, it helps us to formalize how a clever reconstruction of the DNA sequence would look. One possible example of such a formalization could be to search for the shortest string containing all fragments as substrings. Then we could try to solve the formalized problem with computational methods. But this task is not as easy as it may seem. Due to the enormous number of possible fragment orderings, we cannot rely on computational power alone; even this very intuitive and apparently simple problem leaves us with a great algorithmic challenge. Furthermore, even if we somehow succeed in finding this shortest string containing all fragments, the question remains open as to whether this string really coincides with the prospected DNA sequence, or whether we have to look for some refined model incorporating further aspects of the problem.

In this book, we will deal with questions of this type. In particular, we will describe how to derive a formal model from a biological problem, and how to find a solution to this formal model algorithmically.

Research problems in molecular biology typically do not arise with a given completely formal description. In contrast, one has on the one hand a rather concrete idea of the desired result, but on the other hand only a vague intuition of how this goal could be achieved. Different biological or biotechnological approaches or methods in use can lead to the need for completely different computational methods. Understanding the biological methods and approaches in detail is in many cases possible for biologists only, but knowledge about the basic principles and connections is very helpful for the computer scientists dealing with these kinds of problems. Such knowledge alone enables them to understand the real problems at hand and to possibly suggest modifications to the methods that could help to make the underlying computational tasks easier.

One major concern is that all biological data is inherently inexact. Experimental methods are always error prone, and these errors have to be taken into account in further processing steps. On the other hand, every *solution* derived by computational methods for some biological problem is only a *solution hypothesis* in reality. Only by further investigations does it become clear whether such a hypothesis really induces a biologically relevant solution or whether different aspects of the original problem have to be incorporated into the formal model to gain modified or enhanced solution hypotheses.

To be treated using algorithmic methods, a biological problem has to be transformed into a formal model, i.e., into a formal specification of the problem identifying in particular the data at hand and the desired result. Without such a formal model, it remains unclear how to reach the desired goal using computational methods. In this context, each developed model has to be evaluated according to its ability to describe the relevant real-life aspects of the given biological problem. For example, a common model of the DNA molecule consists of a description of the linear sequence of the nucleotides in terms of a string, as described above. In most cases, this model is sensible and offers a convenient possibility for further processing using computational methods.

But a DNA molecule is no string! For instance, if we want to examine the spatial properties of a DNA molecule, modelling it as a string is clearly not sufficient, and we have to look for new or enhanced models.

When we eventually succeed in finding an appropriate formal model for our biological research problem, we can start to look for algorithmic approaches to solving this formal problem. Even then, we should take the biological realities as a guideline for our examinations. As an example, we will again consider the string model of DNA molecules. Although strings can in general be composed of arbitrarily many different letters, we do not have to take this into account for our considerations, since the DNA is made up of only *four* different nucleotides.

With this reasoning, we will try throughout this book to proceed using the scheme

problem – model – algorithm

This means that we will, for every biological problem, describe the corresponding formal model, or even several models, and discuss their advantages and shortcomings. Only after that will we examine the algorithmic properties of the resulting formal models. Nevertheless, the focus of our attention will lie on the formal models and the algorithmic approaches for finding solutions for them, but not without keeping an eye on the underlying biological applications.

The complete spectrum of computational methods is used for investigating problems from molecular biology. In particular, methods from the fields of database management, statistics, and algorithmics are used. In this book, we will focus on the algorithmic aspects. In the subsequent chapters, we will also present some of the basic concepts from algorithm theory, for example, algorithm design methods like dynamic programming, divide and conquer, backtracking, branch and bound, and many more. We will in particular consider the concept of approximation algorithms as it arises in the field of combinatorial optimization. But all the presented methods will be embedded into the context of actual biological problems and are illustrated using concrete examples.

This book is designed as a self-contained textbook. Its goal is to explain the basic principles of algorithmic bioinformatics to students, and to help lecturers prepare introductory courses on the subject. This book is primarily targeted at graduate and advanced undergraduate students of computer science and at life sciences students interested in algorithmics. To successfully read this book, only very basic knowledge of data structures and algorithms is assumed. We try to cover a multitude of problems, and to present an overview of the models and the methods used for solving them. In this way, we hope to impart a solid basis of knowledge enabling readers to build on and intensify their studies in their respective research fields. Nevertheless, we have to mention that it is impossible for an introductory textbook to cover all topics in depth; but we hope that our choice of topics stimulates the reader's interest.

This book is divided into three parts. The first part serves as an introduction to the field of bioinformatics. After a short overview of the biological background and the basic notions of algorithmics, we will deal here with algorithmic methods concerning strings, for instance, string matching algorithms and methods for computing the similarity of strings. The second part is devoted to the field of DNA sequencing, which provides an important source of data for further investigations. In this part, we deal with generating physical maps and with different approaches and models of the actual DNA sequencing process. This is followed by the third part, in which we analyze a multitude of various problems arising in molecular biology. Among other problems, we deal with signal finding in DNA sequences, phylogenies, haplotyping, and the computation of spatial molecular structures. Furthermore, each chapter contains, besides those sections in which we present the problems and their solutions in detail, one section summarizing the presented material. This overview of results assists the readers in self-checking their understanding of the presented material. Moreover, every chapter concludes with a section pointing to the papers and books we used for preparing the chapter as well as to additional literature, and thus directs the interested reader to sources for further study.

We have tried to motivate and to describe the presented topics as accurately as possible. We appreciate any comments and suggestions as well as any information about remaining errors. Please note the following website:

`http://www.ite.ethz.ch/publications/bioinformatics`

Finally, we hope that we will be able to create interest and excitement for the field of bioinformatics. Please enjoy reading!

Part I

Introduction and Basic Algorithms

2 Basics of Molecular Biology

If one wants to consider questions in the area of molecular biology, as we do in this book, it is an obligatory prerequisite for the development and the evaluation of abstract models and techniques to have at least a basic knowledge about the fundamental principles of molecular biology. We therefore devote this chapter to fundamentals concerning the topics of subsequent chapters, and especially to the classes of biologically most relevant molecules, namely proteins and nucleic acids. The description will be an abstraction and recapitulation of biological knowledge only, and does not claim to be comprehensive or fully detailed. It instead provides an overview of the basic relations in molecular biology serving as a solid background for the problems we are going to consider later in this book. We will thus omit all details that are not necessary for understanding the text.

We start this chapter by the description of proteins in Section 2.1 and nucleic acids in Section 2.2, and their interaction for protein biosynthesis in Section 2.3. Afterwards, we will present some standard techniques used for the analysis of nucleic acids in Section 2.4, and we complete the chapter with some bibliographic notes in Section 2.5.

2.1 Proteins

Proteins represent one of the most important of the molecule classes in living organisms. Their functions include the catalysis of metabolic processes in the form of enzymes; they play an important role in signal transmission, defense mechanisms, and molecule transportation; and they are used as building material, for example in hair.

Proteins are chains of smaller molecular entities, so-called *amino acids*, which consist of a central carbon atom, denoted as C_α, connected to an amino group (NH_2), a carboxyl group ($COOH$), and a side chain (R), which is specific for the particular amino acid. The fourth free binding site of C_α is saturated by a single hydrogen (H) atom. In Figure 2.1, this general structure

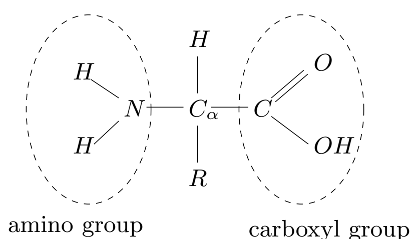


Fig. 2.1. Structure of an amino acid consisting of a central carbon C_α together with an amino and a carboxyl group, and a specific side chain R

is shown. According to this, the particular amino acids only differ with respect to their side chains, which also determine their chemical characteristics. We will, however, not consider the detailed chemical structure of the side chains any further.

In nature, there are several known amino acids, but only twenty of them serve as standard building blocks of proteins; these are given in Table 2.1. Next to the names of the amino acid, the corresponding abbreviation and the so-called one letter code is shown. Furthermore, we refer to the polarity of the amino acids, i.e., to their affinity to water, which will play an important role in Section 13.3.

name	Alanine	Valine	Leucine	Isoleucine	Phenylalanine
abbreviation	Ala	Val	Leu	Ile	Phe
code	A	V	L	I	F
polarity	(H)	(H)	(H)	(H)	(H)
name	Proline	Methionine	Serine	Threonine	Cysteine
abbreviation	Pro	Met	Ser	Thr	Cys
code	P	M	S	T	C
polarity	(H)	(H)	(P)	(P)	(P)
name	Tryptophan	Tyrosine	Asparagine	Glutamine	Aspartic acid
abbreviation	Trp	Tyr	Asn	Gln	Asp
code	W	Y	N	Q	D
polarity	(H)	(P)	(P)	(P)	(P)
name	Glutamic acid	Lysine	Arginine	Histidine	Glycine
abbreviation	Glu	Lys	Arg	His	Gly
code	E	K	R	H	G
polarity	(P)	(P)	(P)	(P)	(P)

Table 2.1. The 20 standard amino acids and their affinity to water. Amino acids that are polar and thus have the property of establishing hydrogen bonds with water are called hydrophilic (P), while nonpolar amino acids are called hydrophobic (H)

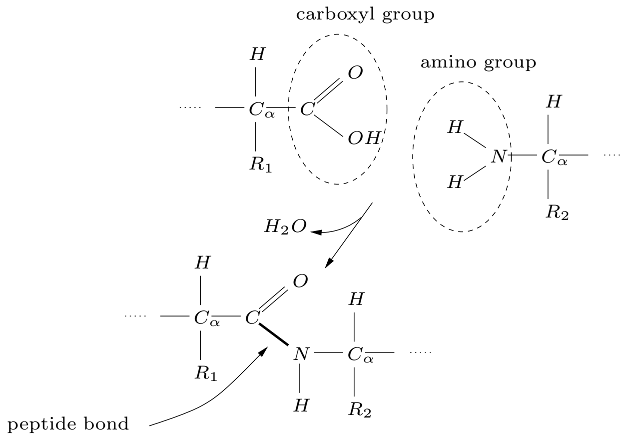


Fig. 2.2. Schematic view of a peptide bond between two amino acids

The amino acids may link to each other by so-called peptide bonds. There, a chemical bond is established between an amino group and a carboxyl group releasing a water molecule. The schematic view of a peptide bond is shown in Figure 2.2. These peptide bonds lead to a linear ordering of the amino acids, forming a polypeptide chain. The backbone of this chain is formed following the pattern

amino group — central carbon — carboxyl group — amino group

where the side chains of the amino acids are attached to the central carbon.

We may consider proteins essentially as polypeptide chains.[1] Hence, the essential information about a protein can be described in terms of the amino acid sequence along the chain. Therefore, we may represent a polypeptide chain as a string, writing down the amino acids using the one letter code and fixing the reading direction from the end with the free amino group to the end with the free carboxyl group. Correspondingly, we may also cast a protein as a string. This string structure is referred to as the *primary structure* of the protein. At the moment, we will ignore the spatial structure of the protein, i.e., the positions of the single atoms in three-dimensional space, but we will reconsider this in Section 13.3, where we will deal with the determination of protein structures.

2.2 Nucleic Acids

Beside proteins, nucleic acids probably form the most important type of molecule in organisms. In all living organisms, nucleic acids are responsible

[1] Some proteins also consist of several polypeptide chains or may be supplemented by other molecular structure, but we will abstract from this fact.

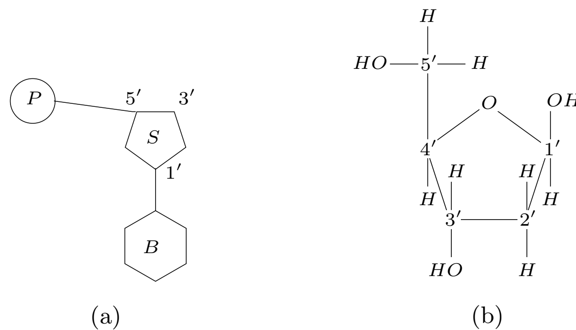


Fig. 2.3. (a) Schematic view of a nucleotide consisting of a phosphate (P), a sugar (S), and a base (B). (b) The detailed structure of the sugar deoxyribose, carbon atoms are numbered $1'$ to $5'$

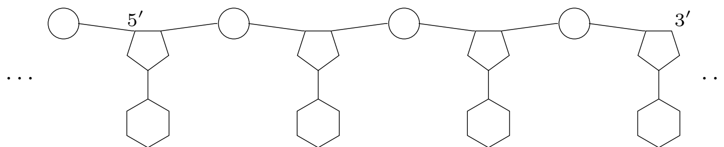


Fig. 2.4. Schematic view of a nucleotide chain in the $5'$–$3'$ direction

for encoding and storing the heredity information. They allow for the transmission of heredity information from one generation to another. Furthermore, there is an amazing connection between nucleic acids and proteins; nucleic acids serve as blueprints for the construction of proteins in the organism. Before we take a closer look at this process called protein biosynthesis in Section 2.3, we first study the structure of nucleic acids.

Similarly to proteins, which consist of a chain of amino acids, nucleic acids are constructed as a chain of smaller entities, the nucleotides. A *nucleotide* consists of a sugar, a phosphate, and a base (see Figure 2.3 (a)). The carbon atoms of the sugar are numbered from $1'$ to $5'$ (see Figure 2.3 (b)). By linking several nucleotides, we obtain a chain, whose backbone consists of sugar and phosphate molecules linked to each other by the $5'$ and $3'$ carbon of the sugar. For this reason one can naturally assign a reading direction to such a chain from the free $5'$ end to the free $3'$ end (see Figure 2.4).

Like the side chains in the case of amino acids, the bases are the characteristic distinctive features for nucleotides. Therefore, we can also write down a nucleic acid as a sequence of bases along a polynucleotide chain.

Essentially, there are two different types of nucleic acids, namely *deoxyribonucleic acid*, the famous DNA genes consist of, and *ribonucleic acid*, or RNA for short. The prefixes *deoxyribo* and *ribo* refer respectively to the sugars deoxyribose and ribose, constituting one of the differences between DNA and RNA. Moreover, DNA usually includes the four bases *adenine* `A`, *cytosine*

C, *guanine* G, and *thymine* T, while in RNA thymine is replaced by the base *uracil* U.

With regard to the structural levels, there are crucial differences between DNA and RNA, as we will see. A DNA molecule typically consists of a double strand of two nucleotide chains. Bases along these two strands will pair with each other with hydrogen bonds according to the patterns adenine-thymine and cytosine-guanine. These pairs of bases are said to be *complementary*, or Watson–Crick complementary; also the term *Watson–Crick pair* is used.[2] As the bases of the strands are complementary to each other, the sequence of bases on one strand can be directly inferred from the sequence on the other strand. However, one has to be aware that the reading directions of the two strands are opposite. We will call the docking of complementary bases as well as of complementary parts of strands *hybridization* in the following. Indeed, the DNA typically occurs double stranded in nature. The resulting rope ladder is twisted around an imaginary central axis to form the famous DNA double helix (see Figure 2.5). In contrast, RNA usually occurs single stranded, which allows regions of the same molecule to connect to complementary ones, resulting in various different shapes. We will discuss this in more detail in Section 13.1.

Regarding nucleic acids, we distinguish three levels of structure: The *primary structure* is again the string representation of the molecule; the *secondary structure* describes the complementary bases paired with each other by means of hydrogen bonds; and finally, the *tertiary structure* refers to the actual folding of the molecule in space.

We have already seen that we can specify nucleic acids by writing down the sequence of bases in the polynucleotide chain. In the case of the double-stranded DNA we often write down the sequence of both strands beneath each other, where the reading direction is determined by the $5'$ to $3'$ orientation of the upper strand. The sequence of the lower strand can be unambiguously[3] determined by the upper one, substituting each base with its complement and subsequently reversing the reading direction. We will call the resulting string the *reverse complementary string*, or the *reverse complement* for short.

Example 2.1. Let $s = \texttt{AGACGT}$ be the string representation of a DNA strand; then $\bar{s} = \texttt{ACGTCT}$ is the reverse complement of s, as one can see from the following view on the DNA double strand.

$$\begin{array}{ll} s: & 5' \ldots \texttt{AGACGT} \ldots 3' \\ \bar{s}: & 3' \ldots \texttt{TCTGCA} \ldots 5' \end{array}$$

According to this, it is not essential to distinguish between single- and double-stranded DNA when referring to it in terms of strings.

---

[2] Referring to James Watson and Francis Crick, who discovered the "secret of life," the structure of DNA, in 1953.

[3] At least for our considerations; in general, a rare number of "mismatches" may also occur.

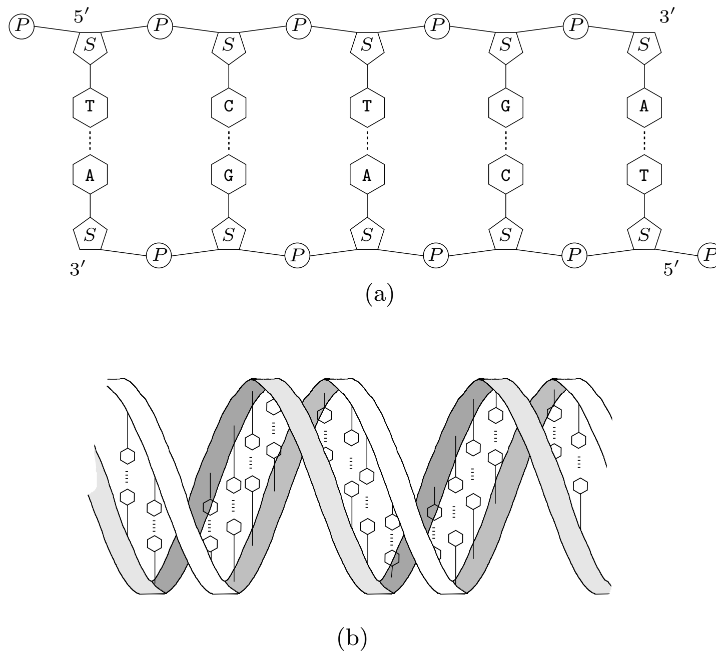


Fig. 2.5. Schematic view of a DNA double strand (a) and a DNA double helix (b)

One possible measure of length for nucleic acids, which we will use exclusively in the following, is to count the number of bases or base pairs occurring in the molecule, or at an even higher level of abstraction, the length of the corresponding string. Since bases typically occur pairwise in a DNA double strand, the length is measured using the unit *bp* (standing for *base pair*). To easily denote larger lengths, we will also use the units *kbp* and *mbp* (standing for *kilo base pair* and *mega base pair*, respectively), where 1 kbp = 1 000 bp and 1 mbp = 1 000 kbp. We will use this units also for RNA.

2.3 Hereditary Information and Protein Biosynthesis

In this section, we will sketch the central process in biology, the aforementioned protein biosynthesis, i.e., the transformation of information encoded in DNA into sequences of polypeptides. But before starting, we first describe some necessary terms in the context of the hereditary information. Here, we restrict ourselves to higher organisms whose cells contain a real nucleus. These organisms are called eucaryotes.

The amino acid sequence of proteins is encoded in the DNA; in particular, we call a region of DNA encoding for a polypeptide a *gene*. A DNA molecule containing several genes, as occurring in nature, is called a *chromosome*. In the

nucleus, chromosomes often appear in pairs, where we will call corresponding chromosomes *homologous chromosomes*. One of these is maternal and the other is paternal; for example, human cells contain 46 chromosomes, 23 from each mother and father. Generally, one also uses the term homology to denote the correlation of related molecules of different organisms. The collection of hereditary information in a cell is called its *genome*.

The DNA mainly occurs in the nucleus[4] and is not able to leave it. On the other hand, the "manufacturing halls" of proteins, the *ribosomes*, are located outside the nucleus. The process of transforming genetic information encoded in a sequence of nucleotides into a sequence of amino acids mainly consists of two steps, as shown schematically in Figure 2.6.

In the first step, the required information is copied from the corresponding gene and the copy leaves the nucleus. This is the so-called *transcription* step. Here, the DNA helix is untwisted and its two strands are separated from each other in the corresponding gene region. Then, catalyzed by an enzyme called RNA polymerase, a complementary RNA copy of one particular of the two strands (the so-called template strand) is synthesized. Not all parts of the gene are used to encode the protein; there also exist "nonrelevant" areas. The region of a gene is therefore divided into so-called exons and introns, where only exons encode for the amino acid sequence and are thus considered exclusively for the final copy of the DNA. The introns, on the other hand, are removed from a preliminary version of the copy by a process called splicing. Hence, the result of the transcription process is an RNA strand, consisting solely of coding regions of the considered gene. Accordingly, this final copy is denoted as *messenger RNA*, or mRNA for short. Next, this mRNA leaves the nucleus and passes the information of the DNA on to the ribosomes.

In the ribosomes, the information stored in the mRNA is transformed into an amino acid sequence. This process is called *translation*. Here, nonoverlapping triples of bases, called *codons*, along the mRNA strand encode for particular amino acids. The mRNA is shifted through the ribosome codon by codon, and at each step at another site of the ribosome the corresponding amino acid is synthesized to a chain of amino acids at the use of another auxiliary RNA molecule, the so-called *transfer RNA*, or tRNA for short, that carries amino acids and a corresponding triple of bases, called *anticodon*, which is complementary to the current codon of the mRNA. This assignment is shown in Table 2.2. Note that there exist $4^3 = 64$ different codons, but only 20 different amino acids are used in protein syntheses. Thus, there is a natural redundancy, i.e., some amino acids are coded by several codons. Furthermore, the termination of the synthesis is encoded in some of the codons; these are therefore called STOP codons.

A tRNA with the corresponding anticodon binds to the mRNA, and releases its amino acid, which joins the so far constructed chain of amino acids.

[4] Furthermore, certain cellular organelles, the mitochondria and chloroplasts, contain DNA for coding information too.

first position	second position				third position
	G	A	C	U	
G	Gly	Glu	Ala	Val	G
	Gly	Glu	Ala	Val	A
	Gly	Asp	Ala	Val	C
	Gly	Asp	Ala	Val	U
A	Arg	Lys	Thr	Met	G
	Arg	Lys	Thr	Ile	A
	Ser	Asn	Thr	Ile	C
	Ser	Asn	Thr	Ile	U
C	Arg	Gln	Pro	Leu	G
	Arg	Gln	Pro	Leu	A
	Arg	His	Pro	Leu	C
	Arg	His	Pro	Leu	U
U	Trp	STOP	Ser	Leu	G
	STOP	STOP	Ser	Leu	A
	Cys	Tyr	Ser	Phe	C
	Cys	Tyr	Ser	Phe	U

Table 2.2. Relation of codons to amino acids and STOP signals

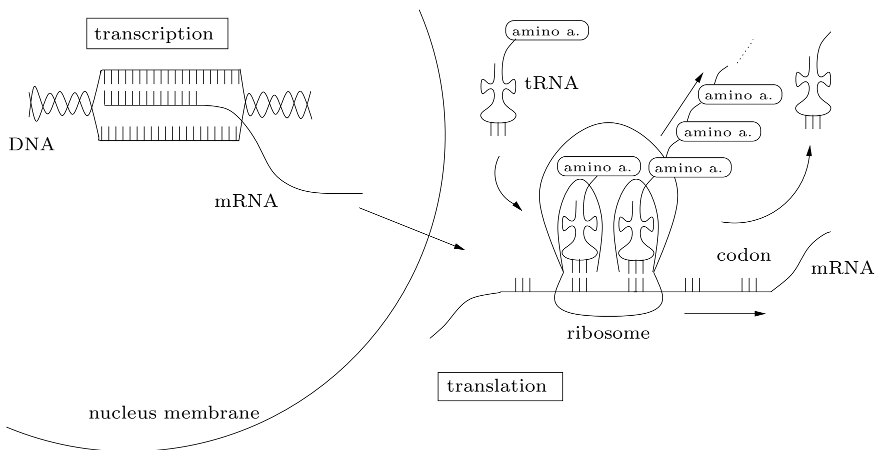


Fig. 2.6. Schematic description of transcription and translation

This process is terminated by a tRNA carrying a STOP signal, the unloaded tRNA molecules can again attach to their specific amino acids, several of which are floating around in the cell's cytoplasm.

2.4 Experimental Techniques

In this section, we will further study relevant basics of molecular biology and some experimental techniques for the examination of nucleic acids. However, we cannot deal with this subject exhaustively; instead, we will only present a small sample of the known methods, partially in a simplified way.

2.4.1 Basic Terms and Methods

Let us consider a double-stranded DNA molecule. There exists the possibility to separate the two strands by heating, a process called *denaturation*. On the other hand, we call the association of complementary bases or even complementary strands *hybridization*.

Furthermore, there exist several ways to cut the sugar-phosphate backbone of double-stranded DNA, such that smaller double strands, and not single strands, as in the case of denaturation, arise again. Such a "vertical" cut can be enabled, for example, by the use of vibration or sonic waves. The cutting of a DNA double strand at a specific site can be performed using so-called *restriction enzymes*. Each restriction enzyme detects a specific area in the DNA sequence, denoted as *restriction site*, and cuts the DNA at this site or near to it. We postpone a more detailed consideration of restriction enzymes and their use for certain tasks to Section 7.1. The enzyme *ligase* enables the reunification of DNA double strands that are cut in this way, i.e., it catalyzes the linkage of the backbones of the nucleotide chains.

2.4.2 Duplication of DNA

To perform experiments, often a single DNA molecule is not sufficient, but a high number of identical copies of the molecule is required. We will next describe two methods to generate copies of DNA molecules.

Routinely, a technique called *cloning* is used. Here, the fragment of DNA which one wants to copy, called *insert* in this context, is embedded into the genome of a host organism (*host* for short), where it is duplicated during the natural reproduction of the host by cell division. After that, the required DNA fragment is extracted from the host's genome again. This process is sketched in Figure 2.7.

There are different types of organisms that may serve as hosts. The choice depends on the length of the DNA fragment that has to be copied. The range of possible lengths varies between 15 and 50 kbp for certain types of bacteria and viruses, and up to several million base pairs for the case of artificial chromosomes that are additionally inserted into the host.

Due to this procedure, it is not at all surprising that again and again contaminations with the corresponding host DNA or changes by means of errors in the replication process of the host are encountered in the duplicated DNA; these have to be carefully considered in further examinations.

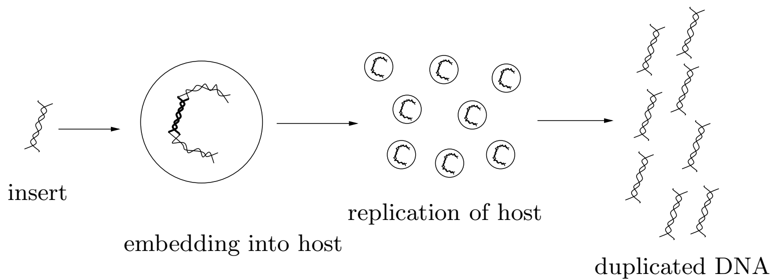


Fig. 2.7. Duplication of DNA by cloning

Moreover, replication of DNA by means of cloning can lead to the loss of complete fragments, since the insertion may have a lethal effect on the host organism and is thus not duplicated at all.

Another complication may result from cloning several fragments from one original sequence. Two fragments, that are nonconsecutive in the sequence, link to each other during the cloning process. In this case, we gain a so-called *chimeric clone*, a duplicated fragment that seems to indicate a consecutive fragment that does not exist in reality. This implies particular complications in the context of DNA sequencing.

Another method for the duplication of DNA was developed by Mullis in 1983 and performs an iterative doubling of DNA molecules in vitro.[5] For this method, we have to know the sequence of bases at the beginning as well as at the end of the molecule. This enables us to construct short single strands of DNA that are complementary to these parts, and that are denoted as *primers* . The method, known as *polymerase chain reaction*, or PCR for short, works as follows.

This procedure is depicted in Figure 2.8. Here, we denote the DNA we intend to duplicate by d, its single strands by d_1 and d_2, and the primers by p_1 and p_2. Note that, by n iterations of this method, 2^n copies of the original DNA may be produced. Besides the knowledge necessary of the primer sequence, another disadvantage of this method is the error rate. The occurrence of an error in an early iteration may obviously replicate exponentially by this method. Nevertheless, one should remember that also cloning, like nearly all biological techniques, is error prone.

2.4.3 Gel Electrophoresis and Direct Sequencing

This section is devoted to a method that enables the separation of DNA molecules according to their size. The corresponding procedure is called *gel electrophoresis* and is based on the following idea. Due to the negative charge

[5] For this technique Kary B. Mullis was awarded the Nobel price for chemistry in 1993.

Method 2.1 Polymerase chain reaction (PCR)

Given: A DNA which has to be duplicated.

Step 1: Add into a test tube:

- The given DNA,
- primers p_1 and p_2 that are complementary to the beginning and the end of the DNA,
- all nucleotides in sufficiently large quantity, and
- DNA polymerase, i.e., an enzyme that successively enables the extension of a primer by single nucleotides according to a template.

Step 2: The following steps are repeated arbitrarily often:

1. Denature the DNA by heating.
2. Let the DNA cool down. During this process, primers will hybridize with the DNA single strands that were dissolved by denaturation.
3. The primers will be elongated to complete complementary strands by means of DNA polymerase.

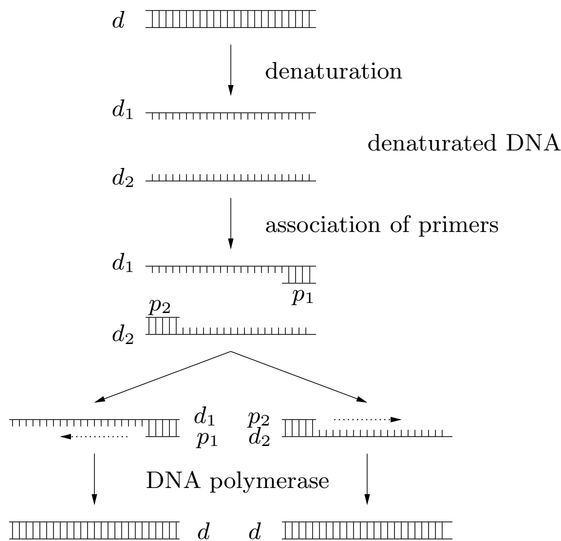


Fig. 2.8. One iteration of Step 2 of the polymerase chain reaction

of DNA molecules, they move from the negative to the positive pole in an electric field. If one applies such an electric field to a gel carrier plate on which DNA is laid out, the migration speed of the molecules is inversely proportional to their size. A smaller molecule will move faster through the gel structure, while a larger one will be more restrained by the gel in its movement. This idea is now utilized for our purpose. If a mixture of DNA molecules with variable length is given on a gel next to the negative pole and the electric field is switched on, the molecules start migrating and hence separate according

to their length with time. To separate molecules of different orders of size, one may use different types of gels. Using reference molecules whose size is known in advance, this procedure may not only serve to separate the molecules according to their size, but also to determine their actual size.

The separation process described above can also be utilized for determining the sequence of bases in the DNA, i.e., the DNA sequence. This approach is known as *chain termination method*.

Method 2.2 Chain termination method for sequencing DNA

Given: Four test tubes, labeled A, C, G, and T.

Step 1: Duplicate the DNA single strand that should be sequenced.

Step 2: Distribute the copies equally among the four test tubes A, C, G and T.

Step 3: Add to each test tube labeled $I \in \{A, C, G, T\}$ all nucleotides that do *not* contain base `I`, i.e., to test tube A add only nucleotides containing cytosine `C`, guanine `G`, and thymine `T`; analogously for test tubes C, G, and T.

Step 4: Add to each test tube $I \in \{A, C, G, T\}$ in a certain ratio:

- nucleotides that contain base `I`, and
- nucleotides that contain a chemically modified base `I` that does not allow for the elongation of a DNA strand by means of DNA polymerase, i.e., after adding such a nucleotide, the construction of the strand stops. (The modified nucleotides are called dideoxynucleotides.)

Step 5: Add to each test tube A, C, G, and T primers of the DNA one wants to sequence and DNA polymerase, and, with this, start the construction of the complementary single strand. Since test tube $I \in \{A, C, G, T\}$ also contains such nucleotides that terminate the construction of the strand (ending with `I`), it finally contains all DNA single strands ending with `I` with high probability.

Step 6: Lay out the content of the test tubes A, C, G, and T next to each other on a gel and start gel electrophoresis. The bands along the paths for the test tubes A, C, G, and T provide us the lengths after which the construction of the single strand is stopped by means of introducing the chemically modified nucleotide. This enables us to actually read the DNA sequence, which is therefore called a *read* in this context.

Step 5 of this process is illustrated in Figure 2.9. The maximal length of DNA molecules that can be sequenced using this method is nevertheless strongly restricted. Currently it is possible to sequence molecules up to a length of about 1 000 bp; afterwards, the errors due to the insufficient resolution of the gel get out of hand, such that the result is not sufficiently reliable.

With this experimental technique of direct sequencing, different types of errors may occur. The read sequence can be erroneous due to inaccurate reading of the experimental outcome, e.g., the bands indicating the spots of migrated DNA may become fuzzy, indistinguishable, or may even miss completely. These *sequencing errors* are also often denoted as base call errors, and one distinguishes essentially three types.

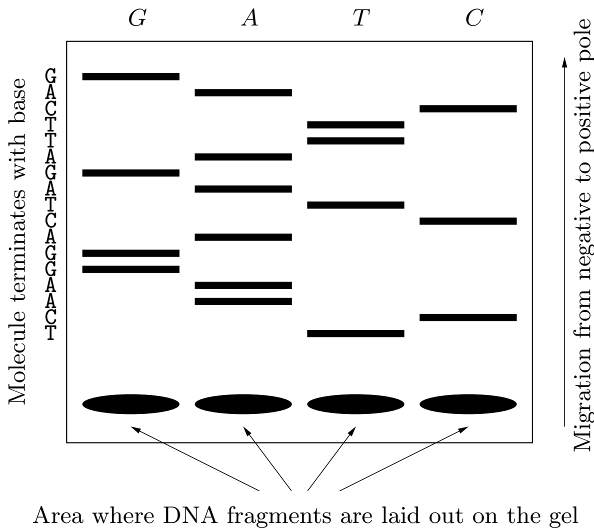


Fig. 2.9. Sequencing a DNA molecule with base sequence `GACTTAGATCAGGAACT` using the chain termination method. The shortest fragment, namely the one which only consists of the nucleotide with base `G`, moves fastest, followed by the second largest fragment with base sequence `GA`, and so on

- *Insertion:* A base appears at a position within the read where it is actually not present.
- *Deletion:* A base is missing in the read, while it occurs in the actual sequence.
- *Substitution:* A base in the actual sequence is substituted by another one in the read.

2.4.4 DNA Chips

Another useful technique enables us to determine whether a certain base sequence appears as a part of a nucleotide sequence or not. For this we use hybridization experiments. Let us denote our nucleotide sequence by s and the base sequence that we want to detect within s by t. We now synthesize the complementary sequence t' from t and test whether s and t' hybridize with each other. If this is the case, s contains the base sequence t; otherwise, it does not.

To perform several of these hybridization experiments in parallel, one uses *DNA chips*, also called *DNA arrays*. Let us assume that we would like to test for the base sequences $t_1, \ldots, t_n$ within a nucleotide sequence s. We do not have to know the actual sequence of s. Now, one "anchors" the complementary single-stranded sequences $t'_1, \ldots, t'_n$ of $t_1, \ldots, t_n$ at specific positions on a surface (to be more precise, lets them "grow" there). The sequences that we

want to test for are usually short nucleotide sequences, called *probes* in this context. The layout of the probes on the surface is known, i.e., it is known which probe corresponds to which position. After constructing this device, which is actually the DNA chip, we add single stranded, labeled (for instance, fluorescing) copies of the nucleotide sequence s. If the probes occur as complementary sequences in s, the copies of s will hybridize, and will thus be attached to the probe's position on the DNA chip. Then, we wash away the copies of s not hybridized with the probes and detect the positions where hybridizations occurred using the labeling. Finally, we can infer which sequences are contained in s and which are not. We summarize this procedure as Method 2.3.

Method 2.3 Detection of subsequences using a DNA chip

Step 1: Anchor probes at certain positions of a surface and obtain in this way a DNA chip.

Step 2: Add labeled copies of the nucleotide sequence we want to scan. Copies will hybridize with complementary probes.

Step 3: Reject non-hybridized copies of the nucleotide sequence.

Step 4: Detect, by means of the labeling, the spots on the DNA chip where hybridizations have occurred and thus obtain the set of subsequences.

Using such hybridization experiments, several errors may occur. We categorize these errors into two classes. We call an error *false positive*, if the DNA chip signals a hybridization of nucleotide sequence s with some probe t_i', although the base sequence t_i does not occur inside s, and therefore no hybridization should have taken place. On the other hand, we call an error *false negative* if the DNA chip does not signal a hybridization, while this should happen, since the corresponding sequence is contained in s.

Finally, we would like to note that although we called the devices DNA chips, the approach described above can in principle also be applied for other (single-stranded) nucleic acids, e.g., for RNA.

2.5 Bibliographic Notes

Please note again that, in this chapter, facts have been presented in a simplified way. For further information we refer the reader to the following literature.

A substantial part of the presented facts is also discussed in the corresponding chapters of the books by Setubal and Meidanis [180], Clote and Backofen [49], and Waterman [201]. Further insights into molecular biology can be found in a multitude of books on this subject, e.g., Campbell and Reece [39] and Lewin [133].

The structure of DNA was investigated by Watson and Crick in 1953 [205], who were together with Wilkins awarded the Nobel Prize for Medicine in 1962

for their discovery. A report worth reading on the discovery of the DNA double helix was authored by Watson [203]. We also recommend another book by Watson [204] for insights into the development and history of genetics during the last few decades. For the development of the PCR method, Mullis [171] was awarded the Nobel Prize for Chemistry in 1993. The chain termination method for DNA sequencing goes back to Sanger, Nicklen, and Coulson [172] and is also known as Sanger method.

3

Basic Concepts: Strings, Graphs, and Algorithms

In this chapter we will give a very brief overview of the basic definitions and concepts for dealing with strings, graphs, and algorithms needed throughout this book. The goal of this chapter is mainly to remind the reader of the most important concepts and to establish our notations. In Section 3.4 we have listed references to some introductory textbooks where the topics sketched in this chapter are presented in greater detail.

3.1 Strings

Many biological structures can be naturally represented by strings, and we will frequently use this representation throughout the book. In this section, we will formally define strings, present some useful notations, and illustrate some simple properties of strings.

Definition 3.1. *An* alphabet Σ *is a finite nonempty set. The elements of an alphabet are called* characters , letters , *or* symbols . *A* string s *over an alphabet* Σ *is a (finite) concatenation of symbols from* Σ. *The* length *of a string* s *is the number of symbols in* s, *it is denoted by* $|s|$. *For each symbol* x, $|s|_x$ *denotes the number of occurrences of the symbol* x *in* s. *This implies*

$$|s| = \sum_{x \in \Sigma} |s|_x \ .$$

The empty string λ *denotes the string of length* 0.

We denote by Σ^n *the set of all strings of length* n *over the alphabet* Σ, *and by* $\Sigma^* = \bigcup_{i \geqslant 0} \Sigma^i$ *the set of all strings over* Σ.

In the existing bioinformatics literature, the notions "string" and "sequence" are often used synonymously. In this book, we will usually use the term "sequence" for biological objects like DNA sequences, while in the context of our abstract models we will only use the term "string."

Definition 3.2. *Let s and t be strings over an alphabet Σ, then $s \cdot t$, or st for short, denotes the* concatenation *of s and t.*

Definition 3.3. *Let s and t be strings over an alphabet Σ.*

- *s is a* substring *of t, if there exist strings u and v such that $t = usv$.*
- *s is a* prefix *of t, if there exists a string v such that $t = sv$.*
- *s is a* suffix *of t, if there exists a string u such that $t = us$.*
- *s is called a* proper *substring [prefix, suffix] of t, if s is a substring [prefix, suffix] of t, and $s \neq t$.*
- *s is a* subsequence *of t, if all symbols of s appear in the same order (but not necessarily contiguously) also in t.*

The empty string λ is considered a prefix (suffix) of any string. In this context, λ is also called the *empty prefix* (*empty suffix*).

Please note that in contrast to the terms "string" and "sequence," which may be used synonymously, the terms "substring" and "subsequence" have different meanings.

The following example illustrates the above definitions.

Example 3.1.

- Let $\Sigma = \{a, b, c\}$. Then $abcabc$ is a string over Σ, $|abcabc| = 6$, and $|abcabc|_a = |abcabc|_b = |abcabc|_c = 2$.
- Let abc and cba be two strings over Σ. Then $abc \cdot cba = abccba$ is the concatenation of abc and cba; furthermore, $\lambda \cdot abc = abc$ and $cba \cdot \lambda = cba$.
- The string abc is a substring of abc, but not a proper substring of abc; the string ca is a proper substring of $abcabc$.
- The string ab is a proper prefix of abc; the string c is a proper suffix of abc.
- The string aa is a subsequence of $abca$, but the string acb is not a subsequence of $abca$. $\Diamond$

We will now define a notation for dealing with substrings.

Definition 3.4. *Let Σ be an alphabet, let $s = s_1 \dots s_n$, for $s_1, \dots, s_n \in \Sigma$, be a string. For all $i, j \in \{1, \dots, n\}$, $i < j$, we denote the substring $s_i \dots s_j$ by $s[i, j]$. Furthermore we denote the i-th symbol s_i of s also by $s[i]$.*

Next, we will consider the overlapping of strings.

Definition 3.5. *Let s and t be strings over an alphabet Σ. If there exists a partition of s and t with the properties*

(i) $s = xy$,
(ii) $t = yz$,
(iii) $x \neq \lambda$ *and* $z \neq \lambda$, *and*
(iv) $|y|$ *is maximal with (i), (ii), and (iii),*

then y is called the overlap *of s and t, denoted by $Ov(s,t)$, and the string xyz is called the* merge *of s and t, denoted by $\langle s,t \rangle$. We denote the string x by $Pref(s,t)$ and the string z by $Suff(s,t)$. By $ov(s,t)$ [$pref(s,t), suff(s,t)$] we denote the length of $Ov(s,t)$ [$Pref(s,t), Suff(s,t)$].*

If $ov(s,t) = 0$ for two strings s and t, then $Ov(s,t)$ is also called the *empty overlap* of s and t; in this case the merge of s and t equals their concatenation.

We now illustrate this definition by an example.

Example 3.2. Let $s = cccababab$ and $t = abababddd$ be two strings over the alphabet $\Sigma = \{a, b, c, d\}$. Then $\langle s,t \rangle = cccabababddd$ is the merge of s and t; the overlap of s and t is $Ov(s,t) = ababab$. Furthermore, $Pref(s,t) = ccc$ and $Suff(x,t) = ddd$. ◊

Please note that Definition 3.5 requires, for any two arbitrary strings s and t, that $Pref(s,t) \neq \lambda$ and $Suff(s,t) \neq \lambda$. It therefore can also be interesting to consider the overlap of a string with itself. For example, for the string $s = ababab$, we have $Ov(s,s) = abab$.

For any two strings s and t, we know $s = Pref(s,t)Ov(s,t)$ and $t = Ov(s,t)Suff(s,t)$.

We close this section with the definition of string homomorphisms.

Definition 3.6. *Let Σ_1 and Σ_2 be two alphabets. A function $h : \Sigma_1^* \to \Sigma_2^*$ is called a* string homomorphism *(or* homomorphism *for short), if all strings $x, y \in \Sigma_1^*$ satisfy*

$$h(x \cdot y) = h(x) \cdot h(y). \tag{3.1}$$

Please note that due to Equation (3.1) a homomorphism h is uniquely determined by the definition of its values $h(a)$ for all $a \in \Sigma_1$, and that in particular $h(\lambda) = \lambda$ holds.

3.2 Graphs

The most important data structures for many algorithms in bioinformatics besides strings are graphs, and especially trees. In this section, we will therefore present the most important notations from the area of graph theory. For a more detailed introduction to graph theory we refer the reader to the references given in Section 3.4.

Definition 3.7. *An* (undirected) graph *G is a pair $G = (V, E)$, where V is a finite set of* vertices *and $E \subseteq \{\{x,y\} \mid x, y \in V \text{ and } x \neq y\}$ is a set of* edges.

Two vertices $x, y \in V$ are called adjacent *if $\{x,y\} \in E$. A vertex $x \in V$ is called* incident *to an edge $e \in E$, if $x \in e$.*

The degree *of a vertex x is the number of edges incident to x; we denote it by $\deg(x)$. The degree of a graph G is defined as the maximum degree over all vertices in V.*

A graph $G = (V, E)$ is called complete *if $E = \{\{x,y\} \mid x, y \in V \text{ and } x \neq y\}$ holds, i.e., if there exists an edge between any pair of distinct vertices.*

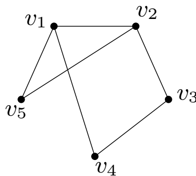


Fig. 3.1. The graph given in Example 3.3

We often visualize a graph by drawing the vertices as dots and the edges as lines between them, as shown in the following example.

Example 3.3. Let $G = (V, E)$ be a graph with $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $E = \{\{v_1, v_2\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_5\}, \{v_3, v_4\}\}$. Then G can be visualized as shown in Figure 3.1. ♢

Now we define the notion of a subgraph of a given graph.

Definition 3.8. *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs. We say that G_2 is a* subgraph *of G_1, if $V_2 \subseteq V_1$ and $E_2 \subseteq E_1$.*

Let $G = (V, E)$ be a graph, let V' be a subset of V. The subgraph of G induced by V' *is the graph $G' = (V', E')$, where $E' = \{\{x, y\} \in E \mid x, y \in V'\}$.*

We next give a definition that allows us to consider sequences of vertices in a graph.

Definition 3.9. *Let $G = (V, E)$ be a graph. A* path *in G is a sequence of vertices $P = x_1, x_2, \ldots, x_m$, $x_i \in V$ for all $i \in \{1, \ldots, m\}$ such that $\{x_i, x_{i+1}\} \in E$ holds for all $i \in \{1, \ldots, m-1\}$. The* length *of the path P is the number $m - 1$ of edges $\{x_i, x_{i+1}\}$ between consecutive vertices in P. A path $P = x_1, x_2, \ldots, x_m$ is called* simple *if either all vertices $x_1, x_2, \ldots, x_m$ of the path are pairwise different or if $x_1, x_2, \ldots, x_{m-1}$ are pairwise different and $x_1 = x_m$ holds.*

A path $P = x_1, x_2, \ldots, x_m$ with $x_1 = x_m$ is called a cycle*; a* simple cycle *is a simple path of length $\geqslant 3$ that is also a cycle. A simple cycle in G containing all vertices from V is called a* Hamiltonian cycle *of G.*

A path (cycle) that uses every edge of E exactly once is called an Eulerian path *(*Eulerian cycle*.)*

An edge between two nonconsecutive vertices on a cycle is called a chord *of the cycle.*

The graph G is said to be connected *if, for all $x, y \in V$, there exists a path between x and y.*

A graph that does not contain any cycle of odd length is called bipartite.

In the following, we typically consider only simple paths and simple cycles, and we call them “paths” and “cycles” respectively for short. If we want to consider a non-simple path or cycle, we mention this explicitly.

Next, we give the formal definition of a tree, which is an important special case of a graph.

Definition 3.10. *Let $T = (V, E)$ be a graph. The graph T is a* tree *, if it is connected and does not contain any simple cycle. The vertices of degree* 1 *in a tree are called* leaves*; the vertices of degree $\geqslant 2$ are called* inner vertices*.*

A tree may have a specially marked vertex that is called the root*. In this case we call the tree a* rooted tree*.*

The following property of trees will be useful in the sequel. For the (simple) proof we refer the reader to the literature referenced in Section 3.4.

Lemma 3.1. *Let $T = (V, E)$ be a tree. Then there exists exactly one simple path between x and y for any two vertices $x, y \in V$.* □

Definition 3.11. *Let $G = (V, E)$ be a graph, let $T = (V', E')$ be a tree. If $V = V'$ and $E' \subseteq E$, then T is called a* spanning tree *of G.*

If for each edge in a graph a direction is given, one gets a directed graph that can be formally defined as follows.

Definition 3.12. *A* directed graph *is a pair $G = (V, E)$, where V is a finite set of vertices and $E \subseteq \{(x, y) \mid x, y \in V \text{ and } x \neq y\}$ is a set of* directed edges*.*

We also say that the edge (x, y) starts *in vertex x and* ends *in vertex y.*

The indegree *of a vertex x, denoted by* $\text{indeg}(x)$*, is the number of edges ending in x; the* outdegree *of x, denoted by* $\text{outdeg}(x)$*, is the number of edges starting in x.*

A (simple) directed path *in a directed graph $G = (V, E)$ is a sequence $P = x_1, \ldots, x_m$ of vertices such that $(x_i, x_{i+1}) \in E$ holds for all $i \in \{1, \ldots, m-1\}$, where either $x_1, \ldots, x_m$ are pairwise distinct, or $x_1, \ldots, x_{m-1}$ are pairwise distinct and $x_1 = x_m$.*

In analogy to Definition 3.9, one can define Hamiltonian paths and cycles, as well as Eulerian paths and cycles, also for directed graphs.

We note that a rooted tree unambiguously induces a directed graph, where all edges are oriented from the root in the direction of the leaves. Therefore, we speak of a *directed tree* in this context. This means that in a directed tree the root has an indegree of 0, all other vertices have indegree 1, and the leaves are exactly the vertices of outdegree 0. If there exists an edge (x, y) in a directed tree, we call x the *parent* of y and we call y the *child* of x.

If furthermore a linear order is defined on the children of every vertex, we call this tree an *ordered tree*.

We will now define a special type of tree, where the degree of the vertices is bounded.

Definition 3.13. *A* directed binary tree *is a directed tree, where each inner vertex has outdegree* 2*.*

An undirected unrooted binary tree *is an (undirected) tree, where each inner vertex has degree* 3*.*

Often one wants to add some inscriptions to the vertices or edges of a graph. In this case, we speak of *edge-labeled* or *vertex-labeled* graphs. The most frequently used form of labeling is the assignment of some costs or weights to each edge via some cost function $\delta : E \to \mathbb{Q}$. A graph with such a cost function is called an *edge-weighted graph*, and we denote it by (V, E, δ).

3.3 Algorithms and Complexity

In this section we present some different types of algorithmic problems and briefly describe how to measure the complexity of algorithms. We give no formal definition of the term "algorithm" here. For our purposes it suffices to understand an algorithm as a sequence of computational steps that is implementable as a program in an arbitrary (imperative) programming language like C or Pascal. In this book, we describe the presented algorithms in a kind of pseudo programming language that allows, besides the usual constructions like while loops, for loops, if-then-else commands, and assignments, also arbitrary mathematical formulas and the use of natural language. This allows us to abstract from implementation details and to focus on clearly presenting the ideas behind the algorithms.

We can use algorithms to solve different types of problems; here, we essentially distinguish between decision problems, optimization problems, and computing problems. In the following, we specify these classes of problems in greater detail; for more formal definitions we refer the reader to the bibliographic notes in Section 3.4.

For a *decision problem*, the task is to test some condition for the given input, and to output Yes if the input satisfies the condition and No if it does not. Thus, to define a decision problem it is sufficient to specify the set of allowed inputs and the condition to be tested. An example of a decision problem is the following.

Example 3.4. The *Hamiltonian cycle problem* is the following decision problem.

Input: An undirected graph $G = (V, E)$.
Output: Yes if G contains a Hamiltonian cycle. No otherwise. ◊

Thus, in the case of a decision problem, the task is to compute a function from the set of inputs to the set $\{\text{Yes}, \text{No}\}$. This class of tasks can obviously be generalized by computing an output from an arbitrary domain for a given input, which leads to the class of *computing problems*. If the output is uniquely

determined for every input, we call the corresponding computing problem a *function computing problem*; otherwise, we call it a *relation computing problem*. The following example shows a computing problem, or more precisely, a function computing problem.

Example 3.5. The problem of computing the overlap of two strings is the following function computing problem.

Input: Two strings s and t over an alphabet Σ.
Output: The overlap $Ov(s, t)$ of s and t. ♢

In the case of an *optimization problem*, we assign to each input a set of *feasible solutions*. Then a *cost function* assigns a cost to each of these feasible solutions and the goal is to find a feasible solution with minimum or maximum cost. For the specification of an optimization problem, it is thus neccessary to specify the following four parameters: the set of input instances, the set of feasible solutions for each input instance, the cost function for the feasible solutions, and the optimization goal. More formally, we describe an optimization problem by a quadruple $\mathcal{U} = (\mathcal{I}, \mathcal{M}, cost, goal)$, where $\mathcal{I}$ is the set of input instances, $\mathcal{M}(I)$ is the set of feasible solutions for input $I \in \mathcal{I}$, *cost* is the cost function that assigns a cost value to each feasible solution (depending on the actual input instance), and *goal* is the optimization goal, i.e., either minimization or maximization.

We will now illustrate the notion of an optimization problem with the example of the Traveling Salesman Problem, which consists of finding a minimum cost Hamiltonian cycle in an edge-weighted complete graph. The name of this problem stems from the following motivation: A traveling salesman wants to visit n customers in n different cities. He knows the pairwise distances between these cities and wants to find a shortest cyclic tour starting at home, visiting each of the n cities exactly once, and eventually returning home.

Definition 3.14. *The* traveling salesman problem, *TSP for short, is the following optimization problem:*

Input: An undirected complete edge-weighted graph $G = (V, E, d)$ with n vertices and a cost function $d : E \to \mathbb{Q}^{\geqslant 0}$.
Feasible solutions: All Hamiltonian cycles in G.
Costs: The costs of a feasible solution $C = x_1, \ldots, x_n, x_1$ correspond to the sum of the edge weights on this cycle, i.e.,

$$cost(C) = \left(\sum_{i=1}^{n-1} d(\{x_i, x_{i+1}\}) \right) + d(\{x_n, x_1\}.$$

Optimization goal: Minimization.

We have just seen how to classify and describe algorithmic problems. Now we want to present some tools for the analysis of algorithms. When we have given an algorithm for some problem, we often want to determine its complexity. There are different measures for the complexity of an algorithm; the most frequently used measures are the running time and the amount of memory used. We will in the following restrict our attention to the analysis of the running time.

The running time of an algorithm A on an input x is considered to be the number of elementary computation steps the algorithm performs on it, we denote it by $Time_A(x)$. Usually, one is not only interested in the running time of an algorithm on one specific input, but one wants to get an estimate of the running time on any input, depending on the length of the input only. For this reason we define the running time for any input of length n as

$$Time_A(n) = \max\{ Time_A(x) \mid x \text{ is an input instance of length } n\}.$$

In this way we can interpret the running time of an algorithm as a function $Time_A : \mathbb{N} \to \mathbb{N}$ that determines the growth of the running time from the input length. Since the exact running time of an algorithm is hard to calculate in most cases, and since it is subject to implementation details, we will in the following restrict ourselves to an asymptotic analysis based on the so-called *Landau symbols*. Using this notation, we can simply refer to the order of the running time, which provides us with the most meaningful information in most cases.

Definition 3.15. *Let $f : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$ and $g : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$ be two arbitrary functions. Then, we define the function classes $O(f(n))$, $\Omega(f(n))$, and $\Theta(f(n))$ as follows:*

$$\begin{aligned}
g(n) \in O(f(n)) &\iff \textit{there exist } n_0 \in \mathbb{N} \textit{ and } c \in \mathbb{N}, \\
&\qquad \textit{such that } g(n) \leqslant c \cdot f(n) \textit{ for all } n \geqslant n_0, \\
g(n) \in \Omega(f(n)) &\iff \textit{there exist } n_0 \in \mathbb{N} \textit{ and } d \in \mathbb{N}, \\
&\qquad \textit{such that } g(n) \geqslant (1/d) \cdot f(n) \textit{ for all } n \geqslant n_0, \\
g(n) \in \Theta(f(n)) &\iff g(n) \in O(f(n)) \textit{ and } g(n) \in \Omega(f(n)).
\end{aligned}$$

Informally speaking, Definition 3.15 says that every function in $O(f(n))$ is asymptotically growing at most as fast as $f(n)$ and that every function in $\Omega(f(n))$ is asymptotically growing at least as fast as $f(n)$. This implies that $\Theta(f(n))$ is the class of all functions that are asymptotically growing exactly as fast as $f(n)$.

It is easy to show that every polynomial $p(n)$ of degree k lies in $\Theta(n^k)$; for example, $3n^3 + 5n^2 - n + 7 \in \Theta(n^3)$. This means that for determining the Landau symbols we can ignore constant factors and lower-order terms.

We call $f(n)$ a *polynomially bounded function* if there exists a $k \in \mathbb{N}$ such that $f(n) \in O(n^k)$.

We have seen above how to determine the complexity of a given algorithm; in the following, we want to analyze the complexity of problems. Let X be an algorithmic problem. We say that X is solvable in $O(f(n))$ time if there exists an algorithm A with $Time_A(n) \in O(f(n))$ that solves the problem X.

Experience shows that in most cases a problem is practically solvable in reasonable time if there exists a polynomial-time algorithm for it, and that getting the solution in reasonable time is in most cases impossible if there is no polynomial-time algorithm known. Thus, we define the class P as the set of all decision problems for which there exists a deterministic polynomial-time algorithm. An algorithm is called deterministic if its whole computation is uniquely fixed by the input.

The class NP is defined as the set of decision problems for which there exists a *nondeterministic* polynomial-time algorithm. For a formal introduction to the concept of nondeterminism we refer the reader to the references given in Section 3.4. Informally, NP is the class of all decision problems for which it is possible to check in polynomial time if a given solution candidate is indeed a solution.

Example 3.6. The Hamiltonian cycle problem presented in Example 3.4 above belongs to NP, since for a given sequence of vertices it is possible to check in polynomial time if, on the one hand all vertices of the input graph appear exactly once, and if, on the other hand, for any two consecutive vertices there exists an edge in the graph. ♢

From the definition of P and NP it is immediately clear that P $\subseteq$ NP holds. It is not known if this inclusion is strict, but it is a common belief that P is a proper subset of NP. There are many problems in NP for which there is no known deterministic polynomial-time algorithm. Although we do not know about a method for proving for any problem in NP that it does not belong to P, there is a method to prove for many problems that they are computationally hard in the following sense: We say that a problem X is *NP-hard* if the assumption that X would be solvable deterministically in polynomial time implies that *every* problem in NP would also belong to P. If an NP-hard problem belongs to the class NP itself, we call it *NP-complete* . Thus, the NP-complete problems are the hardest problems within the class NP. So far, there are some thousands of NP-complete problems known, and for all of them there are only algorithms with exponential running time known. In most cases, these algorithms are of little use for practical applications since the running time on inputs of realistic sizes would amount to thousands or millions of years or even more.

To prove its NP-hardness for a given decision problem X, it suffices to design an algorithm which transforms any input instance I_Y of a problem already known as NP-hard into an input instance I_X of X in polynomial time, such that I_X satisfies the condition of X if and only if I_Y satisfies the condition of Y. Such an algorithm is called a *polynomial-time reduction from*

Y to X. We will present some examples of such reductions throughout this book.

We have just seen how to prove the hardness of a decision problem via the concept of NP-completeness. Now we want to apply this concept also to optimization problems.

We note that one can canonically find a decision problem for each optimization problem by introducing a threshold on the cost of a feasible solution for the input and asking whether (in case of a minimization problem) there exists a feasible solution with costs less than the threshold. We will now formally define this *threshold problem.*

Definition 3.16. *Let $\mathcal{U} = (\mathcal{I}, \mathcal{M}, cost, goal)$ be an optimization problem. If the optimization goal is minimization (maximization), then we define the* threshold problem *for $\mathcal{U}$ to be the following decision problem:*

Input: An input instance $I \in \mathcal{I}$ and a number $t \in \mathbb{Q}^{\geqslant 0}$.
Output: YES *if there exists a feasible solution $x \in \mathcal{M}(I)$ satisfying $cost(x) \leqslant t$ ($cost(x) \geqslant t$).* NO *otherwise.*

We now call an optimization problem *NP-hard* if the corresponding threshold problem is NP-hard. In the following, we also call the threshold problem for a given optimization problem the *corresponding decision problem.*

We have now seen that there are problems for which there is little hope of finding an exact solution within reasonable time, and we have described some methods for proving such hardness for a given problem. This raises the question of how to handle such problems to find at least in some sense a reasonable suboptimal solution. There are many different approaches to answer this question; a detailed overview of these methods can be found in the references given in the last section of this chapter, and we will present some examples of these methods in subsequent chapters. We will just detail one approach in the following, suitable for optimization problems, the design of so-called *approximation algorithms.* The basic idea of this approach is to relax the requirement of finding an optimal solution for the problem instance, and to also accept feasible solutions whose costs differ from the optimal cost only by a (small) constant factor.

We call an algorithm for an optimization problem *consistent* if it outputs a feasible solution for every input instance. For all consistent algorithms, we now can define the notion of the *approximation ratio.*

Definition 3.17. *Let $\mathcal{U} = (\mathcal{I}, \mathcal{M}, cost, goal)$ be an optimization problem; let A be a consistent algorithm for $\mathcal{U}$. For each input instance $I \in \mathcal{I}$ we define the* approximation ratio of A on I *by*

$$R_A(I) = \begin{cases} \dfrac{cost(A(I))}{\mathrm{Opt}(I)} & \text{if } goal = minimization, \\[2ex] \dfrac{\mathrm{Opt}(I)}{cost(A(I))} & \text{if } goal = maximization, \end{cases}$$

Algorithm 3.1 Spanning tree algorithm for the metric TSP

Input: An undirected complete edge-weighted graph $G = (V, E, d)$ with n vertices and an edge weight function $d : E \to \mathbb{Q}^{\geqslant 0}$.

1. Compute a minimum spanning tree T of G, i.e., a spanning tree of minimum weight.
2. Choose an arbitrary vertex $v \in V$ and execute a depth-first search in T from v. Enumerate the vertices in the order in which they are visited for the first time during the depth-first search. Let $v_{i_1}, \ldots, v_{i_n}$ be this enumeration of the vertices.
3. Let $H := v_{i_1}, \ldots, v_{i_n}, v_{i_1}$.

Output: The Hamiltonian cycle H of G.

---

where $\mathrm{Opt}(I)$ *denotes the cost of an optimal solution for* I*.*

For every number $\delta > 1$ *we say that* A *is an* δ-approximation algorithm *for* $\mathcal{U}$ *if*

$$R_A(I) \leqslant \delta$$

holds for all $I \in \mathcal{I}$*.*

This means that a δ-approximation algorithm always outputs a feasible solution whose cost differs at most by a factor of δ from the cost of an optimal solution. We illustrate the definition of an approximation algorithm by the following example. We consider a restricted version of the traveling salesman problem as defined in Definition 3.14, where all edge costs obey the following condition.

Let $G = (V, E, d)$ be an undirected complete edge-weighted graph with n vertices and an edge weight function $d : E \to \mathbb{Q}^{\geqslant 0}$, given as an input for the TSP. We say that d satisfies the *triangle inequality* if, for all $x, y, z \in V$ the following inequality holds:

$$d(\{x, y\}) \leqslant d(\{x, z\}) + d(\{y, z\}). \tag{3.2}$$

Informally speaking, the triangle inequality ensures that no detour can be cheaper then the direct path between any two vertices. The restriction of the TSP to input instances obeying the triangle inequality is called the *metric TSP* or *Δ-TSP*. It is possible to show that the metric TSP is also NP-hard. We consider the so-called spanning tree algorithm for the metric TSP as detailed in Algorithm 3.1.

Before we will show that the spanning tree algorithm is an approximation algorithm for the metric TSP, we will illustrate its work with an example.

Example 3.7. We consider the work of the spanning tree algorithm on the graph with five vertices as shown in Figure 3.2 (a). The edge weights of this graph obviously obey the triangle inequality. The unique minimum spanning

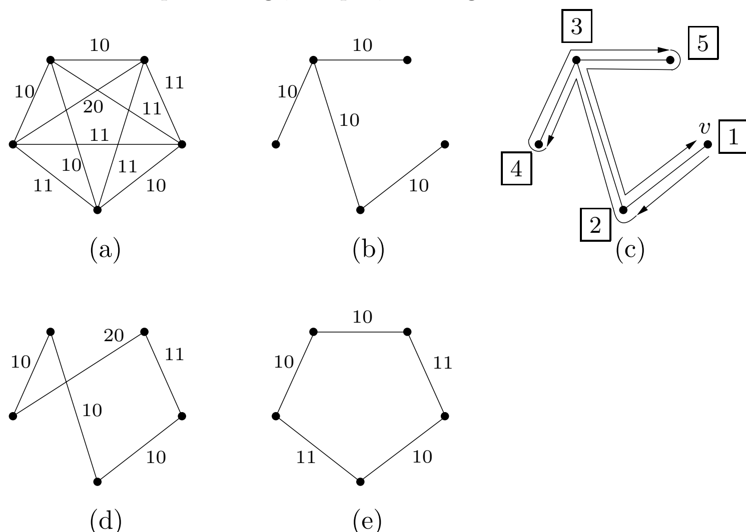


Fig. 3.2. An example for the work of Algorithm 3.1

tree is shown in Figure 3.2 (b). Figure 3.2 (c) shows the execution of a depth-first search in the spanning tree; the resulting Hamiltonian cycle is shown in Figure 3.2 (d). The cost of this Hamiltonian cycle is 61. In contrast to this, an optimal Hamiltonian cycle has a cost of only 52, as shown in Figure 3.2 (e). ◊

Now we will prove that the spanning tree algorithm in any case computes a Hamiltonian cycle that is at most twice as long as the optimal one.

Theorem 3.1. *Algorithm 3.1 is a polynomial 2-approximation algorithm for the metric TSP.*

Proof. The computation of a minimum spanning tree as well as the depth-first search[1] are possible in polynomial time; for a detailed analysis, see the references in Section 3.4. It is thus obvious that Algorithm 3.1 runs in polynomial time.

We will now analyze the approximation ratio of the algorithm on an input instance $G = (V, E, d)$. If we delete an arbitrary edge from an arbitrary Hamiltonian cycle C of G, we get a spanning tree of G. This means that every Hamiltonian cycle is as least as expensive as the minimum spanning tree. Thus, this holds also for the optimal Hamiltonian cycle H_{opt}, i.e.,

$$cost(H_{\text{opt}}) \geqslant cost(T). \tag{3.3}$$

[1] Depth-first search is an efficient method for completely traversing a given graph, for details we refer the reader to the references given in Section 3.4.

The depth-first search in T traverses each edge exactly twice; thus, the traversed (non-simple) path P in G has exactly two times the cost of the constructed minimum spanning tree. The algorithm constructs a Hamiltonian cycle $H = v_{i_1}, \ldots, v_{i_n}, v_{i_1}$ from this path by choosing the first occurrence of each vertex. Thus, in H some subpaths of P, those consisting of already visited vertices, are cut short by direct edges. Due to the triangle inequality, the direct edge from a vertex v_{i_j} to a vertex $v_{i_{j+1}}$ has at most the weight of the corresponding subpath in P; thus,

$$cost(H) \leqslant 2 \cdot cost(T). \tag{3.4}$$

From (3.3) and (3.4) we get

$$\frac{cost(H)}{cost(H_{\text{opt}})} \leqslant \frac{2 \cdot cost(T)}{cost(T)} = 2.$$

Thus, Algorithm 3.1 is a 2-approximation algorithm for the metric TSP. □

As a final remark we note that the best known approximation algorithm for the metric TSP reaches an approximation ratio of $\frac{3}{2}$.

3.4 Bibliographic Notes

A detailed introduction to the theory of strings is given in the textbooks by Gusfield [91] and Crochemore and Rytter [54]. An introduction to graph theory can be found in many good textbooks, such as those by Diestel [58], Harary [99], and Golumbic [85].

The books by Cormen et al. [51], Hromkovič [105], and Aho et al. [3, 4] give a good overview of the field of algorithmics. An introduction to complexity theory including a formal definition of the classes P and NP can be found in the books by Hromkovič [104], Hopcroft et al. [103], and Sipser [181]. The book by Garey and Johnson [79] gives an in-depth description of the theory of NP-completeness and also a list of many NP-complete problems. The theory of approximation algorithms is discussed in detail in the books by Vazirani [196], Hromkovič [105], and Ausiello et al. [18]. The above-mentioned $\frac{3}{2}$-approximation algorithm for the metric TSP goes back to Christofides [43].

4

String Algorithms

In this chapter we will present some basic algorithms and data structures for dealing with strings. This includes methods for the exact comparison of strings and for finding repetitive structures in strings. While some of the presented approaches can be used directly for problems arising in molecular biology, for example, for the search for repeats in a DNA sequence, other methods presented here will serve as subprocedures for solving more complex problems in later chapters of this book.

The chapter is organized as follows: We introduce the basic problem of string matching in Section 4.1, and we present a simple algorithm for it. Sections 4.2 and 4.3 contain improved algorithmic approaches to the string matching problem. Section 4.4 is devoted to the presentation of an important data structure for handling strings, called suffix tree. We will consider in Section 4.5 some variants of the string matching problem that can be solved efficiently using suffix trees. Subsection 4.5.4 deals with another basic problem: the search for identical substrings in a given string. Finally, Section 4.6 introduces another powerful tool for dealing with strings, the suffix array. The chapter concludes with a summary in Section 4.7 and some bibliographic notes in Section 4.8.

4.1 The String Matching Problem

The most elementary problem when dealing with strings is probably the string matching problem, which consists of finding a (usually short) string, the *pattern* , as a substring in a given (usually very long) string, the *text* . This problem arises in many different applications, also outside of molecular biology, for example, in text editors or search engines for the Internet. An important application in molecular biology is the search for a newly sequenced DNA fragment, possibly coding for a gene, in a genome database. Although one usually prefers to allow a certain error rate for this search, an algorithm for

Algorithm 4.1 Naive string matching algorithm

Input: A pattern $p = p_1 \dots p_m$ and a text $t = t_1 \dots t_n$.

```
I := ∅
for j := 0 to n − m do
  i := 1
  while p_i = t_{j+i} and i ⩽ m do
    i := i + 1
  if i = m + 1 then {p_1 ... p_m = t_{j+1} ... t_{j+m}}
    I := I ∪ {j + 1}
```

Output: The set I of positions, where an occurrence of p as a substring in t starts.

the exact string matching problem can be used as a subroutine, as we will see in the context of the FASTA method in Subsection 5.2.1.

First, we give a formal definition of the string matching problem.

Definition 4.1. *Let Σ be an arbitrary alphabet. The (exact)* string matching problem *is the following computing problem:*

Input: Two strings $t = t_1 \dots t_n$ and $p = p_1 \dots p_m$ over Σ.

Output: The set of all positions in the text t, where an occurence of the pattern p as a substring starts, i.e., a set $I \subseteq \{1, \dots, n - m + 1\}$ of indices, such that $i \in I$ if and only if $t_i \dots t_{i+m-1} = p$.

The first naive approach for solving the string matching problem consists of sliding a window with the pattern p over the text t, and testing for each position of the window whether this substring of the text coincides with the pattern. More formally speaking, the algorithm tests for each position $i \in \{1, \dots, n - m\}$ if the condition $t_i \dots t_{i+m} = p$ holds (see Algorithm 4.1).

Figure 4.1 shows an example of the work of the naive string matching algorithm. In the worst case, this algorithm needs m comparisons for each i, which sums up to an overall running time in $O(m \cdot (n - m))$. The strings $t = a^n$ and $p = a^m$ make up a worst-case example for this algorithm. Even if we consider a modification of the algorithm that outputs only the first position j where $t_j \dots t_{j+m-1} = p$ holds, the naive string matching algorithm has a running time in $O(m \cdot (n - m))$, as the input instance $t = a^{n-1}b$ and $p = a^{m-1}b$ shows.

Our goal in the following is to find more efficient methods for solving the string matching problem. We will reach this goal by exploiting the structure of the pattern or of the text in order to save some comparisons. This general idea is illustrated by the following example.

Example 4.1. Let $t = ababb$ and $p = abb$. When comparing p with $t_1t_2t_3$, we recognize that $p_1 = t_1$ and $p_2 = t_2$, but $p_3 \neq t_3$. Since we know that $p_1 \neq p_2$, $p_2 = t_2$ implies that shifting the pattern by one position cannot be successful. ♢

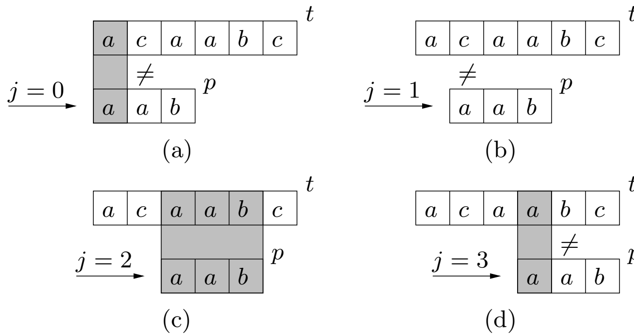


Fig. 4.1. An example for the work of the naive string matching algorithm

In the following sections we will present several different algorithms which use some preprocessing before starting the actual comparison of the strings. We will distinguish two classes of methods, depending on whether the preprocessing is done on the pattern p or on the text t. We will start with some methods using a preprocessing of p. In Section 4.4 we will then present the concept of string matching via suffix trees, which is based on preprocessing the text.

4.2 String Matching Automata

With the first approach, which we will follow in this section, we will show that, after a clever preprocessing of the pattern, one scan of the text from left to right will suffice to solve the string matching problem. Furthermore we will see that the preprocessing can also be realized efficiently; it is possible in time in $O(|p| \cdot |\Sigma|)$, where Σ is the alphabet over which the text t and the pattern p are given. This approach is based on the concept of finite automata. We will now give a very short definition of finite automata, as much as is needed to understand the string matching method. A more detailed presentation can be found, for example, in the textbooks by Hromkovič [104] or Hopcroft et al. [103].

Informally speaking, a finite automaton can be described as a machine that reads a given text once from left to right. At each step, the automaton is in one of finitely many internal states, and this internal state can change after reading every single symbol of the text, depending only on the current state and the last symbol read. By choosing the state transitions appropriately, one can determine from the current state after reading a symbol whether the text contains the pattern as a suffix. This means that a special automaton can find all positions in the text t where the pattern p ends.

Definition 4.2. *A* finite automaton *is a quintuple* $M = (Q, \Sigma, q_0, \delta, F)$*, where*

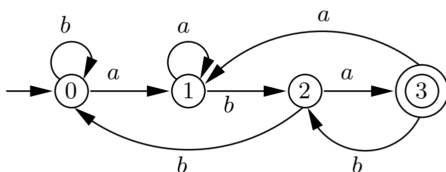


Fig. 4.2. A finite automaton for the string matching problem with the pattern $p = aba$. The states are shown as circles and the transition function is given by arrows labeled with the corresponding symbols from the alphabet Σ. The accepting state is marked by a double circle and the initial state is marked by an incoming unlabeled arrow

- Q *is a finite set of states,*
- Σ *is an input alphabet,*
- $q_0 \in Q$ *is the initial state,*
- $F \subseteq Q$ *is a set of accepting states , and*
- $\delta : Q \times \Sigma \to Q$ *is a transition function describing the transitions of the automaton from one state to another.*

We define the extension $\hat{\delta}$ *of the transition function to strings over* Σ *by* $\hat{\delta}(q, \lambda) = q$ *and* $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$ *for all* $q \in Q, a \in \Sigma$ *and* $x \in \Sigma^*$*. Thus,* $\hat{\delta}(q, x)$ *is the state* M *reaches from the state* q *by reading* x*.*

We say that the automaton M accepts *the string* $x \in \Sigma^*$*, if* $\hat{\delta}(q_0, x) \in F$*.*

To solve the string matching problem we construct a finite automaton, called *string matching automaton*, that, for a given pattern $p = p_1 \dots p_m$, accepts all texts ending with the pattern p. We first illustrate the idea of the construction with an example.

Example 4.2. We construct a string matching automaton M for the pattern $p = aba$. We define $M = (Q, \Sigma, q_0, \delta, F)$, where $\Sigma = \{a, b\}$, $Q = \{0, \dots, 3\}$, $q_0 = 0$, $F = \{3\}$, and δ is given by the following table:

state	input symbol	following state
0	a	1
0	b	0
1	a	1
1	b	2
2	a	3
2	b	0
3	a	1
3	b	2

The automaton M is shown graphically in Figure 4.2.

On the input $t = bababaa$ the automaton passes through the following states:

Read prefix	state of the automaton
λ	0
b	0
ba	1
bab	2
$b\boldsymbol{aba}$	**3**
$babab$	2
$bab\boldsymbol{aba}$	**3**
$bababaa$	1

We note that, after reading a prefix of t, the automaton is in the accepting state 3 if and only if the prefix ends with the pattern $p = aba$. ◊

In the following, we describe a systematic way to construct a finite automaton for any given pattern p that solves the string matching problem for p and an arbitrary text t, based on the method shown in the above example.

The idea behind this string matching automaton is the following: For a pattern p of length m we define a sequence of $m + 1$ states, connected by a path of transitions labeled with p. In addition to these transitions, which are directed from the initial state to the accepting state, we add outgoing transitions for every state labeled with the respective missing symbols from Σ. These transitions point in the backward direction or from a state to itself. The endpoints of these transitions can be determined from the overlap structure of the pattern with itself.

To describe the overlap structure, we need a variant of the overlap of two strings as defined in Section 3.1.

Definition 4.3. *Let s, t be strings. If there exist some strings x, y, and z from Σ^* satisfying the conditions*

(i) $s = xy$,
(ii) $t = yz$, *and*
(iii) $|y|$ *is maximal with (i) and (ii),*

then $\overline{Ov}(s, t) := y$ *is called the* generalized overlap *of s and t. We denote the length of* $\overline{Ov}(s, t)$ *by* $\overline{ov}(s, t)$.

The generalized overlap needed here differs from the overlap defined in Definition 3.5 in that any one of the strings s and t is allowed to be a substring of the other. Now we can determine the transition function of a string matching automaton as follows.

Definition 4.4. *Let $p = p_1 \dots p_m \in \Sigma^m$ for an arbitrary alphabet Σ. We define the* string matching automaton *for p as the finite automaton $M_p = (Q, \Sigma, q_0, \delta, F)$, where $Q = \{0, \dots, m\}$, $q_0 = 0$, $F = \{m\}$, and the transition function δ is defined by*

$$\delta(q, a) = \overline{ov}(p_1 \dots p_q a, p) \quad \text{for all } q \in Q \text{ and } a \in \Sigma.$$

Algorithm 4.2 Construction of a string matching automaton

Input: A pattern $p = p_1 \dots p_m$ over an alphabet Σ.

```
for q := 0 to m do
  for all a ∈ Σ do
    {Compute δ(q, a) = ov(p_1 ... p_q a, p)}
    k := min{m, q + 1} + 1
    repeat
      k := k − 1
    until p_1 ... p_k = p_{q−k+2} ... p_q a
    δ(q, a) := k
```

Output: The string matching automaton $M_p = (\{0, \dots, m\}, \Sigma, 0, \delta, \{m\})$.

Thus, the transition function of the string matching automaton for a pattern p can be constructed as shown in Algorithm 4.2.

In the following we will examine the connection between string matching automata and the calculation of overlaps more formally. We first need the following lemma.

Lemma 4.1. *Let Σ be an alphabet and let $n, m \in \mathbb{N}$. Let $x = x_1 \dots x_n \in \Sigma^n, y = y_1 \dots y_m \in \Sigma^m$, and $a \in \Sigma$. If $i = \overline{ov}(x, y)$, then*

$$\overline{ov}(xa, y) = \overline{ov}(y_1 \dots y_i a, y).$$

Proof. We start by proving the following inequality:

$$\overline{ov}(xa, y) \leqslant \overline{ov}(x, y) + 1. \tag{4.1}$$

To prove Inequality (4.1) we distinguish two cases: If $\overline{ov}(xa, y) = 0$ holds, then the proposition is obvious. Thus, let $\overline{ov}(xa, y) = r > 0$. Then $y_1 \dots y_r$ is a suffix of xa, and thus $y_1 \dots y_{r-1}$ is a suffix of x. This implies $\overline{ov}(x, y) \geqslant r - 1$, from which Inequality (4.1) follows.

We will now prove the claim of the lemma. From $i = \overline{ov}(x, y)$ we know that $x = x' y_1 \dots y_i$ for some $x' \in \Sigma^*$. Furthermore, $\overline{ov}(x' y_1 \dots y_i a, y) = \overline{ov}(y_1 \dots y_i a, y)$ holds, since we know that $\overline{ov}(x' y_1 \dots y_i a, y) \leqslant i + 1$ from Inequality (4.1). This immediately implies the claim. □

With Lemma 4.1 we are now able to prove that it is possible to compute the overlap of all prefixes of an arbitrary text with a pattern p using a string matching automaton.

Lemma 4.2. *Let $p = p_1 \dots p_m \in \Sigma^m$ be a pattern and let $M_p = (Q, \Sigma, q_0, \delta, F)$ be the string matching automaton for p. Let $t = t_1 \dots t_n \in \Sigma^n$ be an arbitrary text. Then, for all $i \in \{0, \dots, n\}$,*

$$\hat{\delta}(q_0, t_1 \dots t_i) = \overline{ov}(t_1 \dots t_i, p).$$

Proof. We will prove the claim by induction on i. For $i = 0$ the claim obviously holds, since $\hat{\delta}(q_0, \lambda) = q_0 = 0 = \overline{ov}(\lambda, p)$. For the induction step from i to $i+1$ we denote by q the state the automaton has reached after reading $t_1 \dots t_i$, i.e., $q = \hat{\delta}(q_0, t_1 \dots t_i)$. Then,

$$\begin{aligned}\hat{\delta}(q_0, t_1 \dots t_{i+1}) &= \delta(\hat{\delta}(q_0, t_1 \dots t_i), t_{i+1}) \\ &= \delta(q, t_{i+1}).\end{aligned}$$

Following the definition of the transition function of M_p, this implies

$$\hat{\delta}(q_0, t_1 \dots t_{i+1}) = \overline{ov}(p_1 \dots p_q t_{i+1}, p).$$

From the induction hypothesis we know that $q = \overline{ov}(t_1 \dots t_i, p)$, which together with Lemma 4.1 implies

$$\begin{aligned}\hat{\delta}(q_0, t_1 \dots t_{i+1}) &= \overline{ov}(p_1 \dots p_q t_{i+1}, p) \\ &= \overline{ov}(t_1 \dots t_i t_{i+1}, p).\end{aligned}$$

This completes the proof of the claim. □

Using Lemma 4.2 we are now able to prove that the string matching automaton for a pattern p really solves the string matching problem for an arbitrary text $t \in \Sigma^*$ and the pattern p.

Theorem 4.1. *Let $p = p_1 \dots p_m \in \Sigma^m$ be a pattern and let $M_p = (Q, \Sigma, q_0, \delta, F)$ be the string matching automaton for p. Let $t = t_1 \dots t_n \in \Sigma^n$ be an arbitrary text. Then, for all $i \in \{1, \dots, n\}$,*

$$p \text{ is a suffix of } t_1 \dots t_i \iff \hat{\delta}(q_0, t_1 \dots t_i) \in F.$$

Proof. The claim immediately follows from Lemma 4.2 and the fact $F = \{m\}$. □

This theorem enables us to solve the string matching problem using finite automata, as shown in Algorithm 4.3.

Now that we have seen how the string matching automata can solve the string matching problem, we will analyze the time complexity of the method. The construction of the string matching automaton M_p for a given pattern $p = p_1 \dots p_m \in \Sigma^m$ by Algorithm 4.2 needs time in $O(|\Sigma| \cdot m^3)$, since the automaton for p has exactly $m+1$ states, and the computation of a transition needs $O(m^2)$ time for every pair of state and input symbol. Nevertheless, there is a method known to construct the string matching automaton M_p in time $O(|\Sigma| \cdot m)$, but it is quite technical. Therefore we will not present it here, but refer the reader to the references given in Section 4.8.

The application of M_p on a text $t = t_1 \dots t_n$ needs time in $O(n)$ since the automaton reads each symbol of t exactly once. Overall, the string matching problem for p and t is solvable in time $O(n + |\Sigma| \cdot m)$ using finite automata.

Algorithm 4.3 String matching with finite automata

Input: A text $t = t_1 \dots t_n \in \Sigma^n$ and a pattern $p = p_1 \dots p_m \in \Sigma^m$.

Compute the string matching automaton $M_p = (Q, \Sigma, q_0, \delta, F)$ using Algorithm 4.2.
$q := q_0$
$I := \emptyset$
for $i := 1$ **to** n **do**
  $q := \delta(q, t_i)$
  if $q \in F$ **then**
    $I := I \cup \{i - m + 1\}$

Output: The set I of those positions where p starts as a substring in t.

This solution to the string matching problem is especially efficient if the task is to search for a given pattern p of length m in k different texts $t^{(1)}, \dots, t^{(k)}$. In this case one has to construct the string matching automaton for p only once in time $O(|\Sigma| \cdot m)$, and after that the actual search within the text $t^{(i)}$ is possible in time $O(|t^{(i)}|)$.

4.3 The Boyer–Moore Algorithm

In this section we will present the Boyer–Moore algorithm for the string matching problem. It is based on a similar idea as the naive algorithm, but saves a lot of comparisons by using clever preprocessing.

Although the time complexity of the Boyer–Moore algorithm can in the worst case be as high as that of the naive algorithm, in many practical applications its running time usually is very good. Therefore, it is often used in practice.

The Boyer–Moore algorithm utilizes the following basic idea. Similarly to the naive algorithm, it shifts the pattern along the text from left to right and compares $p_1 \dots p_m$ with $t_{j+1} \dots t_{j+m}$ for $0 \leqslant j \leqslant n - m$. But, in contrast to the naive algorithm, the comparison is done from right to left, i.e., p_m is compared to t_{j+m} first. As soon as a position i occurs where p_i and t_{j+i} differ, the pattern is shifted to the right. The number of positions the pattern can be shifted without missing an occurrence of p in t is computed using two rules. To apply these rules efficiently, preprocessing of the pattern is necessary.

The two rules are as follows. The *bad character rule* says that the pattern can be shifted to the rightmost occurrence of the symbol t_{j+i} in the pattern. The *good suffix rule* claims that the pattern can be shifted to the next occurrence of the suffix $p_{i+1} \dots p_m$ in p. On an intuitive level, this means that as we have already matched a suffix of the pattern, we know the following symbols in t, and thus we may shift the pattern to a position where this particular sequence of symbols occurs again. Please note that the bad character rule

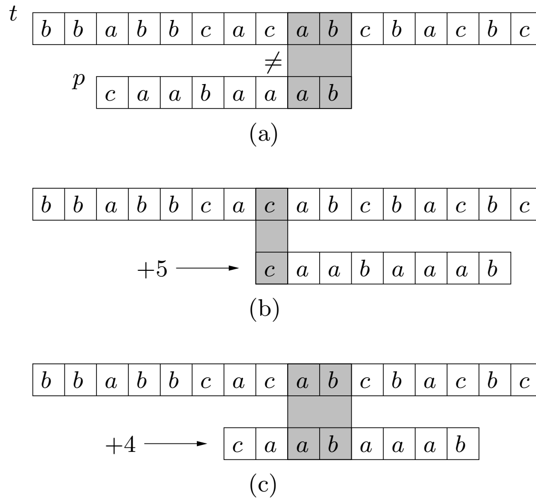


Fig. 4.3. An example for the application of the rules in the Boyer–Moore algorithm: (a) the comparison of the pattern p with a substring of the text t, the longest identical suffix of p is shaded grey; (b) the shift of the pattern for the next comparison according to the bad character rule; (c) the shift according to the good suffix rule

---

Algorithm 4.4 Preprocessing for the bad character rule

Input: A pattern $p = p_1 \dots p_m$ over an alphabet Σ.

for all $a \in \Sigma$ **do** $\beta(a) := 0$
for $i := 1$ **to** m **do** $\beta(p_i) := i$

Output: The function β.

---

might well propose a shift to the left, but such a shift will never be executed since the good suffix rule will always propose a shift to the right.

An example for the application of these rules is shown in Figure 4.3. At each step, the possible shift is calculated according to both rules and the shift with the larger value is realized.

We show that these rules are correct and efficiently implementable. Let us start with the bad character rule. For implementing this rule, it suffices to compute, in a preprocessing step, a function β that assigns the position of its last occurrence in p to each symbol from Σ (or the value 0, if the symbol does not occur in p). This computation can be done using Algorithm 4.4.

Algorithm 4.4 obviously has a running time in $O(m + |\Sigma|)$.

Lemma 4.3. *Let* $p_{i+1} \dots p_m = t_{j+i+1} \dots t_{j+m}$ *and* $p_i \neq t_{j+i} = a$*. Then the pattern can be shifted for the next comparison, according to the bad character rule, by* $i - \beta(a)$ *positions without missing an occurrence of* p *in the text* t*.*

Proof. For the proof we distinguish three cases:

(1) If the symbol a does not occur in p, i.e., if $\beta(a) = 0$ holds, the pattern p can obviously be shifted completely past t_{j+i}; thus, a shift by $i - \beta(a) = i$ positions is possible.
(2) If the last occurrence of a in p is at a position $k < i$, the pattern can be shifted by $i - k = i - \beta(a)$ positions to the right.
(3) If the last occurrence of a in p is at a position $k > i$, the bad character rule requires a shift of the pattern to the right by $i - k$ positions. Since $i - k$ is negative in this case, a shift to the left is indicated, which obviously is not helpful for the algorithm. Such a shift will never be executed since the good suffix rule always allows for a shift to the right, as we see below. □

We now consider the good suffix rule. For this rule, an efficient preprocessing step is also possible, for which we need the notion of suffix similarity of strings.

Definition 4.5. *Let s and t be two strings. We say that s is* suffix similar *to t, or $s \sim t$, if s is a suffix of t or t is a suffix of s.*

Intuitively, Definition 4.5 says that s and t can be aligned at their right ends such that all corresponding symbols are equal.

Using this notation, we reformulate the good suffix rule as follows:

Good Suffix Rule If $p_{i+1} \ldots p_m = t_{j+i+1} \ldots t_{j+m}$ and $p_i \neq t_{j+i}$, then shift the pattern for the next comparison to the right by $m - \gamma(i+1)$ positions, where

$$\gamma(i) = \max\{0 \leqslant k < m \mid p_i \ldots p_m \sim p_1 \ldots p_k\}.$$

We will show in the following that we can compute all values $\gamma(i)$ for all $2 \leqslant i \leqslant m$ in time $O(|\Sigma| \cdot m)$ using string matching automata.

We will do the computation of γ in two steps. By definition, the following holds for all $2 \leqslant i \leqslant m$:

$$\begin{aligned}\gamma(i) &= \max\{0 \leqslant k < m \mid p_i \ldots p_m \sim p_1 \ldots p_k\} \\ &= \max\{\max\{0 \leqslant k < m \mid p_i \ldots p_m \text{ is a suffix of } p_1 \ldots p_k\}, \\ &\qquad\quad \max\{0 \leqslant k < m \mid p_1 \ldots p_k \text{ is a suffix of } p_i \ldots p_m\}\}.\end{aligned}$$

These two cases are shown in Figure 4.4.

We start with determining, for all $2 \leqslant i \leqslant m$, the value

$$\gamma'(i) = \max\{0 \leqslant k < m \mid p_i \ldots p_m \text{ is a suffix of } p_1 \ldots p_k\}.$$

This is equivalent to solving the following subproblem.

Given a pattern $p = p_1 \ldots p_m$, compute the last occurrence of each suffix $p_i \ldots p_m$ of p within $p_1 \ldots p_{m-1}$, for $2 \leqslant i \leqslant m$.

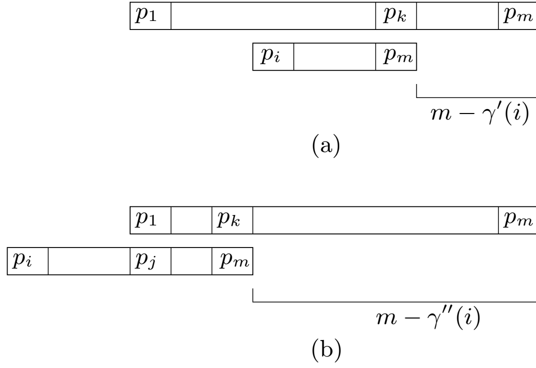


Fig. 4.4. (a) The case where $p_i \dots p_m$ is a substring of $p_1 \dots p_{m-1}$, where $\gamma'(i)$ denotes the last position $k \leqslant m-1$, such that $p_i \dots p_m$ is a suffix of $p_1 \dots p_k$. (b) The case where a suffix of $p_i \dots p_m$ is a prefix of $p_1 \dots p_{m-1}$, where $\gamma''(i)$ denotes the last position $k \leqslant m-1$, such that $p_1 \dots p_k$ is a suffix of $p_i \dots p_m$

To solve this subproblem we compute the function $\gamma' : \{2, \dots, m\} \to \{0, \dots, m-1\}$, where $\gamma'(i) = \max\{1 \leqslant k \leqslant m-1 \mid p_i \dots p_m = p_{k-m+i} \dots p_k\}$, if such a k exists, and $\gamma'(i) = 0$ otherwise.

To compute γ' using string matching automata, we reformulate the above problem.

> Given a pattern $p = p_1 \dots p_m$, compute the first occurrence of each prefix $p_m \dots p_i$ of $p^R = p_m \dots p_1$ in $p_{m-1} \dots p_1$ for $2 \leqslant i \leqslant m$.

This obviously is an equivalent formulation for all $1 \leqslant i \leqslant m-1$, $\gamma'(i) = \min\{1 \leqslant k \leqslant m-1 \mid p_m \dots p_i = p_k \dots p_{k-m+i}\}$, if such a k exists.

A first, naive approach for solving this problem could be to construct the string matching automata for $p_m \dots p_i$ for all $i \in \{2, \dots, m\}$ and to apply them to the text $p_{m-1} \dots p_1$. But this approach is far too time consuming, as it needs time in $O(|\Sigma| \cdot m^2)$. But the string matching automata for $p_m \dots p_i$ are very similar for all i. The next lemma shows that it suffices to construct the automaton for $p_m \dots p_2$.

Lemma 4.4. *Let $M_i = (\{0, \dots, m-i+1\}, \Sigma, \delta_i, 0, \{m-i+1\})$ be the string matching automaton for $p_m \dots p_i$ and let $t = t_1 \dots t_n$ be an arbitrary text. If $t_k \dots t_{k-m+i}$ is the first occurrence of $p_m \dots p_i$ in t, the following holds for all $1 \leqslant j \leqslant k-m+i$:*

$$\hat{\delta}_i(q_0, t_1 \dots t_j) = \hat{\delta}_2(q_0, t_1 \dots t_j). \tag{4.2}$$

If $p_m \dots p_i$ does not occur as a substring in t, Equation (4.2) holds even for all $1 \leqslant j \leqslant n$.

Proof. The claim of the lemma directly follows from the construction of the string matching automata. The outgoing transitions of the first $m-i+1$ states

are the same in both automata, M_i and M_2. To get into one of the additional states in M_2, the automaton must have reached the state $m - i + 1$ before. But this implies that the automaton has read the substring $p_m \dots p_i$. $\square$

Since we are only interested in the first occurrence of each prefix $p_m \dots p_i$, Lemma 4.4 allows us to simulate the computation of all string matching automata M_i for $2 \leqslant i \leqslant m$ by one computation of M_2 by just storing the sequence of reached states while reading the text $p_{m-1} \dots p_1$. Let $q_0, q_{m-1}, \dots, q_1$ be this sequence of states. Then $\gamma'(i)$ can for all $2 \leqslant i \leqslant m$ be determined as $\gamma'(i) = \max\{j \mid q_j = i\}$ if the state i was reached while reading the string $p_{m-1} \dots p_1$, and as $\gamma'(i) = 0$ otherwise.

We will now show how we can use the sequence $q_0, q_{m-1}, \dots, q_1$ of states to compute for all $2 \leqslant i \leqslant m$ the value of

$$\gamma''(i) = \max\{0 \leqslant k < m \mid p_1 \dots p_k \text{ is a suffix of } p_i \dots p_m\},$$

and thus also the function γ we are actually interested in.

If the string matching automaton M_2 for $p_m \dots p_2$ ends in state $q_1 = j$ after reading the text $p_{m-1} \dots p_1$, then $p_m \dots p_j$ are the last symbols read due to the construction of the automaton, and j is minimal with this property. In other words, $p_m \dots p_j$ is a suffix of $p_{m-1} \dots p_1$, and $p_m \dots p_{j'}$ is not a suffix of $p_{m-1} \dots p_1$ for all $j' < j$. This means that $p_j \dots p_m$ is a prefix of $p_1 \dots p_{m-1}$ and that j is minimal with this property. Thus, we can set $\gamma''(i) = m - j + 1$ for all $2 \leqslant i \leqslant j$ (see Figure 4.4 (b)). For all other values of i, $\gamma''(i) = 0$ holds.

These considerations can be put together to yield Algorithm 4.5 for computing the function γ for the good suffix rule.

The construction of the string matching automaton needs time in $O(|\Sigma| \cdot m)$ as we have seen in Section 4.2; all further steps of Algorithm 4.5 can obviously be done in time $O(m)$. Thus, the preprocessing for the good suffix rule needs time in $O(|\Sigma| \cdot m)$ overall.

The Boyer–Moore algorithm, as shown in Algorithm 4.6, now combines the above preprocessing steps with the scanning of the text according to the two rules.

We now analyze the running time of the Boyer–Moore algorithm. The preprocessing of the two functions β and γ needs time in $O(|\Sigma| \cdot m)$ as shown above. After each comparison, the pattern is shifted according to these two rules. Since $\gamma(i) < m - 1$ holds for all i, the shift proposed by the good suffix rule is always positive. But in the worst case it is possible that the pattern is shifted exactly one position to the right in every step, and the computation of this shift might even need a comparison of the complete pattern in every step. This means that the Boyer–Moore algorithm has a worst-case running time in $O(|\Sigma| \cdot m + n \cdot m)$, which does not improve over the naive algorithm. On the other hand, the Boyer–Moore algorithm is quite fast in practice; the worst case occurs very rarely. By modifying the preprocessing, it is also possible to guarantee a worst-case running time in $O(n + m)$ (see the bibliographical notes at the end of this chapter).

Algorithm 4.5 Preprocessing for the good suffix rule

Input: A pattern $p = p_1 \dots p_m$ over an alphabet Σ.

1. Construct the string matching automaton $M = (Q, \Sigma, q_0, \delta, F)$ for $p_m \dots p_2$.
2. Determine the sequence $q_0, q_{m-1}, \dots, q_1$ of states M is traversing while reading the input $p_{m-1} \dots p_1$.
3. Compute the function γ':

```
for i := 2 to m do γ'(i) := 0
for j := 1 to m − 1 do γ'(q_j) := j
```

4. Compute the function γ'':

```
for i := 2 to m do
  if i ⩽ q_1 then
    γ''(i) := m − q_1 + 1
  else
    γ''(i) := 0
```

5. Compute the function γ:

```
for i := 2 to m do γ(i) := max{γ'(i), γ''(i)}
```

Output: The function γ.

Algorithm 4.6 Boyer–Moore algorithm

Input: A pattern $p = p_1 \dots p_m$ and a text $t = t_1 \dots t_n$ over an alphabet Σ.

1. Compute from p the function β for the bad character rule.
2. Compute from p the function γ for the good suffix rule.
3. Initialize the set I of positions, where p starts in t, by $I := \emptyset$.
4. Shift the pattern p along the text t from left to right:

```
j := 0
γ(m + 1) := m {Good suffix rule not applicable for p_m = t_{j+m}}
while j < n − m do
  {Compare p_1 ... p_m and t_{j+1} ... t_{j+m} starting from the right}
  i := m
  while p_i = t_{j+i} and i > 0 do
    i := i − 1
  if i = 0 then
    I := I ∪ {j} {The pattern p starts in t at position j}
  {Compute the shift of the pattern according to the bad character rule and
  the good suffix rule}
  j := j + max{i − β(t_{j+i}), m − γ(i + 1)}
```

Output: The set I of all positions j in t where the pattern p starts.

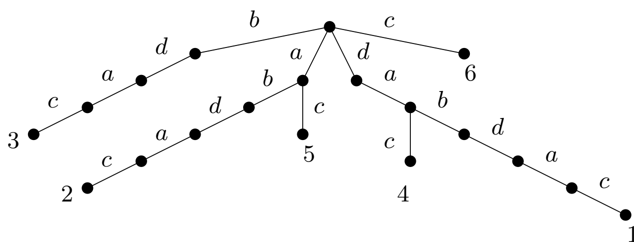


Fig. 4.5. A simple suffix tree for the text $t = dabdac$

4.4 Suffix Trees

In the preceding sections we have seen how to speed up the search for a solution for the string matching problem by preprocessing the pattern. In this section we will see that preprocessing of the text also can be advantageous. We will present the method of suffix trees which enables us, after a preprocessing of the text, to solve the string matching problem in linear time with respect to the length of the pattern. This method can offer a great speedup, especially in cases where we want to compare many different patterns to the same text. This problem often occurs in molecular biology; for example, when we want to compare many newly sequenced DNA fragments to a database of known DNA sequences. Furthermore, the method is applicable to many variants and extensions of the string matching problem as we show in the next section.

We know that the pattern occurs in the text if and only if it is the prefix of a suffix of the text. The idea behind the method of suffix trees is to compute and to efficiently store information about the suffix structure of the text. We will start with the definition of a simple variant of suffix trees, also called a *trie* in the literature.

Definition 4.6. *Let $t = t_1 \dots t_n \in \Sigma^n$ be a text. A directed tree $T_t = (V, E)$ with a root r is called a* simple suffix tree *for t if it satisfies the following conditions.*

1. *The tree has exactly n leaves which are labeled $1, \dots, n$.*
2. *The edges of the tree are labeled with symbols from Σ.*
3. *All outgoing edges from an inner vertex to its children are labeled with pairwise different symbols.*
4. *The path from the root to the leaf i is labeled $t_i \dots t_n$. (In this context, the labeling of a path is the concatenation of the edge labels on this path.)*

Figure 4.5 shows a simple suffix tree for the text *dabdac* as an example.[1]

Now, a natural question is whether there exists a simple suffix tree for any arbitrary text t. Unfortunately, this is not the case. A simple suffix tree for a

[1] We assume that all edges are directed from the root in direction to the leaves, and therefore we do not mark the direction of the edges in our figures.

Algorithm 4.7 Construction of a simple suffix tree

Input: A string $t = t_1 \ldots t_n \in \Sigma^n$.

Let $t' := t\$$ for a symbol $\$ \notin \Sigma$, let $\Sigma' = \Sigma \cup \{\$\}$.
{Compute a simple suffix tree $T_{t'}$ for t'}
Initialize $T_{t'}$ with a root r and an empty set of edges.
for $i := 1$ **to** n **do**
  {Insert the suffix $t_i \ldots t_n\$$ into the tree}
  Starting from the root r, search for a path in $T_{t'}$, labeled with a maximal prefix $t_i \ldots t_{j_i}$ and ending in the vertex x_i. {This path is uniquely determined and does not end in a leaf.}
  Add a path $x_i, y_{i_{j_i+1}}, \ldots, y_{i_n}, y_{i_{n+1}}$, labeled $t_{j_i+1} \ldots t_n\$$, to the tree, where $y_{i_{j_i+1}}, \ldots, y_{i_n}, y_{i_{n+1}}$ are new, additional vertices.
  Label the new leaf $y_{i_{n+1}}$ with i.

Output: The constructed simple suffix tree $T_{t'}$ for $t' = t\$$.

---

text t exists if and only if no suffix of t is also a prefix of another suffix of t. Since, if a suffix s is a prefix of another suffix s', then the path labeled s, which is uniquely determined according to condition (3), does not end in a leaf of the tree, contradicting condition (4) of the definition of a simple suffix tree. Fortunately, there is a simple strategy to extend the applicability of suffix trees to arbitrary texts. This strategy is to append a new symbol $\$ \notin \Sigma$ to the text t, to construct the suffix tree for $t\$$, and to subsequently ignore all edge labels \$. The use of the additional symbol \$ guarantees that no suffix of $t\$$ can be the prefix of another suffix.

Algorithm 4.7 constructs a simple suffix tree for a text $t\$$. In Figure 4.6, an example for the construction of a simple suffix tree is shown. We will prove the correctness of Algorithm 4.7 in the following.

Theorem 4.2. *Let $t = t_1 \ldots t_n \in \Sigma^n$ be a string and let $\$ \notin \Sigma$. Then Algorithm 4.7 constructs a simple suffix tree for $t\$$.*

Proof. We have to show that the graph constructed by Algorithm 4.7 is a directed tree satisfying conditions (1) to (4) from Definition 4.6. In each step, only one path, consisting of newly added vertices, is added to the graph via exactly one new edge. Since the construction starts with a single vertex, which is a tree, the resulting graph is also a tree after each step. We now show that conditions (1) to (4) are also satisfied.

(1) The suffixes are inserted into the tree in order of decreasing length. This ensures that in every step i the vertex x_i, where the new path is appended, is not a leaf. Furthermore, since no suffix of $t\$$ can be the prefix of another suffix, the path added in step i is nonempty for all i. In each of the $n + 1$ steps the number of leaves hence increases by one; thus, condition (1) is satisfied.

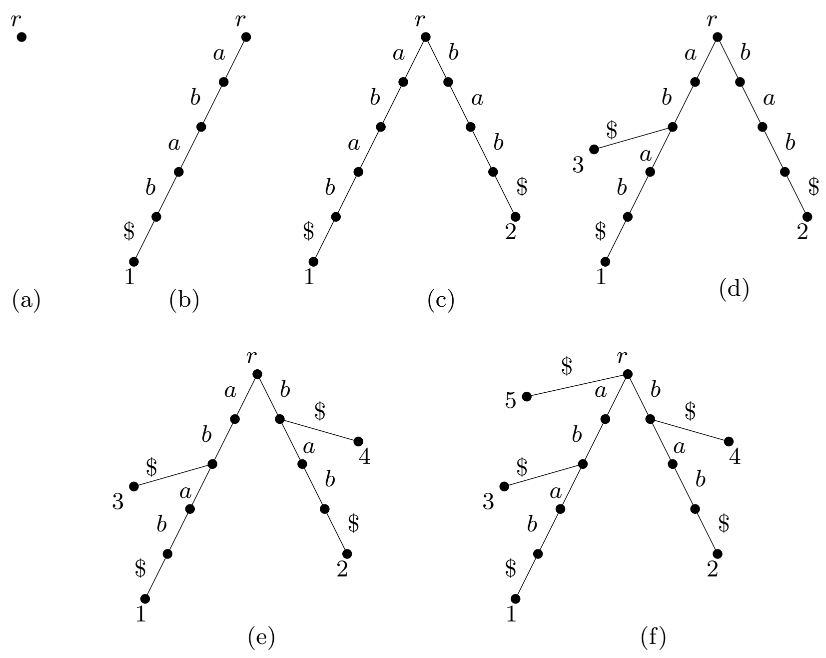


Fig. 4.6. The construction of a simple suffix tree for the text $abab\$$ using Algorithm 4.7

(2) Condition (2) obviously holds (for the alphabet $\Sigma' = \Sigma \cup \{\$\}$).
(3) Condition (3) follows from the fact that in step i the algorithm searches for the path of maximal length, starting in the root and labeled with a prefix of $t_i \dots t_n\$$, and appends the new path at the end vertex of this maximal path. If the algorithm would add an edge to x_i with a label $a \in \Sigma'$ already existing on another outgoing edge of x_i, the path to x_i would not be maximal.
(4) Condition (4) directly follows from the construction of the algorithm. □

If we know a simple suffix tree T_t for the text t, we can easily solve the string matching problem for t and an arbitrary pattern p. We start in the root of the tree T_t and search for a path labeled p. If such a path exists, it is uniquely determined due to condition (3) in the definition of a simple suffix tree. In this case, p is the prefix of a suffix of t, i.e., it is a substring of t. Every leaf in the subtree rooted at the end vertex of this path corresponds to an occurrence of p in t. If there is no path from the root in T_t that is labeled p, then p obviously is no substring of t.

Using this method we can decide in $O(|p|)$ time whether p is a substring of t or not. If we want to find all positions where p starts in t, the running time depends on the size of the subtree rooted at the end vertex labeled p.

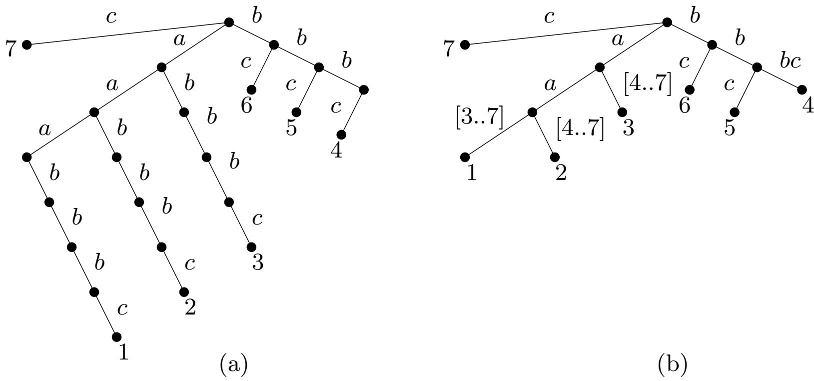


Fig. 4.7. (a) The simple suffix tree for the text $t_3 = a^3b^3c$ from Example 4.3; (b) the compact suffix tree for t_3

This raises a problem, since the simple suffix tree for a string t can reach a size in $\Omega(|t|^2)$, and thus the subtree can be too large for efficient traversal.

Example 4.3. An example where the size of a simple suffix tree is of quadratic order is the text $t_n = a^n b^n c$ for $n \in \mathbb{N}$. The simple suffix tree for t_3 is depicted in Figure 4.7 (a). $\Diamond$

This problem can be solved by using a more efficient representation of suffix trees. To shorten the description of a suffix tree we use two ideas. First, we can eliminate all vertices of outdegree 1 in a suffix tree if we allow for (nonempty) strings over Σ as edge labels. Second, we do not have to write down the strings occurring as edge labels. Since all of them are substrings of t, it suffices to write down their starting and ending positions in t. This shortens the representation of an edge label of length k if $2 \cdot \log_2 n \leqslant k \cdot \log_2 |\Sigma|$, and thus $k \geqslant 2 \cdot \log_2(n - |\Sigma|)$ holds. Note that we need $\log_2 |\Sigma|$ bits for coding one alphabet symbol, which is especially important in the case of large alphabet sizes. These considerations lead us to the following definition.

Definition 4.7. *Let $t = t_1 \dots t_n \in \Sigma^n$ be a text. A directed tree $T_t = (V, E)$ with a root r is called a* compact suffix tree *for t if it satisfies the following conditions:*

1. *The tree has exactly n leaves, labeled $1, \dots, n$.*
2. *Every inner vertex of T_t has at least two children.*
3. *The edges of the tree are labeled with substrings of t; every substring of length k is represented by its starting and ending position in t if $k \geqslant 2 \cdot \log_2(n - |\Sigma|)$ holds.*
4. *All labels of the edges going out from an inner vertex to its children start with pairwise different symbols.*

5. *The path from the root to the leaf i is labeled $t_i \dots t_n$. Here, the labeling of a path is understood as the concatenation of the (uncompressed) edge labels on the path.*

As an example, Figure 4.7 (b) shows the compact suffix tree for the text $t_3 = a^3b^3c$ from Example 4.3. In the following we determine the size of a compact suffix tree.

Lemma 4.5. *Let $t = t_1 \dots t_n \in \Sigma^n$ be a text. A compact suffix tree T_t for t has $O(n \log n)$ size.*[2]

Proof. Every suffix tree for t has exactly n leaves. Since T_t is a compact suffix tree for t, every inner vertex of T_t has at least two children. This implies that T_t has at most $n-1$ inner vertices. Thus T_t has at most $2n-1$ vertices and hence at most $2n-2$ edges. The label on each edge has length in $O(\log n)$; thus, the representation of T_t needs space in $O(n \log n)$ overall. $\diamondsuit$

We will now describe an algorithm that computes a compact suffix tree for a given string t by first constructing a simple suffix tree for t and then transforming it into a compact suffix tree. For this algorithm we first need the following definition.

Definition 4.8. *Let $t = t_1 \dots t_n \in \Sigma^n$ be a string and let $t' = t\$$ for a symbol $\$ \notin \Sigma$. Let $T_{t'} = (V, E)$ be a (simple or compact) suffix tree for t' with an edge labeling function label : $E \to \Sigma^*$. For every vertex $x \in V$, we define the* path label *pathlabel(x) as the labeling of the path from the root to x, i.e., as the concatenation of the (uncompressed) edge labels on the path. Furthermore, we define the* string depth *depth* of x in *$T_{t'}$ to be the length of the path label pathlabel(x), i.e., the string depth is the sum of the lengths of the (uncompressed) edge labels on the path from the root to x.*

Moreover, if $T_{t'}$ is a simple suffix tree, we define Pos(x) for every vertex $x \in V$ as the minimal label of a leaf in the subtree of $T_{t'}$ rooted at x.

In a simple suffix tree the string depth of a vertex coincides with the usual graph-theoretic definition of the depth of this vertex, i.e., with the number of edges on the path from the root to this vertex.

---

[2] In the literature, the size of a compact suffix tree is often given as linear, i.e., as in $O(n)$. This results from the use of two different complexity measures. If we assume that all occurring values are of approximately the same size, we can view the size as a constant. This is called a measurement according to the *uniform cost measure*. If this simplification is not necessary for the analysis, or the size of the values significantly varies and depends on the input size, we often use the so-called *logarithmic cost measure* that takes into account the size of the used values. This cost measure enlarges the computed complexity by a logarithmic factor. But one should keep in mind that the input size also has to take into account this logarithmic factor. For example, the size of an input string $s = s_1 \dots s_n$ is $n \cdot \log_2 |\Sigma|$ within this model.

Algorithm 4.8 Construction of a compact suffix tree

Input: A string $t = t_1 \dots t_n \in \Sigma^n$.

1. Let $t' := t\$$ for a symbol $\$ \notin \Sigma$, let $\Sigma' := \Sigma \cup \{\$\}$, and compute a simple suffix tree $T_{t'} = (V, E)$ with edge labeling $label : E \to \Sigma$ for t' using Algorithm 4.7.
2. Eliminate the vertices of degree 2:

   Let $X := \{v \in V \mid v \text{ has exactly one child}\}$.
   while $X \neq \emptyset$ **do**
     Choose an $x \in X$, let y be the parent of x, let z be the child of x.
     Replace the edges (y, x) and (x, z) by the edge (y, z) with the label $label(y, z) = label(y, x)label(x, z)$ and delete the vertex x.
3. Compress the long edge labels:

   for all $e = (x, y) \in E$ **do**
     if $|label(e)| \geqslant 2 \cdot \log_2(n - |\Sigma'|)$ **then**
       {Replace the label of the edge by the corresponding pair of starting and ending positions in t'}
       $label'(e) := [Pos(y) + depth(x), Pos(y) + depth(x) + |label(e)| - 1]$
     else
       $label'(e) := label(e)$

Output: The constructed compact suffix tree with edge labeling $label'$.

Using the notions from Definition 4.8, we are now able to compute the edge labels of the compact suffix tree. To determine the compressed label of an edge we have to find a position in t' where the uncompressed label of the edge starts as a substring. For an edge (v, x) in the tree, $Pos(x) + depth(v)$ gives the first position in t', where the label of (v, x) starts: $Pos(x)$ is the starting position of the first (longest) suffix of t' containing the label of this edge as a substring, and $depth(v)$ is the number of symbols of this suffix that have already been read. Based on this idea, Algorithm 4.8 constructs a compact suffix tree for a given string.

Since the simple suffix tree can be of quadratic size in the length of string t, Algorithm 4.8 has a worst-case running time in $O(n^2)$. But there are algorithms known for constructing a compact suffix tree directly in linear time, without constructing the simple suffix tree in an intermediate step.

Theorem 4.3. *A compact suffix tree for a given string $s = s_1 \dots s_n$ can be constructed in $O(n \log n)$ time, i.e., in time linear in the size of the suffix tree.* □

We will not prove Theorem 4.3 here since the known algorithms for linear-time suffix tree construction are technically quite involved. Instead, we refer the reader to the references given in Section 4.8.

The following Algorithm 4.9 describes how the string matching problem can be solved using compact suffix trees. The idea behind this algorithm is to construct the compact suffix tree for the given text, and to search for a path

in the suffix tree that starts in the root and is labeled with the given pattern. If the given pattern ends within the label of the edge (v, x), the labels of all leaves in the subtree rooted at x correspond exactly to those positions in the text where the pattern starts as a substring.

We now analyze the time complexity of Algorithm 4.9.

Theorem 4.4. *Algorithm 4.9 solves the string matching problem for a text $t = t_1 \dots t_n$ and a pattern $p = p_1 \dots p_m$ over an alphabet Σ in time in $O(n \log n + m \cdot |\Sigma| + k)$, where k is the number of occurrences of p in t, and $|\Sigma| < \frac{n}{c}$ holds for some constant $c \geqslant 2$.*

Proof. Following the considerations above, it is clear that Algorithm 4.9 solves the string matching problem. The construction of the compact suffix tree is possible in $O(n \log n)$ due to Theorem 4.3. We will in the following analyze the time used by an appropriate implementation of the remainder of Algorithm 4.9.

The initialization in step 2 can obviously be done in constant time. We now determine the time necessary for executing the main loop in step 3. We start by determining the overall time the algorithm needs for finding the correct child of the current vertex in all executions of step 3 (a). We note that an edge label of the form $[a..b]$ codes for a substring of length $\Omega(\log n)$, since $|\Sigma| < \frac{n}{c}$ holds, and thus the minimum length $2 \cdot \log_2(n - |\Sigma|)$ of a compressed label is in $\Omega(\log n)$. This means that, while reading p, at most $O(\frac{m}{\log n})$ compressed edge labels can occur on the path from the root. Reading such a compressed edge label needs time in $O(\log n)$. Every current vertex has up to $|\Sigma|$ children, which have all to be checked in the worst case. If the algorithm checks the children of x in an order where all edges with uncompressed labels precede those with compressed labels, we can guarantee that compressed edge labels are only read in steps in which the correct label also is compressed. Computing such an ordering is obviously possible in $O(|\Sigma|)$ time. Thus, the reading of compressed edge labels overall needs time in $O(\frac{m}{\log n} \cdot \log n \cdot |\Sigma|) = O(|\Sigma| \cdot m)$. For all uncompressed edge labels, only the first symbol has to be read to find the correct edge. Any path from the root to a vertex in the tree at which it can be decided whether p occurs as a substring in p or not, has at most m edges. Thus, the algorithm needs at most $O(|\Sigma| \cdot m)$ time for reading the uncompressed edge labels. Overall, this implies that finding the correct outgoing edge for the actual current vertex needs $O(|\Sigma| \cdot m)$ time.

The comparison of p with the labels on the correct edges in all executions of step 3 (b) can be done in $O(m)$ time since every symbol in p is compared exactly once, and since for the edges with compressed edge labels, only a prefix of the uncompressed label is considered that does not contain any symbol not needed for the comparison.

It remains for us to estimate the time complexity for traversing the subtree whose leaves determine the starting positions of p in t in Step 4. Since this subtree has exactly k leaves, it has at most $2k - 1$ vertices overall, since each

Algorithm 4.9 String matching with compact suffix trees

Input: A pattern $p = p_1 \dots p_m$ and a text $t = t_1 \dots t_n$ over an alphabet Σ.

1. Construct the compact suffix tree $T_{t'} = (V, E)$ for $t' = t\$, \$ \notin \Sigma$, with root r and edge labeling function *label*$'$.
2. Initialization:
   $x := r$ {Current vertex}
   $i := 1$ {Current position in p}
   found := false {Pattern not yet found}
   possible := true {Finding the pattern still possible}
3. **while not** *found* **and** *possible*
   a) Search for an edge starting at x, whose label starts with p_i:
      fitting := false {Label p_i not yet found}
      U := set of the children of x {Children of x that still have to be investigated}
      while not *fitting* **and** $U \neq \emptyset$ **do**
         Choose $v \in U$.
         if $label'((x, v)) = p_i\alpha$ for some $\alpha \in (\Sigma \cup \{\$\})^*$ **then**
            fitting := true
            $label := label'((x, v))$
         else if $label'((x, v)) = [k..l]$ **and** $t_k = p_i$ **then**
            fitting := true
            $l' := \min\{l, k + m - i\}$
            $label := t_k \dots t_{l'}$ {Read only the part of the edge label that is necessary for the comparison with the remainder of p}
         else
            $U := U - \{v\}$

   b) Compare edge label with that part of p that still has to be read:
      if ($p_i \dots p_m$ is no prefix of *label*) **and** (*label* is no prefix of $p_i \dots p_m$) **then**
         possible := false {p does not occur as a substring in t}
      else if *label* is a prefix of $p_i \dots p_m$ **then**
         $x := v$
         $i := i + |label|$
      else {$p_i \dots p_m$ is a prefix of *label*}
         $x := v$
         found := true
4. **if** *found* **then**
   Compute the set I of leaf labels in the subtree rooted at x, for example, using a depth-first search in this subtree.

Output: The set I of positions in t, where p starts as a substring.

inner vertex has at least two children. Due to the fact that the traversal can ignore the edge labels, it can be executed in $O(k)$ time. □

From the running time analysis of Algorithm 4.9 we can conclude that the use of suffix trees for the string matching problem is especially useful if one wants to search for several different patterns within the same text. In this case, the suffix tree has to be constructed only once and each given pattern p of length m that occurs k times in the text can then be found in $O(m \cdot |\Sigma| + k)$ time.

4.5 Further Applications of Suffix Trees

In this section, we will introduce some variants and generalizations of the string matching problem as well as some other string problems that can be efficiently solved using suffix trees.

4.5.1 Generalized Suffix Trees and the Substring Problem

This subsection is dedicated to a generalization of suffix trees that can be used to store information about several different texts. We furthermore present the substring problem as the first application of these generalized suffix trees. Let us start with the definition of the problem.

Definition 4.9. *The* substring problem *is the following computing problem:*

Input: A pattern p and N texts $t_1, \ldots, t_N$ over an alphabet Σ.
Output: A set $I \subseteq \{1, \ldots, N\}$ of indices such that $i \in I$ if and only if p is a substring of t_i.

Database searching is one of the typical applications of the substring problem, for example, finding all DNA sequences in a given database that contain a newly sequenced DNA fragment as a substring.

For solving the substring problem efficiently, we construct a suffix tree that, for a given sequence of texts $t_1, \ldots, t_N$ over an alphabet Σ, contains all suffixes of the single texts. The idea behind the construction of such a suffix tree is to first concatenate all texts, separated by pairwise different separator symbols $\$_1, \ldots, \$_N \notin \Sigma$. After that we construct a compact suffix tree for the string $t' = t_1\$_1t_2\$_2 \ldots t_N\$_N$ without compressing the edge labels. This suffix tree then obviously contains all suffixes of the strings $t_1, \ldots, t_N$ in its path labels, but the suffixes of the texts $t_1, \ldots, t_{N-1}$ do not end in the leaves of the tree. We will now show that all separator symbols only occur in the labels of edges incident to the leaves of the tree.

Lemma 4.6. *Let the texts $t_1, \ldots, t_N$ over an alphabet Σ be given. Let T be the compact (uncompressed) suffix tree for $t' = t_1\$_1t_2\$_2 \ldots t_N\$_N$, where $\$_1, \ldots, \$_N \notin \Sigma$ are pairwise different. Then the separator symbols $\$_i$, $1 \leqslant i \leqslant N$, only occur in the labels of edges incident to a leaf of T.*

Proof. If the separator symbol $\$_i$ would occur in the label of an edge between two inner vertices x and y, where x is the parent of y, then at least two different suffixes $w\$_iu$ and $w\$_iv$ would have to exist in t', since y has at least two children. But this obviously contradicts the fact that every separator symbol occurs exactly once in t'. □

This lemma implies that we can get the desired tree containing all suffixes of all given texts by deleting, for every edge incident to a leaf, the part of the label after the first occurrence of a separator symbol. In the resulting tree, we can then obviously compress all long edge labels as described in Section 4.4 by replacing them with pointers to their starting and ending positions in t'. We will summarize these considerations in the following definition.

Definition 4.10. *Let $t_1, \ldots, t_N$ be texts over an alphabet Σ, and let $\$_1, \ldots, \$_N \notin \Sigma$ be pairwise different symbols. A* generalized suffix tree for $t_1, \ldots, t_N$ *is constructed from a compact, uncompressed suffix tree for $t' = t_1\$_1t_2\$_2 \ldots t_N\$_N$ by the following steps:*

1. *Replace every edge label of the form $u\$_iw$, where $u \in \Sigma^*$ and $w \in (\Sigma \cup \{\$_j \mid 1 \leqslant j \leqslant N\})^*$, with the edge label $u\$_i$.*
2. *Label every leaf with a pair (i, j) consisting of the index of the corresponding text t_i and the starting position j of the corresponding suffix in t_i.*
3. *Compress the long edge labels as described in Section 4.4.*

In Step 1, the suffix after the first occurrence of a separator symbol is removed from each edge label. The labeling of the leaves in Step 2 can be done efficiently as follows: The parameter i is the same as the index of the separator symbol in the label of the incident edge, and j can be determined from the leaf label of the compact suffix tree for t' and the lengths of the given texts. We will now illustrate Definition 4.10 with an example.

Example 4.4. We consider the two texts aba and ab. Figure 4.8 (a) shows the compact suffix tree for $aba\$_1ab\$_2$, and Figure 4.8 (b) shows the generalized suffix tree for aba and ab.

As an example, the label of the leaf $(2, 1)$ can be determined as follows: The first component of the leaf label is given as the index of the separator symbol that is the end symbol of the label of the edge incident to the leaf, i.e., it indicates the text to which the suffix corresponding to the leaf belongs. The second component indicates the starting position of this suffix in the text; it can be calculated from the label of the corresponding leaf in the compact suffix tree, as shown in Figure 4.8 (a), by subtracting the length of the first text (including the first separator symbol). ◊

Using generalized suffix trees, we can now solve the substring problem in a way analogous to the one we have described for the string matching problem using suffix trees in Section 4.4. We first construct the generalized suffix tree for the given texts and then search for a path starting from the root and

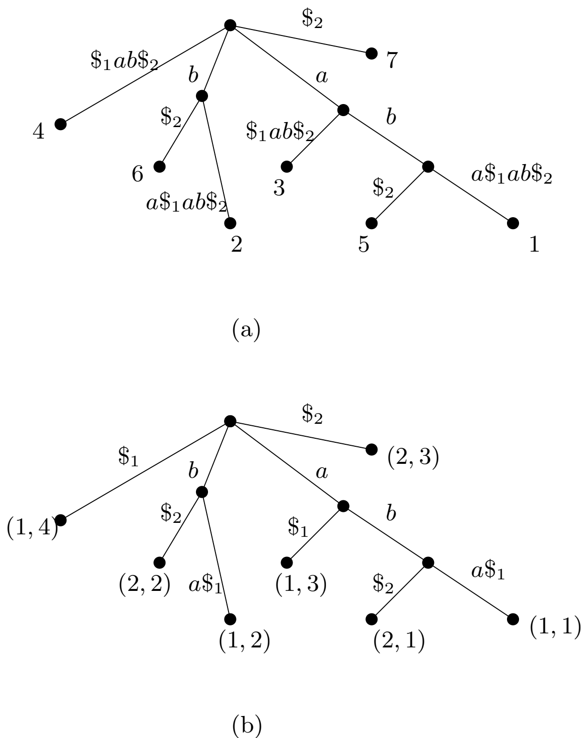


Fig. 4.8. The construction of a generalized suffix tree for the texts *aba* and *ab*

labeled with the given pattern. If we have found such a path, the leaves of the subtree beneath this path describe all occurrences of the pattern in the texts. The desired index set I can thus be determined as the set of all first components of leaf labels in this subtree, or equivalently as the set of all indices i such that $\$_i$ occurs in an edge label in this subtree.

Since the generalized suffix tree has a size in $O(n \log n)$, where n denotes the total length of all texts, and since it can also be constructed in time $O(n \log n)$, with a construction similar to that of the compact suffix tree for a single text (see the references in Section 4.8), we have the following time complexity for solving the substring problem.

Theorem 4.5. *Let* $p = p_1 \dots p_m$ *be a pattern and let* $t_1, \dots, t_N$, $N \in \mathbb{N} - \{0\}$ *be* N *texts of total length* n *over an alphabet* Σ. *Then the substring problem is solvable in* $O(n \log n + m \cdot |\Sigma| + k)$ *time, where* k *is the number of occurrences of* p *in* $t_1, \dots, t_N$, *and* $|\Sigma| < \frac{n}{c}$ *holds for some constant* $c \geqslant 2$.

Proof. The proof is analogous to that of Theorem 4.4. The construction of the generalized suffix tree for $t_1, \dots, t_N$ is possible in $O(n \log n)$ time, the search for the pattern p in the suffix tree needs time in $O(m \cdot |\Sigma|)$, and the traversal of the subtree can be done in $O(k)$ time. □

Algorithm 4.10 Computation of the longest common substring

Input: A sequence $t_1, \ldots, t_N$ of N strings over an alphabet Σ.

1. Construct the generalized suffix tree T for $t_1, \ldots, t_N$.
2. Label every inner vertex x of the suffix tree with a subset $M(x) \subseteq \{1, \ldots, N\}$, such that $i \in M(x)$ if and only if there exists a leaf labeled (i, j) in the subtree rooted at x for some arbitrary j, i.e., if a suffix of t_i is contained in this subtree.
3. Between all inner vertices of T with label $\{1, \ldots, N\}$, find a vertex $x_{\max}$ with maximum string depth, i.e., with $depth(x_{\max}) = \max\{depth(x) \mid M(x) = \{1, \ldots, N\}\}$.
4. Compute $\alpha_{\max}$ as the label of the path from the root of the suffix tree to $x_{\max}$.

Output: The longest common substring $\alpha_{\max}$ of $t_1, \ldots, t_N$.

4.5.2 Longest Common Substrings

This subsection is dedicated to determining the longest common substring of a set of given strings. This task can also be easily and efficiently accomplished using generalized suffix trees. The problem of finding common substrings occurs in molecular biology, for example, in the context of finding especially important regions in the DNA sequence: To find regions in the DNA containing genes which are important for the survival of the organism, we can use the following approach. We examine the DNA of several closely related organisms. Then we can assume that these DNA sequences agree (almost) perfectly in those regions necessary for the survival of the organism, since mutations in these regions are much more improbable than elsewhere. This means that common substrings can point out important coding regions in the DNA. In the following, we present a method for finding the longest common substring of a given set of strings. This problem can formally be posed as follows.

Definition 4.11. *The problem of determining the longest common substring of a given set of strings, the* longest common substring problem*, is the following optimization problem:*

Input: A set $M = \{t_1, \ldots, t_N\}$ of strings over an alphabet Σ.
Feasible solutions: All strings t that are a substring of t_i for all $1 \leqslant i \leqslant N$.
Costs: The costs of a feasible solution t are $cost(t) = |t|$.
Optimization goal: Maximization.

Algorithm 4.10 solves the longest common substring problem. For any vertex x of the suffix tree, $depth(x)$ denotes the sum of the lengths of the edge labels on the path from the root to the vertex x, as defined in Definition 4.8.

Before proving the correctness of Algorithm 4.10, we illustrate its work with an example.

Example 4.5. Let the strings $t_1 = bcabcac$, $t_2 = aabca$, and $t_3 = bcaa$ be given. A generalized suffix tree for t_1, t_2, and t_3 is shown in Figure 4.9. The boxed

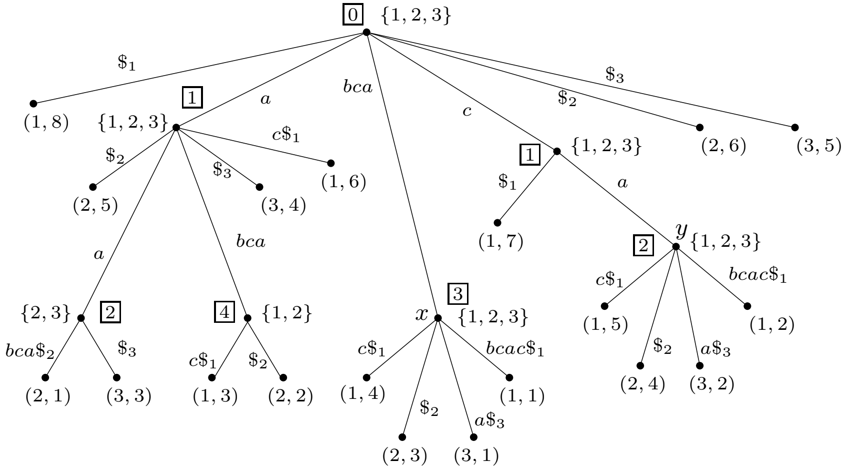


Fig. 4.9. The determination of a longest common substring for the strings $t_1 = bcabcac$, $t_2 = aabca$, and $t_3 = bcaa$ using Algorithm 4.10

numbers in the figure show the string depth $depth(z)$ for every inner vertex z. Furthermore, for every inner vertex z, the set $M(z)$ constructed in step 2 of Algorithm 4.10 is shown. The algorithm now determines the vertex with the greatest string depth, whose M-set contains the indices of all input strings. In our example this is the vertex x with string depth $depth(x) = 3$. The longest common substring, in our example the string bca, can now be read as the path label on the path from the root to the vertex x.

This example shows that it is important to use the string depth, i.e., the total length $depth(x)$ of all edge labels on the path from the root to the vertex x, as a measure for the depth of a vertex x, and not, for example, the number of edges on this path. The deepest vertex according to the latter measure is the vertex y, whose path label corresponds to the non-maximal common substring ca. ◊

Theorem 4.6. *Let N strings $t_1, \ldots, t_N$, $N \in \mathbb{N} - \{0\}$, of total length n over an alphabet Σ be given. Then, Algorithm 4.10 computes the longest common substring of $t_1, \ldots, t_N$ in $O(n \cdot (\log n + N \cdot (N + |\Sigma|)))$ time.*

Proof. We first show that Algorithm 4.10 computes the longest common substring for $t_1, \ldots, t_N$. Let, for every vertex x of the generalized suffix tree, $\alpha(x)$ be the path label on the path from the root to the vertex x. Then $i \in M(x)$ holds if and only if $\alpha(x)$ is a prefix of a suffix of t_i. Thus, $\alpha(x)$ is a common substring of all strings t_i with $i \in M(x)$. Hence, $\alpha(x)$ is a common substring of $t_1, \ldots, t_N$ if and only if $M(x) = \{1, \ldots, N\}$ holds. Since $x_{\max}$ is chosen by the algorithm such that $\alpha_{\max} = \alpha(x_{\max})$ is the longest substring with this property, $\alpha_{\max}$ is a longest common substring.

We will now analyze the running time of Algorithm 4.10. The construction of the generalized suffix tree needs time in $O(n \log n)$. The labeling of the inner vertices can be done bottom-up, starting with the leaves. The label of an arbitrary inner vertex x can be computed as the union of the sets of labels of its children. Since the separator symbols can only occur in the labels of edges incident to a leaf of the generalized suffix tree according to Lemma 4.6, only the parent vertex of a leaf can have up to $N + |\Sigma|$ children. All other inner vertices have at most $|\Sigma|$ children. Thus, for the parent vertices of the leaves, a union of up to $N + |\Sigma|$ sets has to be constructed, while, for all other inner vertices, a union of at most $|\Sigma|$ sets of cardinality $\leqslant N$. In total, this needs time in $O(n \cdot (N + |\Sigma|) \cdot N + n \cdot |\Sigma| \cdot N) = O(n \cdot N \cdot (N + |\Sigma|))$. For the execution of Algorithm 4.10, we only need the information whether or not, for a given vertex x, the set $M(x)$ equals the set $\{1, \ldots, N\}$ of all indices; we can easily store this information in every inner vertex while constructing the sets $M(x)$. Then, checking this condition is possible in constant time for every vertex x. Finding the vertex $x_{\max}$ and reading $\alpha_{\max}$ is thus obviously possible in linear time with respect to the size of the suffix tree, i.e., in $O(n \log n)$. □

One can easily see that the above method can be generalized to compute the longest common substring of at least k of the given N strings. Furthermore, the starting positions of all occurrences of the longest common substring in $t_1, \ldots, t_N$ can also be computed using this method.

Moreover, this approach can be used to compute all common substrings of a given length l. This can be done by outputting the path labels of all paths leading to a vertex with label $\{1, \ldots, N\}$ and string depth l. This task is motivated for $N = 2$ by the biological problem of DNA contamination. If a DNA sequence is duplicated by cloning, it is often contaminated by parts of the DNA of the host organism. If the DNA sequence of the host organism is known, one can test after duplication, whether the duplicated DNA is contaminated by comparing it to the host DNA and searching for common substrings exceeding a certain length threshold.

4.5.3 Efficient Computation of Overlaps

In this subsection we will present an efficient method for computing all pairwise overlaps for a given set of strings using generalized suffix trees. The computation of all pairwise overlaps is an important subproblem whose solution is used in some approaches for DNA sequencing. We will discuss this issue in greater detail in Chapter 8.

The naive computation of the overlap of two strings of lengths n and m, as used in Algorithm 4.2 for computing the transition function of a string matching automaton, needs time in $O((\min\{n, m\})^2)$. For the computation of all pairwise overlaps of N strings of approximately the same size $\frac{n}{N}$, this approach would give us an algorithm with running time in $O(N^2 \cdot (\frac{n}{N})^2) = O(n^2)$, where n is the total length of all given strings. Using generalized suffix

Algorithm 4.11 Computation of all pairwise overlaps

Input: A sequence $t_1, \ldots, t_N$ of N strings over an alphabet Σ.

1. Construct the generalized suffix tree T for $t_1, \ldots, t_N$ with root r.
2. Label each inner vertex x of the suffix tree with a subset $L(x) \subseteq \{1, \ldots, N\}$, such that $i \in L(x)$ holds if and only if x is incident to an edge with label $\$_i$.
3. **for** $j := 1$ **to** N **do**
   $x := r$
   while x is no leaf **do**
   for all $i \in L(x)$ **do**
   if $depth(x) < \min\{|t_i|, |t_j|\}$ **then**
   $u(t_i, t_j) := depth(x)$
   $U(t_i, t_j) := pathlabel(x)$
   $x :=$ child of x on the path with label $t_j\$_j$

Output: The overlaps $Ov(t_i, t_j) = U(t_i, t_j)$ and their lengths $ov(t_i, t_j) = u(t_i, t_j)$, for all $i, j \in \{1, \ldots, N\}$.

---

trees, the time complexity can be reduced significantly, as we will show with the following Algorithm 4.11, which solves this task in $O(n \cdot (\log n + |\Sigma| + N))$ time.

Before proving the correctness of Algorithm 4.11 and estimating its running time, we will illustrate its work with an example.

Example 4.6. We want to compute the pairwise overlaps of the strings $t_1 = aba$, $t_2 = bab$, and $t_3 = aabb$. According to Algorithm 4.11, we first construct a generalized suffix tree for these strings. This suffix tree is shown in Figure 4.10. To enhance readability, the labels of the leaves are not shown since they are not used by this algorithm. For every inner vertex x, the set $L(x)$ and the string depth $depth(x)$ are annotated; the latter is shown boxed.

We will now show, as an example, how the algorithm computes the overlap of t_2 with t_1 in Step 3. In this case, $j = 1$ and $i = 2$. The algorithm starts in the root r of the suffix tree. Since r is not a leaf, i is contained in $L(r)$, and $depth(r) = 0 < 3 = \min\{|t_2|, |t_1|\}$ holds; it defines $u(t_2, t_1) = 0$ und $U(t_2, t_1) = \lambda$. Then the algorithm proceeds to the next vertex on the path with label t_1, in this example, vertex y. Since $2 \notin L(y)$, u and U are not modified here, but the algorithm proceeds to vertex z, which is the next vertex on the path labeled t_1. The index 2 is contained in $L(z)$, and $depth(z) = 2 < 3 = \min\{|t_2|, |t_1|\}$ holds; thus, $u(t_2, t_1) = depth(z) = 2$ and $U(t_2, t_1) = pathlabel(z) = ab$ are defined. The next vertex visited by the algorithm is already a leaf; hence the algorithm outputs the correct result $ov(t_2, t_1) = u(t_2, t_1) = 2$ and $Ov(t_2, t_1) = U(t_2, t_1) = ab$. ◊

Theorem 4.7. *Let $t_1, \ldots, t_N$ be N strings, $N \in \mathbb{N} - \{0\}$, of total length n over an alphabet Σ. Then Algorithm 4.11 computes all pairwise overlaps in time $O(n \cdot (\log n + |\Sigma| + N))$.*

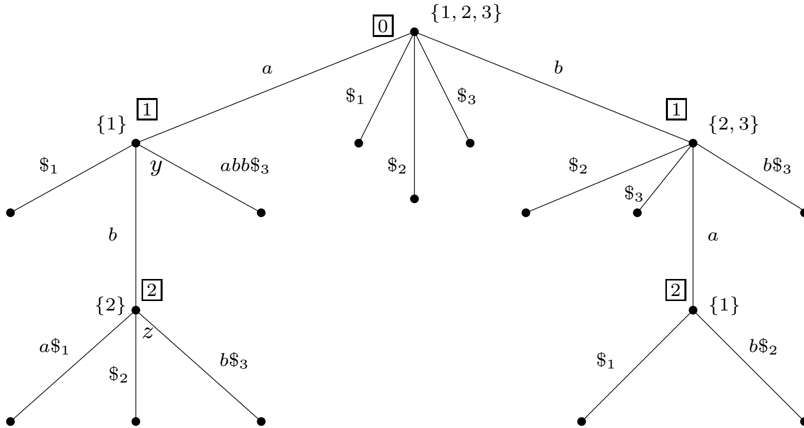


Fig. 4.10. The generalized suffix tree for computing all pairwise overlaps of the strings *aba*, *bab*, and *aabb* using Algorithm 4.11

Proof. We will first prove the correctness of Algorithm 4.11. If $i \in L(x)$ for an inner vertex x of the suffix tree and $i \in \{1, \ldots, N\}$, then the label $pathlabel(x)$ of the path from the root to x is a suffix of t_i. This holds since the vertex x is incident to an edge labeled with the separator symbol $\$_i$ according to the definition of $L(x)$, and is thus adjacent to a leaf labeled (i, k) for some k. Since $pathlabel(x)$ is also a prefix of t_j, if x lies on the path labeled $t_j\$_j$, $pathlabel(x)$ denotes a (not necessarily maximal) overlap of t_i and t_j. On the other hand, following the construction of a suffix tree, all possible overlaps of t_i and t_j can be described in this way. This implies that the algorithm determines the correct overlap, since it computes the overlap of t_i with t_j as $pathlabel(x)$ for the deepest vertex x on the path labeled by $t_j\$_j$, for which $i \in L(x)$ holds and for which the length $depth(x)$ of the path label on the path from the root to x is still smaller as the lengths of t_i and t_j. The latter condition guarantees that neither t_i is a substring of t_j nor vice versa. Please note that initializing the values $u(t_i, t_j)$ and $U(t_i, t_j)$ is not necessary since for each j the algorithm starts in the root r, for which $L(r) = \{1, \ldots, N\}$ holds, and hence the overlap is initialized with λ at this stage of the algorithm..

Now we will analyze the time complexity of the algorithm. The construction of the generalized suffix tree is again possible in $O(n \log n)$ time. The calculation of the sets $L(x)$ needs time in $O(N + |\Sigma|)$ for each vertex, summing up to $O(n \cdot (N + |\Sigma|))$ in total. Thus, it remains for us to analyze the time complexity of step 3. Since the total length of all given strings is n, the total length of all explored paths is in $O(n)$. Furthermore, the total cardinality of all sets $L(x)$ is also in $O(n)$, since every occurrence of an index in a set $L(x)$ corresponds to a certain suffix of t_i. Thus, in Step 3 at most $O(n)$ values are changed overall. The time for finding the correct paths also depends on the number of children of the visited vertices. Every vertex can have at most $|\Sigma| + N$ chil-

dren; this corresponds to the size of the alphabet, extended by the separator symbols. This means that the computation of the correct paths needs overall time in $O(n \cdot (|\Sigma| + N))$. For the complete algorithm, this results in a running time in $O(n \log n + n \cdot (N + |\Sigma|) + n + n \cdot (|\Sigma| + N)) = O(n \cdot (\log n + |\Sigma| + N))$. □

4.5.4 Repeats in Strings

The search for repeated substrings, *repeats* for short, in a given DNA sequence is another interesting biological task. Knowing these repeats can be helpful for designing improved models or algorithms in many biological applications. We distinguish between *exact repeats* , where the same substring occurs several times in the given sequence, and *approximate repeats* , where very similar, but not necessarily the same, substrings occur several times. Furthermore, we distinguish overlapping and disjoint (i.e., nonoverlapping) repeats.

Exact repeats, where overlaps are allowed, can be efficiently determined using suffix trees as we will show in the following. Since, with any string x, also every substring of x is an exact repeat, we are mainly interested in *maximal repeats* that cannot be expanded at their ends. We will start with the formal definition of exact repeats.

Definition 4.12. *Let* $t = t_1 \dots t_n \in \Sigma^n$ *and* $p = p_1 \dots p_m \in \Sigma^m$ *be strings with* $0 < m \leqslant n$. *Let* $t_0 = t_{n+1} = \$ \notin \Sigma$. *Then* p *is an* exact repeat *in* t *if there exist* i, j *with* $0 \leqslant i, j \leqslant n-1$, $i \neq j$, *and* $p = t_{i+1} \dots t_{i+m} = t_{j+1} \dots t_{j+m}$. *If, furthermore,* $t_i \neq t_j$ *and* $t_{i+m+1} \neq t_{j+m+1}$ *holds,* p *is called a* maximal exact repeat *in* t.

In the remainder of this section we only consider exact repeats, and we will call them repeats for brevity. First, we formally define the problem of finding maximal repeats.

Definition 4.13. *The problem of determining all maximal exact repeats (repeat search problem) is the following computing problem:*

Input: A string $t = t_1 \dots t_n$ *over an alphabet* Σ.
Output: The set R *of all maximal exact repeats in* t.

Again, we will utilize the concept of a suffix tree for this task. The following lemma shows the connection between maximal repeats for a string t and the suffix tree for it.

Lemma 4.7. *Let* t *be a string,* T *be the compact suffix tree for* t, *and* p *be a maximal repeat in* t. *Then there exists an inner vertex* x *in* T *with* $p = pathlabel(x)$.

Proof. Let $p = p_1 \dots p_m$. Since p is a maximal repeat, there are two different positions i and j in t, such that $p = t_{i+1} \dots t_{i+m} = t_{j+1} \dots t_{j+m}$ and $t_{i+m+1} \neq t_{j+m+1}$ hold. Hence there are two different substrings in t starting with p; thus,

the vertex y with $p = pathlabel(y)$ has to have at least two children in a simple suffix tree for t. This implies that the compact suffix tree T for t contains an inner vertex x with $p = pathlabel(x)$. □

Lemma 4.7 implies the following claim about the possible number of maximal repeats.

Corollary 4.1. *Let $t = t_1 \dots t_n \in \Sigma^n$ be a string. Then there are at most $n-1$ maximal repeats in t.*

Proof. The compact suffix tree T for t has n leaves and thus at most $n-1$ inner vertices. □

Note that it is nevertheless possible that several different maximal repeats in a string t share the same starting position: In the string $t = aabcbabacabcc$, for example, both ab (due to $a\boldsymbol{ab}cb\boldsymbol{ab}acabcc$) and abc (due to $a\boldsymbol{abc}babac\boldsymbol{abc}c$) are maximal repeats.

In the following, we present a method for finding those inner vertices of a suffix tree whose path labels correspond to maximal repeats. For this, we will need the following definition.

Definition 4.14. *Let $t = t_1 \dots t_n \in \Sigma^n$ be a string and let $2 \leqslant i \leqslant n$. Then t_{i-1} is called the* left-symbol *of i.*

Let T be a suffix tree for t and let x be a leaf of T. Then the left-symbol of *x is defined as the left-symbol of the starting position of the suffix $pathlabel(x)$. Let v be an inner vertex of T. Then v is called* left-diverse *if the subtree rooted at v contains two leaves with different left-symbols.*

Note that, together with a vertex v, all of its ancestors in the suffix tree are left-diverse. The property of left-diversity now gives us a criterion for finding the maximal repeats.

Theorem 4.8. *Let $t = t_1 \dots t_n \in \Sigma^n$ be a string and let T be a compact suffix tree for t with root r. Then a string $p \in \Sigma^*$ is a maximal repeat in t if and only if there exists a left-diverse vertex $x \neq r$ in T with $p = pathlabel(x)$.*

Proof. We show first that the path label of a left-diverse vertex always is a maximal repeat. Let x be a left-diverse vertex with path label p. Then there exist two leaves y_1 with left-symbol a and y_2 with left-symbol b in the subtree rooted at x, such that $a \neq b$ holds.

If the paths from x to y_1 and y_2 are disjoint, as shown in Figure 4.11 (a), then there are two substrings $apcw_1$ and $bpdw_2$ of t, where $c, d \in \Sigma$ and $c \neq d$ hold. This implies that p is a maximal repeat in this case.

If the paths from x to y_1 and y_2 are not disjoint, i.e., if the labels of both paths start with the same symbol $c \in \Sigma$, then there exists a further path from x to another leaf y_3 with left-symbol e, whose label starts with a symbol $d \neq c$, since x is an inner vertex of a compact suffix tree. This situation is

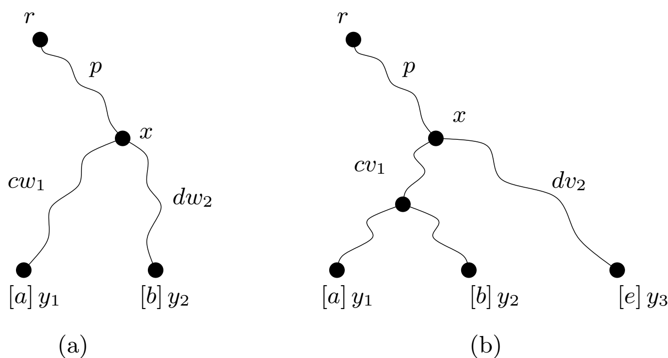


Fig. 4.11. The two possible situations in the proof of Theorem 4.8. The left-symbols of the leaves are shown in brackets

shown in Figure 4.11 (b). Also in this case p is a maximal repeat in t because of the substrings epd and apc occurring in t if $e \neq a$ holds, or because of the substrings epd and bpc if $e \neq b$ holds.

It remains for us to show that a maximal repeat p always corresponds to the path label of a left-diverse vertex. From Lemma 4.7 we know that there exists a vertex x in T with $pathlabel(x) = p$. But the vertex x also has to be left-diverse, since there have to exist two starting positions of p in t with different left-symbols. □

Since the property of left-diversity is passed from one vertex to all its successors, the left-diverse vertices of a compact suffix tree can be computed in linear time, starting from the parents of the leaves. For this computation we just have to store in each inner vertex x either the unique left-symbol in all leaves of the corresponding subtree or the fact that x is left-diverse. We can then deduce the left-diversity of a vertex from information about its children. Thus, the repeat search problem can be solved efficiently according to Theorem 4.8. Furthermore, Theorem 4.8 suggests a compact representation of the set of all maximal repeats. If we delete vertices from the suffix tree for the given string that are not left-diverse, we get a subtree where the path labels are in one-to-one correspondence with the maximal repeats. For a given string of length n, this representation is of size $O(n \log n)$, although the total length of all maximal repeats can be of order $\Omega(n^2)$.

4.6 Suffix Arrays

In this section, we consider another very useful data structure for dealing with strings, the so-called suffix array. The suffix array for a given string is a linear array containing all suffixes of the string in lexicographical order. Similarly to suffix trees, suffix arrays can also be used for efficiently solving the string

matching problem, as well as for solving several other string problems. For convenience, we consider the uniform cost measure for measuring the time complexity of our algorithms throughout this section; for example, we will assume that any comparison of numbers can be performed within constant time.[3]

We start with the formal definition of the lexicographical order of strings.

Definition 4.15. *Let $\Sigma = \{a_1, \ldots, a_k\}$ be an alphabet with a linear ordering $a_1 \prec \ldots \prec a_k$ on its symbols. Let $s = s_1 \ldots s_n$ and $t = t_1 \ldots t_m$ be two strings over Σ. Then, s is* lexicographically smaller *than t, written as $s \prec_{\text{lex}} t$, if there exists an index $j \leqslant \min\{m, n\}$ such that $s_1 \ldots s_{j-1} = t_1 \ldots t_{j-1}$ and either $s_j \prec t_j$ or $j = n < m$.*

A sequence of strings $p_1, \ldots, p_l$ over Σ is said to be in lexicographical order*, if $p_1 \prec_{\text{lex}} \ldots \prec_{\text{lex}} p_l$ holds.*

We are now ready to define the suffix array for a given string. We assume for the remainder of this section that all alphabets used are ordered; thus, the lexicographical order of the considered strings is always well defined.

Definition 4.16. *Let $s = s_1 \ldots s_n$ be a string over an ordered alphabet. The* suffix array *for s is an array $\mathcal{A}(s) = (j_1, \ldots, j_n)$ of indices such that $s[j_1, n] \prec_{\text{lex}} s[j_2, n] \prec_{\text{lex}} \ldots \prec_{\text{lex}} s[j_n, n]$ holds.*[4]

We illustrate this definition with an example.

Example 4.7. Consider the string $s = ababbabbb$. If we assume $a \prec b$, this implies the following lexicographical order of the suffixes of s:

$$\begin{aligned} s[1,9] &= ababbabbb \\ s[3,9] &= abbabbb \\ s[6,9] &= abbb \\ s[9,9] &= b \\ s[2,9] &= babbabbb \\ s[5,9] &= babbb \\ s[8,9] &= bb \\ s[4,9] &= bbabbb \\ s[7,9] &= bbb \end{aligned}$$

From this order we can directly read out the suffix array

$$\mathcal{A}(s) = (1, 3, 6, 9, 2, 5, 8, 4, 7)$$

for the string s. ◊

[3] Using the logarithmic cost measure and thus taking into account the actual length of the numbers involved, as we did it in the previous sections for the suffix tree algorithms, would result in an increase in the running time by a logarithmic factor.

[4] Please recall that $s[i, j]$ denotes the substring $s_i \ldots s_j$ of s according to Definition 3.4.

Algorithm 4.12 String matching using a suffix array

Input: A text t and a pattern p over an ordered alphabet Σ, and a suffix array $\mathcal{A}(t)$ for the text t.

1. Use binary search to find the first position i and the last position j in the suffix array such that p starts as a substring in t at positions $\mathcal{A}(t)[i]$ and $\mathcal{A}(t)[j]$.
2. $I := \{\mathcal{A}(t)[i], \mathcal{A}(t)[i+1], \ldots, \mathcal{A}(t)[j]\}$.

Output: The set I of all positions, where p starts as a substring in t.

If we are given a suffix array for a text t, we can very efficiently solve the string matching problem for t and any pattern p. Due to the lexicographical ordering of the suffixes, all suffixes starting with the pattern p appear consecutively within the suffix array. Thus, the string matching problem can be solved very efficiently using binary search in the suffix array as shown in detail in Algorithm 4.12.

Theorem 4.9. *Algorithm 4.12 solves the string matching problem for a text* $t = t_1 \ldots t_n$ *and a pattern* $p = p_1 \ldots p_m$ *over an ordered alphabet* Σ *in* $O(m \log n + k)$ *time, where* k *is the number of occurrences of* p *in* t.

Proof. It is immediate from the definition of a suffix array that the algorithm solves the string matching problem. We will now analyze its time complexity. The binary search needs $O(\log n)$ steps; in each step the pattern has to be compared to the respective substring of t given by the suffix array. Each of these comparisons needs at most $O(m)$ time. Thus, Step 1 of the algorithm can be implemented in $O(m \log n)$ time. Outputting all k positions obviously needs time in $O(k)$. □

Theorem 4.9 implies that the string matching problem can be solved very efficiently once we are given a suffix array of the text. In the following we present an efficient algorithm for constructing a suffix array for a given string. We start with describing a special sorting strategy for integer values, the radix sort algorithm, which we need as a subprocedure in our algorithm. The radix sort algorithm is a stable sorting algorithm in the sense of the following definition.

Definition 4.17. *Let* A *be some sorting algorithm for integer values.* A *is called* stable*, if any two items with the same value appear in the same order in the sorted output as in the input.*

We first describe a well-known sorting strategy for efficiently sorting an array of integers from a limited range of values. This sorting method is called *counting sort* (or *bucket sort*), and it is based on counting the number of occurrences in the input for each element of the value range. This method is shown in detail in Algorithm 4.13.

Algorithm 4.13 Counting Sort

Input: An array $A = (A(1), \ldots, A(n))$ of integers from the range $\{0, \ldots, k\}$.

1. Counter initialization:
   for $i := 0$ **to** k **do**
     $c(i) := 0$
2. Count the number of elements of each type:
   for $j := 1$ **to** n **do**
     $c(A(j)) := c(A(j)) + 1$
3. Count the number of elements less or equal to i:
   for $i := 1$ **to** k **do**
     $c(i) := c(i) + c(i-1)$
4. Calculate the position of each element in the sorted array:
   for $j := n$ **downto** 1 **do**
     $B(c(A(j))) := A(j)$
     $c(A(j)) := c(A(j)) - 1$

Output: The sorted array $B = (B(1), \ldots, B(n))$.

Lemma 4.8. *Algorithm 4.13 is a stable algorithm for sorting an array of n integers from the range $\{0, \ldots, k\}$ with running time in $O(n + k)$.*

Proof. Obviously, Algorithm 4.13 correctly sorts the given input array in time $O(n + k)$. We now show that it also is a stable sorting algorithm. After Step 3, the counter $c(i)$ contains the last position that will contain value i in the sorted array, for all $i \in \{0, \ldots, k\}$. In Step 4, the elements from A are inserted into their respective last possible positions in B in descending order, i.e., from $A(n)$ to $A(1)$; thus the order of elements with the same value is not changed. □

We now use Algorithm 4.13 as a subprocedure in our radix sort algorithm for sorting arbitrary integers. We assume that the input numbers are given in k-ary notation for some k, for example, in decimal representation. The idea behind the radix sort algorithm is to sort the numbers according to each digit of their representation, starting with the least significant digit. This method is shown in detail in Algorithm 4.14. We assume that the index 1 corresponds to the least significant digit.

Theorem 4.10. *Algorithm 4.14 sorts a given array $A = (a_1, \ldots, a_n)$ of integers of length d in k-ary representation in time $O((n + k) \cdot d)$.*

Proof. The time complexity of Algorithm 4.14 is a direct consequence of the time complexity of Algorithm 4.13. We now show that Algorithm 4.14 correctly sorts its input by using induction on the length d of its input numbers. Obviously, the algorithm gives the correct result for $d = 1$. For numbers of

Algorithm 4.14 Radix Sort

Input: An array $A = (a_1, \ldots, a_n)$ of integers of length d in k-ary representation.

for $i := 1$ **to** d **do**
  Sort A according to the i-th digit using Algorithm 4.13.

Output: The sorted array.

---

length d, the algorithm ignores the most significant digit during the first $d-1$ iterations of the for loop and thus sorts the numbers correctly according to the $d-1$ least significant digits due to the induction hypothesis. In the last iteration, the numbers are sorted according to the most significant digit using Algorithm 4.13, which is a stable sorting algorithm according to Lemma 4.8. This stability implies that the numbers are correctly sorted after the last iteration. □

We are now ready to present an efficient algorithm for constructing a suffix array. It is called *skew algorithm*, and it is based on the following idea: For a given string s, the algorithm first recursively constructs a suffix array $\mathcal{A}^{1,2} = \mathcal{A}^{1,2}(s)$ for two thirds of all suffixes, more precisely, for all suffixes with starting positions $i \not\equiv 0 \pmod 3$. With the help of this array $\mathcal{A}^{1,2}$, it then constructs a second suffix array $\mathcal{A}^0 = \mathcal{A}^0(s)$ for all suffixes with starting positions $i \equiv 0 \pmod 3$. In the last step, it merges these two arrays into one.

The main idea behind this algorithm is a clever implementation of the merging step. For merging the two constructed arrays, we have to succesively compare the elements from both arrays to each other. The idea is to compare a suffix $s[j,n]$ with $j \equiv 0 \pmod 3$ to a suffix $s[i,n]$ with $i \equiv 1 \pmod 3$ by comparing the pairs $(s_j, s[j+1,n])$ and $(s_i, s[i+1,n])$. This pair representation enables us to perform the comparison in constant time since both $j+1 \not\equiv 0 \pmod 3$ and $i+1 \not\equiv 0 \pmod 3$, and thus their relative order is already given by $\mathcal{A}^{1,2}$. Analogously, a suffix $s[j,n]$ with $j \equiv 0 \pmod 3$ and a suffix $s[i,n]$ with $i \equiv 2 \pmod 3$ can be compared in constant time by comparing the triples $(s_j, s_{j+1}, s[j+2,n])$ and $(s_i, s_{i+1}, s[i+2,n])$.

The skew algorithm is presented in more detail in Algorithm 4.15. For analyzing its correctness and running time, we first need a more detailed description of Step 4 of the algorithm. Merging two sorted arrays can be easily done in linear time[5] if it is possible to compare two elements in constant time. We now show that, after a linear-time preprocessing step, we are indeed able to compare an element from $\mathcal{A}^{1,2}$ to an element from $\mathcal{A}^0$ in constant time. For this, we first need the following definition of an inverse suffix array.

---

[5] We can use the merge phase of the well-known mergesort algorithm which is described for instance in Chapter 2.3.1 of [51].

Algorithm 4.15 Skew algorithm for constructing a suffix array

Input: A string $s = s_1 \dots s_n$ over the ordered alphabet $\{1, \dots, n\}$.

1. Initialization:
   a) Define $s_{n+1} := 0$, $s_{n+2} := 0$, and $s_{n+3} := 0$, and let s denote the string $s_1 \dots s_{n+3}$.
   b) Let $S_i := s_i \dots s_n$ be the i-th suffix of s for all $0 \leqslant i \leqslant n+1$.
   c) Let $k_i = \max\{k \leqslant n+1 \mid k \equiv i \pmod 3\}$, for $i \in \{1, 2\}$;
   let $l_i := |\{k \leqslant n+1 \mid k \equiv i \pmod 3\}|$, for $i \in \{1, 2\}$.
2. Construct the suffix array $\mathcal{A}^{1,2}$ for the set $\mathcal{S}^{1,2}$ of all suffixes S_i such that $i \not\equiv 0 \pmod 3$:
   a) Let $t_i := s[i, i+2]$ for all $0 \leqslant i \leqslant n+1$ such that $i \not\equiv 0 \pmod 3$.
   b) Sort the triples t_i, for all $i \leqslant \max\{k_1, k_2\}, i \not\equiv 0 \pmod 3$, using radix sort (Algorithm 4.14).
   c) Assign lexicographical names $t'_i \in \{1, \dots, \lceil \frac{2}{3}(n+1) \rceil\}$ to the triples, i.e., define the t'_i such that $t'_i = t'_j$ if $t_i = t_j$, and $t'_i < t'_j$ if $t_i \prec_{\text{lex}} t_j$.
   d) If all t'_i are distinct, construct $\mathcal{A}^{1,2}$ directly from the order of the t_i; else, recursively compute the suffix array $\tilde{\mathcal{A}}$ for the string

   $$\tilde{s} = \tilde{s}_1 \dots \tilde{s}_{l_1+l_2} := t'_1 t'_4 t'_7 \dots t'_{k_1} \cdot t'_2 t'_5 t'_8 \dots t'_{k_2}$$

   and construct $\mathcal{A}^{1,2}$ from $\tilde{\mathcal{A}}$ by substituting the indices of the t'_i for the indices of the $\tilde{s}_j$.
3. Construct the suffix array $\mathcal{A}^0$ for the set $\mathcal{S}^0$ of all suffixes S_i such that $i \equiv 0 \pmod 3$:
   a) Represent S_i by the pair (s_i, S_{i+1}) for all $1 \leqslant i \leqslant n$ such that $i \equiv 0 \pmod 3$.
   b) Consider the order of $\mathcal{S}^0$ as given by the order of the second component of its elements in $\mathcal{A}^{1,2}$, and sort $\mathcal{S}^0$ by counting sort (Algorithm 4.13) with respect to the first components.
4. Merge the two suffix arrays $\mathcal{A}^{1,2}$ and $\mathcal{A}^0$ into a suffix array $\mathcal{A}(s)$.

Output: The constructed suffix array $\mathcal{A}(s)$.

Definition 4.18. *Let $\mathcal{A}$ be a suffix array for a string $s = s_1 \dots s_n$. The* inverse suffix array *for s is an array $\overline{\mathcal{A}}$ such that $\overline{\mathcal{A}}(i) = j$ if and only if $\mathcal{A}(j) = i$, for all $1 \leqslant i \leqslant n$.*

Intuitively speaking, the j-th entry $\overline{\mathcal{A}}(j)$ of the inverse suffix array describes the position of element j in the suffix array $\mathcal{A}$. Thus, the inverse suffix array enables us to determine the relative order of two given elements within a given suffix array in constant time. Obviously, the inverse suffix array can be constructed from the suffix array in linear time. The following lemma shows that it can be used for comparing elements from $\mathcal{A}^0$ and $\mathcal{A}^{1,2}$ in constant time.

Lemma 4.9. *Let $s = s_1 \dots s_n$ be a string over the ordered alphabet $\{1, \dots, n\}$ and let $\mathcal{A}^{1,2}$ and $\mathcal{A}^0$ be the suffix arrays constructed in steps 2 and 3 of Algorithm 4.15. If the inverse suffix array $\overline{\mathcal{A}}^{1,2}$ corresponding to $\mathcal{A}^{1,2}$ is given,*

then any suffix corresponding to an element from $\mathcal{A}^0$ can be compared to any suffix corresponding to an element from $\mathcal{A}^{1,2}$ in constant time.

Proof. For the proof we distinguish two cases. To compare a suffix S_j with $j \equiv 0 \pmod 3$ to a suffix S_i with $i \equiv 1 \pmod 3$, we represent S_j by the pair (s_j, S_{j+1}) and S_i by the pair (s_i, S_{i+1}). Both S_{j+1} and S_{i+1} correspond to entries from $\mathcal{A}^{1,2}$, since $j + 1 \equiv 1 \pmod 3$ and $i + 1 \equiv 2 \pmod 3$. Their relative order is given by their positions in $\mathcal{A}^{1,2}$, which can be determined in constant time from $\overline{\mathcal{A}}^{1,2}$.

To compare a suffix S_j with $j \equiv 0 \pmod 3$ from $\mathcal{A}^0$ to a suffix S_i with $i \equiv 2 \pmod 3$, we represent S_j by the triple (s_j, s_{j+1}, S_{j+2}) and S_i by the triple (s_i, s_{i+1}, S_{i+2}). In this case, both S_{j+2} and S_{i+2} correspond to entries from $\mathcal{A}^{1,2}$, since $j+2 \equiv 2 \pmod 3$ and $i+2 \equiv 1 \pmod 3$. Their relative order is thus given by their positions in $\mathcal{A}^{1,2}$, and can again be read from $\overline{\mathcal{A}}^{1,2}$ in constant time. □

Before we proceed with proving the correctness of the skew algorithm and analyzing its running time, we illustrate its work on two examples. We start with a simple example which does not need any recursion.

Example 4.8. We consider the string $s = 12123221$. The second step of the algorithm constructs a suffix array for all suffixes starting at positions $i \not\equiv 0 \pmod 3$. The following table shows the suffixes, the corresponding triples t_i, and their lexicographical names t'_i as constructed in Step 2 of the algorithm.

i	s_i	S_i	t_i	t'_i
1	1	12123221	121	2
4	2	23221	232	5
7	2	21	210	3
2	2	2123221	212	4
5	3	3221	322	6
8	1	1	100	1

Since in this example all t_i are different, all t'_i are also different, and the algorithm can directly construct the suffix array $\mathcal{A}^{1,2} = (8, 1, 7, 2, 4, 5)$.

In Step 3, the algorithm constructs a suffix array for all suffixes starting at positions $i \equiv 0 \pmod 3$. These suffixes and their pair representations as used by the algorithm are shown in the following table.

i	s_i	S_i	(s_i, S_{i+1})
3	1	123221	$(1, S_4)$
6	2	221	$(2, S_7)$

The suffixes S_4 and S_7 appear in $\mathcal{A}^{1,2}$ in the order S_7, S_4; the algorithm therefore stably sorts the sequence $(2, S_7), (1, S_4)$ with respect to the first component, and thus computes the suffix array $\mathcal{A}^0 = (3, 6)$ in Step 3.

In Step 4, the algorithm merges the two suffix arrays $\mathcal{A}^{1,2} = (8, 1, 7, 2, 4, 5)$ and $\mathcal{A}^0 = (3, 6)$. The pair and triple representations of the suffixes used for the comparisons are shown in the following table.

i	(s_i, S_{i+1})	(s_i, s_{i+1}, S_{i+2})
3	$(1, S_4)$	$(1, 2, S_5)$
6	$(2, S_7)$	$(2, 2, S_8)$
1	$(1, S_2)$	
4	$(2, S_5)$	
7	$(2, S_8)$	
2		$(2, 1, S_4)$
5		$(3, 2, S_7)$
8		$(1, 0, S_{10})$

The first comparison performed by the algorithm is between S_8 and S_3. Since $8 \equiv 2 \pmod 3$, the triple representation is used for this comparison. $S_8 = (1, 0, S_{10}) \prec_{\text{lex}} (1, 2, S_5) = S_3$ holds due to the second component of the triples. Thus, the suffix array starts with $\mathcal{A}(1) = 8$.

Secondly, the algorithm compares S_1 and S_3. Since $1 \equiv 1 \pmod 3$, the pair representation of the suffixes is used for this comparison. $S_1 = (1, S_2) \prec_{\text{lex}} (1, S_4) = S_3$ holds since the index 2 occurs before the index 4 in $\mathcal{A}^{1,2}$. Thus, $\mathcal{A}(2) = 1$.

After another four comparisons of this type, all elements of $\mathcal{A}^0$ are inserted into the constructed suffix array $\mathcal{A}$, and the algorithm returns $\mathcal{A} = (8, 1, 3, 7, 2, 6, 4, 5)$. ♢

With the next example, we will illustrate the recursive structure of the skew algorithm.

Example 4.9. We consider the string $s = 1221221$. The triples t_i and their lexicographical names t'_i for all $i \not\equiv 0 \pmod 3$, as constructed in the second step of the algorithm, are shown in the following table.

i	t_i	t'_i
1	122	2
4	122	2
7	100	1
2	221	3
5	221	3

In this example, the lexicographical names t'_i are not pairwise distinct; to obtain the suffix array $\mathcal{A}^{1,2}$, the skew algorithm is recursively started on the string

$$\tilde{s} = 22133.$$

The following table shows the correspondence between positions in $\tilde{s}$ and s.

i	j	$t'_i = \tilde{s}_j$
1	1	2
4	2	2
7	3	1
2	4	3
5	5	3

The algorithm now recursively constructs the suffix array $\tilde{\mathcal{A}} = (3, 2, 1, 5, 4)$ for $\tilde{s}$. Converting the indices according to the above table yields the partial suffix array $\mathcal{A}^{1,2} = (7, 4, 1, 5, 2)$ for s. In Step 3, the algorithm computes the array $\mathcal{A}^0 = (6, 3)$, and in step 4, these two arrays are merged into the suffix array $\mathcal{A} = (7, 4, 1, 6, 3, 5, 2)$. $\diamondsuit$

We are now ready for analyzing the skew algorithm.

Theorem 4.11. *Let $s = s_1 \dots s_n$ be a string over the ordered alphabet $\{1, \dots, n\}$. Then Algorithm 4.15 constructs a suffix array $\mathcal{A}(s)$ for s in time $O(n)$.*

Proof. We start with proving the correctness of Algorithm 4.15. Step 1 of the algorithm contains some initializations; the correctness of steps 3 and 4 is obvious. The crucial part is to prove the correctness of step 2. If all considered triples t_i are distinct, the corresponding suffixes are also distinct, and the lexicographical names t'_i obviously imply the order of these suffixes, and thus the suffix array $\mathcal{A}^{1,2}$. But if some of the triples, and hence also some of the lexicographical names, are identical, we have to show that the recursively computed suffix array $\tilde{\mathcal{A}}$ for the string $\tilde{s}$ is, after renaming the indices, equal to the array $\mathcal{A}^{1,2}$ for the string s.

To see this, we first observe that $t_i t_{i+3} \dots t_{k_1} = S_i$ for all $i \equiv 1 \pmod 3$, and $t_i t_{i+3} \dots t_{k_2} = S_i$ for all $i \equiv 2 \pmod 3$. This implies that $t'_i t'_{i+3} \dots t'_{k_1} = \tilde{s}[\frac{i+2}{3}, l_1]$ represents the suffix S_i for all $i \equiv 1 \pmod 3$ and $t'_i t'_{i+3} \dots t'_{k_2} = \tilde{s}[l_1 + \frac{i+1}{3}, l_2]$ represents the suffix S_i for all $i \equiv 2 \pmod 3$.

Concatenating $t'_1 \dots t'_{k_1}$ and $t'_2 \dots t'_{k_2}$ does not cause any problem if we can prove that $\tilde{s}_{l_1} = t'_{k_1}$ is always unique, i.e., distinct from all other t'_i, since in this case comparing a suffix of $\tilde{s}$ starting at position $j \leqslant l_1$ to another suffix x of $\tilde{s}$ always yields the same result as comparing $\tilde{s}[j, l_1]$ to x. It remains for us to show that $\tilde{s}_{l_1}$ is always unique.

Obviously, t_{k_1} and t_{k_2} are the only triples that may contain the symbol 0, and the number of zeros in t_{k_1} and t_{k_2} always has to be different. Since we defined $k_1 = \max\{k \leqslant n + 1 \mid k \equiv 1 \pmod 3\}$, we have guaranteed that t_{k_1} contains at least one zero in any case (more precisely, it contains one zero if $n \equiv 1 \pmod 3$), two zeros if $n \equiv 2 \pmod 3$, and three zeros if $n \equiv 0 \pmod 3$). Thus, the triple t_{k_1}, and also its lexicographical naming t'_{k_1}, have to be unique. This proves the correctness of the recursion in step 2, and thus of the skew algorithm.

We will now analyze the running time of the algorithm. Obviously, the initialization in step 1 can be done in linear time. Determining the triples and assigning the lexicographical names in step 2 can obviously be implemented in $O(n)$ time, the same time bound follows for the radix sorting from Theorem 4.10. Thus, step 2 without the recursive call can be executed in linear time. Step 3 is also implementable in linear time; finding the order of the elements from $\mathcal{S}^0$ requires one pass through the array $\mathcal{A}^{1,2}$, and the counting sort is possible in linear time according to Lemma 4.8. The linear running time of step 4 follows from Lemma 4.9.

With every recursive call, the length of the string is reduced by a factor of roughly $\frac{2}{3}$; thus, the overall running time $T(n)$ of the skew algorithm on a string of length n satisfies the recurrence $T(n) = T(\lceil\frac{2n}{3}\rceil) + O(n)$. It can be fairly easily seen that this recurrence has the solution $T(n) = O(n)$. We will not give a detailed proof here.[6] □

4.7 Summary

The string matching problem is a basic string problem that consists of finding all positions in a given text where a given pattern starts as a substring. This problem can be solved with a naive approach by sliding the pattern along the text and testing for equality at each position. This naive algorithm takes time in $O(m \cdot (n-m))$, where n is the length of the text and m is the length of the pattern.

More efficient algorithms for the string matching problem are based on preprocessing either the pattern or the text. One possible approach is based on the concept of finite automata. When constructing a string matching automaton in a clever way, the string matching problem can be solved using this automaton in $O(n + m \cdot |\Sigma|)$ time, where Σ is the alphabet over which the text and the pattern are given.

The Boyer–Moore algorithm is another method based on preprocessing the pattern. This algorithm is based on a similar strategy as that of the naive algorithm, but in the case of inequality of the pattern and the text at one position, it can shift the pattern more than one position for the next comparison. The Boyer–Moore algorithm often achieves the best running time in practice although its worst-case time complexity is not better than that of the naive algorithm.

A preprocessing of the text can be done using suffix trees, which constitute a clever representation of all suffixes of the text and allows for efficient solutions for many variants of string matching problems. A suffix tree for a given text of length n can be constructed in $O(n \log n)$ time. After that, the actual

[6] A detailed introduction to solving recurrence equations, also covering the so-called master theorem, from which the solution for this recurrence directly follows, can be found in [51].

string matching with a given pattern of length m is possible in $O(m \cdot |\Sigma| + k)$ time, where k is the number of occurrences of the pattern in the text. Using suffix trees is especially advantageous if one wants to compare many different patterns to the same text.

Furthermore, suffix trees can be used to solve many other string problems efficiently. One example is the substring problem consisting of finding all those texts in a given set of texts containing a given pattern. Further problems which are solvable using suffix trees include the computation of longest common substrings or of all pairwise overlaps of a given set of strings. Also, exact repeats within a given string can be computed efficiently using suffix trees.

Further useful data structures for string processing are suffix arrays. A suffix array is an array describing the lexicographical order of all suffixes of a given string. Suffix arrays allow for an efficient solution of the string matching problem and can be constructed in linear time according to the uniform cost measure.

4.8 Bibliographic Notes

A detailed discussion of various string problems can be found in the textbooks by Gusfield [91] and Crochemore and Rytter [54]. The different approaches for solving the string matching problem based on preprocessing the pattern are discussed in detail in the textbook by Cormen, Leiserson, and Rivest [50].[7]

The usage of finite automata for the string matching problem was introduced in the book by Aho, Hopcroft, and Ullman [3]. A general introduction to the theory of finite automata can be found, for example, in the books by Hromkovič [104] and Hopcroft, Motwani, and Ullman [103]. Knuth, Morris, and Pratt [123] invented another algorithm for the string matching problem which we unfortunately could not present in this book due to space limitations. The Knuth-Morris-Pratt algorithm runs in time $O(n + m)$ for a text of length n and a pattern of length m. The idea underlying the Knuth-Morris-Pratt algorithm can also be used to construct a string matching automaton in linear time. A detailed and illustrative presentation of the Knuth-Morris-Pratt algorithm can be found in [50, 51].

The Boyer–Moore algorithm was presented by Boyer and Moore in [37]. Generalizations of this algorithm with linear worst-case running time are due to Rytter [170] as well as Apostolico and Giancarlo [15].

Simple suffix trees were used for the first time for the string matching problem by Aho, Hopcroft, and Ullman [3]. The first algorithms for constructing a compact suffix tree in linear time are due to Weiner [207] and McCreight [144]. The easiest and most often used linear-time algorithm for constructing a compact suffix tree was designed by Ukkonen [193].

[7] The corresponding chapter in the second edition [51] has been shortened and no longer includes the presentation of the Boyer–Moore algorithm.

Further applications of suffix trees are described in the paper by Apostolico [13], and in the books by Gusfield [91] and Crochemore and Rytter [54].

The concept of suffix arrays was introduced by Manber and Myers [143]. The presented algorithm for linear-time construction of a suffix array is due to Kärkkäinen and Sanders [112]. Abouelhoda, Kurtz, and Ohlebusch [1] presented a method for transferring many suffix tree algorithms to suffix arrays with an additional table containing the longest common prefix of each of two consecutive suffixes. This additional table can also be constructed in linear time [112].

5

Alignment Methods

After having seen some basic algorithms for the exact string matching problem in the preceding chapter, in this chapter we consider problems arising from dealing with erroneous string data. Since the data we are provided with in biological applications are always obtained through experiments, and are thus subject to measurement errors, we will generalize the question from the last chapter: Instead of exactly comparing two strings, we want to know if the given strings coincide *approximately*, or if a given pattern is *approximately* contained as a substring in a given text. To reach this goal we will define different notions of similarity for strings, and we will present algorithms for comparing strings. The idea behind these so-called *alignment methods* is to align the given strings by inserting gaps such that as many positions as possible coincide.

In the literature, the notion of sequences (instead of strings) is mostly used in the context of alignment methods. This is motivated by the fact that in this context subsequences play a more important role than substrings. To improve readability and to avoid using two notions for the same object, we will also use the term strings in this chapter.

Alignment methods have multiple applications in molecular biology, such as the comparison of different sequences of the same gene (from different experiments or labs), or the search for a given string as subsequence in a database of strings. Furthermore, alignment algorithms are often used as subprocedures for solving more complex problems like the computation of approximate overlaps, which are useful for some approaches to DNA sequencing.

The importance of these alignment methods is not based only on the handling of error-prone data. They are, for example, also needed for the comparison of DNA sequences (or protein sequences) of different organisms. The similarity of the sequences determined in this way can then be used as a measure of the degree of relationship of these organisms.

This chapter is organized as follows. In Section 5.1 we will present methods for the alignment of two strings. Section 5.2 is dedicated to some heuristic approaches based on alignments to searching a database. The generalization of the alignment notion to multiple strings can be found in Section 5.3. In

Section 5.4 we summarize the results of this chapter, and in Section 5.5 we make some bibliographic remarks.

5.1 Alignment of Two Strings

In this section we will consider the basic task of computing an optimal alignment of two given strings, i.e., an alignment leading to a maximal similarity of the aligned strings. We will first define the notion of alignment formally, and then describe an efficient algorithm computing an optimal alignment of two strings. After that, we will consider some variants and generalizations of the described method.

5.1.1 Basic Definitions

As described in the introduction to this chapter, we will denote by the *alignment* of strings the result of inserting gaps into the strings such that afterwards as many positions as possible coincide. We now give a formal definition of an alignment.

Definition 5.1. *Let $s = s_1 \ldots s_m$ and $t = t_1 \ldots t_n$ be two strings over an alphabet Σ. Let $\texttt{-} \notin \Sigma$ be a gap symbol and let $\Sigma' = \Sigma \cup \{\texttt{-}\}$. Let $h : (\Sigma')^* \to \Sigma^*$ be a homomorphism defined by $h(a) = a$ for all $a \in \Sigma$, and $h(\texttt{-}) = \lambda$.*

An alignment *of s and t is a pair (s', t') of strings of length $l \geqslant \max\{m, n\}$ over the alphabet Σ', such that the following conditions hold:*

(a) $|s'| = |t'| \geqslant \max\{|s|, |t|\}$,
(b) $h(s') = s$,
(c) $h(t') = t$, *and*
(d) there is no position containing a gap symbol in s' as well as in t', i.e., $s'_i \neq \texttt{-}$ or $t'_i \neq \texttt{-}$ for all $i \in \{1, \ldots, l\}$.

Informally speaking, conditions (b) and (c) in Definition 5.1 state that deleting all gap symbols from s' (or t') yields the string s (or t). We illustrate this definition by the following example.

Example 5.1. Let s = `GACGGATTATG` and let t = `GATCGGAATAG`. A possible alignment of s and t is:

$$s' = \texttt{GA-CGGA}\underline{\texttt{T}}\texttt{TATG}$$
$$t' = \texttt{GATCGGA}\underline{\texttt{A}}\texttt{TA-G}$$

By inserting a gap after the second symbol of s and before the last symbol of t, the resulting strings s' and t' coincide at all positions, except for the third, eighth, and eleventh. ◊

As shown in Example 5.1, the two strings of an alignment are often written one below the other, i.e., we can consider the alignment to be a $(2 \times l)$-matrix. The columns of this matrix are also called the columns of the alignment. We distinguish four types of columns:

Insertion: The first string has a gap symbol in this column.
Deletion: The second string has a gap symbol in this column.
Match: Both strings coincide in this column.
Substitution/Mismatch: Both strings do not coincide in this column, but there is no gap symbol.

This means that an alignment can also be described as a sequence of insertions, deletions, and substitutions applied to the first string in order to yield the second string. Inserting a gap into the second string corresponds to deleting a symbol from the first string; inserting a gap into the first string corresponds to inserting an additional symbol.

We want to use alignments for defining a measure of similarity of strings. We define the score of an alignment as follows.

Definition 5.2. *Let s and t be two strings over an alphabet Σ. Let $p(a, b) \in \mathbb{Q}$ for all $a, b \in \Sigma$, and let $g \in \mathbb{Q}$.*

The score δ *of an alignment (s', t') of length l with $s' = s'_1 \dots s'_l$ and $t' = t'_1 \dots t'_l$ is first defined column-wise: For $x, y \in \Sigma$, let $\delta(x, y) = p(x, y)$. Furthermore, let $\delta(-, y) = \delta(x, -) = g$. The score of an alignment is then defined as the sum of the values over all columns, i.e.,*

$$\delta(s', t') = \sum_{i=1}^{l} \delta(s'_i, t'_i).$$

For an alignment score δ, we furthermore define an optimization goal *$goal_\delta \in \{\min, \max\}$.*

Here, the optimization goal $goal_\delta$ depends on the choice of the parameters $p(a, b)$ and g. For the parameters $p(a, b)$, we normally assume that $p(a, b) = p(b, a)$ holds for all $a, b \in \Sigma$.

Using the score of an alignment, we now define the similarity of two strings as the score of an optimal alignment of these two strings according to the optimization goal of the scoring function.

Definition 5.3. *Let s and t be two strings over an alphabet Σ, and let δ be an alignment scoring function. The* similarity *$sim_\delta(s, t)$ of s and t according to δ is the score of an optimal alignment of s and t, i.e.,*

$$sim_\delta(s, t) = goal_\delta\{\delta(s', t') \mid (s', t') \text{ is an alignment of } s \text{ and } t\}.$$

If the alignment scoring function is clear from the context, we also write sim instead of sim_δ.

There are many ways to choose these parameters, we will postpone a detailed discussion of this problem to Section 5.1.4 and only present two simple variants here.

The simplest way to score an alignment consists of simply counting the number of insertions, deletions, and substitutions. In this case we define $p(a, b) = 1$, if $a \neq b$, and $p(a, a) = 0$ for all $a, b \in \Sigma$, and $g = 1$. The resulting similarity measure is called *edit distance* or *Levenshtein distance*.

Another commonly used choice of the parameters is $p(a, a) = 1$, $p(a, b) = -1$, if $a \neq b$, and $g = -2$. In this case, the similarity of two strings is the maximum score of an alignment of the strings.

5.1.2 Global Alignment

In this subsection we present a method for the comparison of two strings. We generalize this method in the next subsection such that it is also suitable for the computation of similar substrings. To distinguish these two approaches, we call the computation of the similarity of the entire strings *global alignment*, and the computation of similar substrings *local alignment*.

We now present an algorithm that computes an alignment with optimal score for two given strings. We start by giving a formal definition of the problem.

Definition 5.4. *The problem of determining an optimal global alignment of two strings, the* global alignment problem*, is the following optimization problem:*

Input: Two strings s and t over an alphabet Σ and an alignment scoring function δ with an optimization goal $goal_\delta$.
Feasible solutions: All alignments of s and t.
Costs: For each alignment $A = (s', t')$ of s and t, the costs are $cost(A) = \delta(A)$.
Optimization goal: The optimization goal $goal_\delta$ of the scoring function δ.

For simplicity, in the remainder of this section, we assume that the optimization goal of the scoring function is to maximize the score over all alignments. This is no real restriction since all presented approaches directly carry over to the case of minimization.

The algorithm for computing an optimal alignment, as we will describe it, is based on the method of dynamic programming. The main idea is to compute an optimal alignment for all pairs of prefixes of the given strings.

Let $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$ be two given strings over an alphabet Σ. If we also count the empty prefix, there are $m + 1$ possible prefixes of s and $n + 1$ possible prefixes of t. We construct an $((m + 1) \times (n + 1))$-matrix M of similarity values for the optimal alignments of all pairs of prefixes. This means that the matrix entry $M(i, j)$ gives the score of an optimal alignment of $s_1 \dots s_i$ and $t_1 \dots t_j$. Particularly, the matrix entry $M(m, n)$ gives the score of an optimal alignment of s and t. We call this matrix M the *similarity matrix* for s and t.

The idea behind the dynamic programming now is to calculate the similarity value for two strings from the similarity values of their shorter prefixes.

This approach makes it possible to start with the shortest prefixes and to successively compute the similarity values for all longer prefixes.

The score of the optimal alignment of $s_1 \ldots s_i$ with the empty prefix of t is obviously $sim(s_1 \ldots s_i, \lambda) = g \cdot i$, for all $i \in \{1, \ldots, m\}$. Analogously, $sim(\lambda, t_1 \ldots t_j) = g \cdot j$ holds for all $j \in \{1, \ldots, n\}$. Using these values we can initialize the first row and the first column of our matrix M.

We now want to compute an optimal alignment of the prefixes $s_1 \ldots s_i$ and $t_1 \ldots t_j$ under the assumption that we already know the optimal alignments for all pairs of shorter prefixes. For the last column of the alignment[1] there are three possibilities: The last column consists of s_i and t_j or exactly one of the two strings of the alignment ends with a gap symbol. In each of these cases the score of the alignment can be calculated from the score of an alignment of shorter, already known prefixes adding the score of the last column. Thus, the score of an optimal alignment can be computed as the maximum value of the three cases, as follows:

$$sim(s_1 \ldots s_i, t_1 \ldots t_j) = \max \begin{cases} \underbrace{sim(s_1 \ldots s_{i-1}, t_1 \ldots t_j) + g,}_{\text{Insertion}} \\ \underbrace{sim(s_1 \ldots s_i, t_1 \ldots t_{j-1}) + g,}_{\text{Deletion}} \\ \underbrace{sim(s_1 \ldots s_{i-1}, t_1 \ldots t_{j-1}) + p(s_i, t_j)}_{\text{Match/Mismatch}} \end{cases} \tag{5.1}$$

From this formula, we can see that for the computation of the matrix entry $M(i, j)$ only the matrix entries $M(i-1, j)$, $M(i, j-1)$, and $M(i-1, j-1)$ are needed. After initializing row 0 and column 0 of the matrix with the multiples of g, the matrix can be filled row-wise (or column-wise) as shown in Figure 5.1.

Every path through this matrix starting at position $(0, 0)$, ending at position (m, n), and using only steps to the next position at the right, below, or diagonally down right, corresponds to a feasible alignment of s and t. Here, a step to the right corresponds to an insertion, i.e., to a gap in s, a step down corresponds to a deletion, i.e., to a gap in t, and a diagonal step corresponds to a match or mismatch.

Example 5.2. For the two strings $s =$ `AAAT` and $t =$ `AGT`, an application of Equation (5.1) results in the similarity matrix shown in Figure 5.2 (a). Here, we assume $p(a, a) = 1$, $p(a, b) = -1$ for $a \neq b$, and $g = -2$. ♢

Algorithm 5.1 computes the similarity of two given strings s and t by constructing the similarity matrix for s and t using dynamic programming according to Equation (5.1).

[1] Recall that we can view an alignment as a $(2 \times l)$-matrix.

Fig. 5.1. Computing an optimal alignment using dynamic programming

$s \backslash t$	0	A 1	G 2	T 3
0	0	−2	−4	−6
A 1	−2	1	−1	−3
A 2	−4	−1	0	−2
A 3	−6	−3	−2	−1
T 4	−8	−5	−4	−1

(a)

$s \backslash t$	0	A 1	G 2	T 3
0	0	−2	−4	−6
A 1	−2	1	−1	−3
A 2	−4	−1	0	−2
A 3	−6	−3	−2	−1
T 4	−8	−5	−4	−1

(b)

Fig. 5.2. (a) The similarity matrix for the strings from Example 5.2; (b) the computation of an optimal alignment for the strings using Algorithm 5.2

Using the similarity matrix, we now can recursively compute an optimal alignment of s and t with Algorithm 5.2. The idea behind Algorithm 5.2 is to recursively reduce the computation of an optimal alignment of $s_1 \dots s_i$ and $t_1 \dots t_j$ to the computation of an optimal alignment of $s_1 \dots s_{i-1}$ and $t_1 \dots t_j$,

Algorithm 5.1 Computation of the similarity

Input: Two strings $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$.

1. Initialization:
   for $i = 0$ **to** m **do**
     for $j = 0$ **to** n **do**
       $M(i, j) := 0$
2. Initialization of the borders:
   for $i = 0$ **to** m **do**
     $M(i, 0) = i \cdot g$
   for $j = 0$ **to** n **do**
     $M(0, j) = j \cdot g$
3. Filling the matrix:
   for $i = 1$ **to** m **do**
     for $j = 1$ **to** n **do**
       $M(i, j) := \max\{M(i-1, j) + g,$
       $M(i, j-1) + g,$
       $M(i-1, j-1) + p(s_i, s_j)\}$

Output: $sim(s, t) = M(m, n)$.

---

of $s_1 \dots s_i$ and $t_1 \dots t_{j-1}$, or of $s_1 \dots s_{i-1}$ and $t_1 \dots t_{j-1}$, depending on which of the three cases was used for computing $M(i, j)$.

Example 5.3. The similarity for the two strings from Example 5.2 is $sim(s, t) = -1$.

Every path from position $(4, 3)$ to position $(0, 0)$ following the arrows (as shown in Figure 5.2 (b)) corresponds to an optimal alignment; the optimal alignment computed by Algorithm 5.2 corresponds to the path shown with bold arrows. In this example, the optimal alignments of s and t are:

```
AAAT  AAAT  AAAT
-AGT  A-GT  AG-T
```

◊

We now analyze the time complexity of Algorithms 5.1 and 5.2.

Theorem 5.1. *Let $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$ be two strings over an alphabet Σ. Using Algorithms 5.1 and 5.2, an optimal alignment of s and t can be computed in $O(m \cdot n)$ time.*

Proof. Computing the similarity matrix M with Algorithm 5.1 needs time in $O(n \cdot m)$, since for the computation of every single matrix entry only constantly many calculations are needed. Algorithm 5.2, called with the parameters n and m, needs at most $m + n$ recursive calls before termination, since in every

Algorithm 5.2 Computation of an optimal alignment

Input: A similarity matrix M for two strings $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$.
Call the recursive procedure Align(m, n).
Output: The alignment (s', t') of s and t.
Procedure Align(i, j):

if $i = 0$ and $j = 0$ **then**
  $s' := \lambda$
  $t' := \lambda$
else
  if $M(i,j) = M(i-1,j) + g$ **then**
    $(\overline{s}, \overline{t}) :=$ Align$(i-1, j)$
    $s' := \overline{s} \cdot s_i$
    $t' := \overline{t} \cdot$ -
  else if $M(i,j) = M(i,j-1) + g$ **then**
    $(\overline{s}, \overline{t}) :=$ Align$(i, j-1)$
    $s' := \overline{s} \cdot$ -
    $t' := \overline{t} \cdot t_j$
  else $\{M(i,j) = M(i-1,j-1) + p(s_i, t_j)\}$
    $(\overline{s}, \overline{t}) :=$ Align$(i-1, j-1)$
    $s' := \overline{s} \cdot s_i$
    $t' := \overline{t} \cdot t_j$
return (s', t')

recursive call the value of at least one of the parameters is strictly reduced. Since the remainder of Algorithm 5.2 has only constant time complexity, the algorithm has an overall running time in $O(n + m)$. □

Algorithm 5.2 can be enhanced such that it outputs the set of all optimal alignments. But note that there can be exponentially many optimal alignments. For example, for every $n \in \mathbb{N}$ there are $\binom{2n}{n}$ optimal alignments for the strings $s = \mathtt{A}^{2n}$ and $t = \mathtt{A}^n$, since the score does not depend on the placement of the n gaps, and there are $\binom{2n}{n}$ possible placements. An algorithm that explicitly outputs all optimal alignments thus has an exponential worst-case running time.

In the following we present another approach for computing the optimal alignment of two strings relying on a graph-theoretic formulation of the problem.

Definition 5.5. *Let $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$ be two strings over an alphabet Σ and let δ be an alignment scoring function with parameters $p(a, b)$ and g for all $a, b \in \Sigma$. The* edit graph *for s and t according to δ is a directed acyclic, edge-weighted graph $G_\delta(s, t) = (V, E, c)$, where*

- $V = \{0, \dots, m\} \times \{0, \dots, n\}$,
- $E = \{((i,j),(i,j+1)) \mid 0 \leqslant i \leqslant m \text{ and } 0 \leqslant j \leqslant n-1\}$
  $\cup \{((i,j),(i+1,j)) \mid 0 \leqslant i \leqslant m-1 \text{ and } 0 \leqslant j \leqslant n\}$
  $\cup \{((i,j),(i+1,j+1)) \mid 0 \leqslant i \leqslant m-1 \text{ and } 0 \leqslant j \leqslant n-1\}$, *and*

- $c : E \to \mathbb{Q}$, $c((i,j),(i,j+1)) = c((i,j),(i+1,j)) = g$, $c((i,j),(i+1,j+1)) = p(s_{i+1}, t_{j+1})$ *for all* i, j.

We illustrate this definition with the two strings from Example 5.2.

Example 5.4. Figure 5.3 shows the edit graph for the strings $s =$ AAAT and $t =$ AGT. ♢

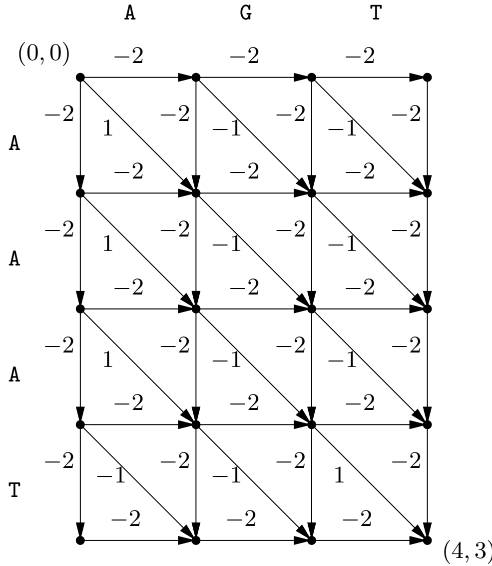


Fig. 5.3. The edit graph for the strings $s =$ AAAT and $t =$ AGT from Example 5.2

In the edit graph, the horizontal edges correspond to the gaps in s, the vertical edges correspond to the gaps in t, and the diagonal edges correspond to the matches and mismatches. Thus, every path from the vertex $(0,0)$ to the vertex (m,n) corresponds to a feasible alignment of the given strings. An optimal alignment of s and t can hence be computed by finding a path of maximal weight in $G_\delta(s,t)$ from the vertex $(0,0)$ to the vertex (m,n). There are simple and efficient algorithms known for computing such a path between two given vertices in a directed graph. We will not present such an algorithm here, since it can be found in many introductory textbooks about algorithms, as in [51].

5.1.3 Local and Semiglobal Alignment

We now show how to use the methods described in the previous subsection for the comparison of substrings of two given strings. This is a natural generalization of the methods for finding exact substrings as presented in Chapter 4 for the error-prone case.

The first problem we want to look at is the computation of an optimal local alignment according to the following definition.

Definition 5.6. *Let two strings $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$ over an alphabet Σ and an alignment scoring function δ with optimization goal maximization be given. A* local alignment *of s and t is a (global) alignment of substrings $\overline{s} = s_{i_1} \dots s_{i_2}$ and $\overline{t} = t_{j_1} \dots t_{j_2}$.*

An alignment $A = (\overline{s}', \overline{t}')$ of the substrings $\overline{s}$ and $\overline{t}$ is an optimal local alignment *of s and t according to δ if the following holds:*

$$\delta(A) = \max\{sim(\overline{s}, \overline{t}) \mid \overline{s} \text{ is a substring of } s, \overline{t} \text{ is a substring of } t\}.$$

Now we can formally define the problem of finding an optimal local alignment as follows.

Definition 5.7. *The problem of computing an optimal local alignment, the* local alignment problem, *is the following optimization problem:*

Input: Two strings s and t over an alphabet Σ and an alignment scoring function δ with maximization as its optimization goal.
Feasible solutions: All local alignments of s and t, i.e., all global alignments for all possible substrings $\overline{s}$ of s and $\overline{t}$ of t.
Costs: For a local alignment $A = (\overline{s}', \overline{t}')$ of the substrings $\overline{s}$ and $\overline{t}$, the costs are $cost(A) = \delta(A)$.
Optimization goal: Maximization.

For computing local alignments one only uses alignment scoring functions with maximization as the optimization goal. If one would use a scoring function like the edit distance, whose optimization goal is minimization, the optimal alignment would often be very short, only one symbol long in the worst case, and thus would not give any reasonable information.

The computation of an optimal local alignment is used, for example, for comparing unknown DNA or protein sequences. In these sequences, often only some parts are similar to each other; these similarities often cannot be found via a global alignment, as shown by the following example.

Example 5.5. Let $s =$ `AAAAACTCTCTCT` and $t =$ `GCGCGCGCAAAAA` be two strings. Consider the scoring function from Example 5.2 which scores a match with $+1$, a mismatch with -1 and a gap with -2. In each of the two strings, the substring `AAAAA` obviously occurs. Using the method of local alignment, we can find the region of high similarity. An optimal local alignment is

```
        AAAAA(CTCTCTCT)
(GCGCGCGC)AAAAA
```

with a score of 5. However, the optimal global alignment

```
AAAAACTCTCTCT
GCGCGCGCAAAAA
```

with score -11 does not give much information. ◇

To compute an optimal local alignment, it suffices to modify the global alignment algorithms presented above. We again compute an $((m+1) \times (n+1))$-matrix M, where $M(i,j)$ equals the highest score of an alignment between a *suffix* of $s_1 \dots s_i$ and a *suffix* of $t_1 \dots t_j$. For computing M we can use the following recursion:

$$M(i,j) = \max \begin{cases} M(i-1,j) + g, \\ M(i,j-1) + g, \\ M(i-1,j-1) + p(s_i,t_j), \\ 0 \end{cases}$$

As in the case of the global alignment, we additionally store, for each entry of the matrix, which of the four cases yielded the maximum in the recursion formula. Row 0 and column 0 of the matrix can be initialized with 0, since an alignment of the empty substrings with score 0 is always a feasible solution.

The score of an optimal local alignment is the highest value occurring in the matrix M. The corresponding local alignment can be found using a path along the stored pointers from the position of the highest entry to a position with value 0.

The method of using edit graphs can also be generalized to the case of local alignments: First construct the edit graph $G_\delta = (V, E, c)$ for two given strings $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$ according to an alignment scoring function δ, and then add edges of weight 0 from the vertex $(0,0)$ to all other vertices and from all other vertices to the vertex (m,n). In this extended edit graph, any path of maximal weight from the vertex $(0,0)$ to the vertex (m,n) corresponds to an optimal local alignment of s and t.

Another generalization of global alignment is the so-called *semiglobal alignment*, where one looks for an alignment of the entire strings, but where gap symbols at the beginning or at the end of the strings may be added without cost. The semiglobal alignment can be applied to compare strings of significantly different length. The computation of a semiglobal alignment for a long string t and a short string p can be viewed as a method for approximate string matching. The semiglobal alignment of t and p yields the substring of t that is most similar to p. The following example shows that neither the global nor the local alignment is suitable for this task.

Example 5.6. We consider the strings

$$s = \texttt{ACTTTATGCCTGCT} \text{ and } t = \texttt{ACAGGCT}$$

and the alignment scoring function scoring a match with 1, a mismatch with -1, and a gap with -2. An optimal *global* alignment of s and t is

```
ACTTTATGCCTGCT
AC---A-G---GCT
```

with score -7. But the alignment

```
ACTTTAT-GCCTGCT
-----ACAGGCT---
```

with score -13 looks more reasonable, since the string t stays more compact. If now the gaps before the first symbol of t and after the last symbol of t are ignored, this leads to a score of 0. Thus, using this semiglobal alignment method, this alignment get a significantly better score than the optimal global alignment.

Please note that our favorite alignment cannot be derived from the optimal local alignment of s and t. This optimal local alignment is

```
(ACTTTATGCCT)GCT
         (ACAGG)GCT.
```

◇

We distinguish several variants of semiglobal alignment, such as whether the gaps are for free only at the beginning, only at the end, or at both the beginning and end of the strings. These different variants can be useful in different applications. For example, if we ignore the gaps at the beginning of the first string and at the end of the second string, we have the possibility to compute an *approximate overlap* of the two strings. The computation for all of these variants can be easily reduced to the computation of a global alignment.

Let $s = s_1 \dots s_m$ and $t = t_1 \dots t_n$ be two strings over an alphabet Σ. We consider the case where gaps without cost are allowed at the end of the first string. Let (s', t') be an alignment of s and t of length l, where $t'_j = s_m$ holds for a position $1 \leqslant j < l$, i.e., where $s'_{j+1} \dots s'_l$ consists of gap symbols only. If we score the columns $j+1, \dots, l$ with 0, the score of this alignment equals the score of a global alignment of s with the prefix $t'_1 \dots t'_j$ of t. In the matrix M computed by Algorithm 5.1, the last row contains the scores of optimal alignments of s with all prefixes of t. Thus, the score of an optimal semiglobal alignment of this type is just the maximum of all values in the bottom row of M. Analogously, the score of an optimal semiglobal alignment ignoring the gaps at the end of t can be computed as the maximum of the values in the rightmost column of M.

We now consider the case where the gap symbols at the beginning of s are for free. The score of an optimal semiglobal alignment of this type equals the score of an optimal global alignment between s and a suffix of t. This score can be computed with a variant of Algorithm 5.1, where the first row of the matrix is initialized with zeros instead of multiples of the gap score. We leave a formal proof of the correctness of this modified algorithm to the reader. Analogously, the score of an optimal semiglobal alignment where the gaps at the beginning of t are ignored can be computed with a variant of Algorithm 5.1, where the first column of the matrix is initialized with zeros.

These changes of Algorithm 5.1 for the semiglobal alignment are summarized in Table 5.1. Please note that all four variants can also be arbitrarily

Gap symbols without cost	Change of Algorithm 5.1
Beginning of first string	Initialize first row of M with zeros
End of first string	Similarity equals maximum of the last row
Beginning of second string	Initialize first column of M with zeros
End of second string	Similarity equals maximum of the last column

Table 5.1. The different variants of semiglobal alignment

s \ t	0	A 1	G 2	T 3	A 4
0	0	−2	−4	−6	−8
A 1	0	1	−1	−3	−5
A 2	0	1	0	−2	−2
A 3	0	1	0	−1	−1
T 4	0	−1	0	1	−1

Fig. 5.4. The similarity matrix for computing the approximate overlap of the strings from Example 5.7

combined. An algorithm for determining approximate overlaps can be constructed by initializing the first column of the matrix M with zeros and taking the maximum of all values in the last row of M as output.

The following example shows how to actually determine the approximate overlap of two strings using semiglobal alignment.

Example 5.7. Consider the strings s = AAAT and t = AGTA. We want to compute the approximate overlap of s and t according to the scoring function from Example 5.2 that scores a match with 1, a mismatch with −1, and a gap with −2. This leads to the similarity matrix shown in Figure 5.4.

In this matrix, the first column was initialized with zeros and the first row with the multiples of the gap score as in Algorithm 5.1. Then the matrix was filled using the recurrence from step 3 of Algorithm 5.1. The maximum value 1 in the last row of the matrix now gives the score of an optimal approximate overlap of s and t. The corresponding semiglobal alignment

```
AAAT-
-AGTA
```

can now be computed with Algorithm 5.2, starting from the matrix entry $(4,3)$. $\Diamond$

5.1.4 Generalized Scoring Functions

In the previous subsections we have always used very simple scoring functions for the alignments. In this subsection we will present several possibilities for generalizing these scoring functions. These generalizations can be used to design more exact models of the underlying biological problems.

Scoring of Gaps

One problem that can arise with the scoring functions introduced above is the scoring of gaps. When comparing biological sequences, an alignment where the gap symbols occur in longer blocks should be scored better than an alignment where the gap symbols occur solitarily and scattered over the strings. To deal with this problem we will use the following definition.

Definition 5.8. *Let s and t be two strings, let (s',t') be an alignment of s and t. A substring $s'_{i+1}\dots s'_{i+k} = \text{-}^k$, where $s'_i, s'_{i+k+1} \neq \text{-}$ (or a substring $t'_{j+1}\dots t'_{j+k} = \text{-}^k$, where $t'_j, t'_{j+k+1} \neq \text{-}$) is called a* gap of length *k.* [2]

Up to now, we have scored a gap of length k with $k \cdot g$, where g was the score of a single gap symbol. In many biological contexts, the *affine gap score*, where a gap of length k is scored with $-(\varrho + \sigma k)$ for some $\varrho, \sigma > 0$, is more appropriate. In addition to the part σk that is proportional to the length of the gap, the opening of a gap is also penalized with a negative score. Depending on the choice of the parameters ϱ and σ, this can lead to a better score for long gaps than in the initial approach of counting the number of gap symbols only.

Computing an optimal alignment with affine gap scores is still possible using the dynamic programming approach, but the recurrence formula gets far more complicated. We will not present the resulting algorithm in detail here, but refer the reader to the references given in Section 5.5.

Scoring Matrices

Another problem occurs mainly when comparing protein sequences. Here, substitutions between certain pairs of amino acids are more likely than between others. This can be explained by the fact that there are amino acids with similar chemical properties as well as chemically very different amino acids. If a mutation replaces one amino acid by another one with very different properties, then it is very likely that this will cause the protein to lose its biological

[2] In the literature, "–" is often called *space* and a gap of arbitrary length is called *gap*.

function. In many cases this can result in a disadvantage for the concerned organism, in the worst case its death, so that a protein changed in this way has a low probability of being passed on to the next generation.

To take into account these differences in alignment of protein sequences, we have to assign different scores to different substitutions, i.e., we have to determine the parameters $p(a, b)$ of the scoring function in an adequate way. These parameters obviously can be written as a $(|\Sigma| \times |\Sigma|)$-matrix, which we call the *scoring matrix*. For the alignment of protein sequences we hence need a (20×20)-matrix containing a score for each pair of amino acids. There are essentially two methods for generating such a scoring matrix for protein alignments, which we briefly describe in the following.

The first approach is based on the construction of so-called PAM matrices. [3] To describe PAM matrices, we first need the following definition.

Definition 5.9. *An* accepted mutation *of a protein is a mutation that does not at all or only to a small extent alter the biological function of a protein, so that it can be passed on to the next generation.*

Two protein sequences s and t are said to be one PAM unit apart from each other *or* at PAM distance 1 *if s was transformed into t by a sequence of accepted point mutations (i.e., substitutions of single amino acids, not insertions or deletions) such that there occurs one point mutation per* 100 *amino acids on average.*

Two protein sequences that are k PAM units apart do not necessarily differ in k percent of their positions, since several mutations might occur at the same position. The PAM distance as defined above is an idealistic measure that cannot be determined exactly in practice. A *k-PAM matrix* is a scoring matrix that is suitable for comparison of protein sequences that are k PAM units apart. This raises the question of how to determine the k-PAM matrices.

We describe the construction of a k-PAM matrix in an ideal case. We assume that we know many pairs of homologous[4] protein sequences, of which we also know that they are k PAM units apart. Since we measure the distance between two sequences as the number of substitutions, we further assume that we know the optimal alignment, i.e., the positions of the gaps, for each pair of homologous sequences.

Let A be the set of all alignments of the given set of sequence pairs, and let $Col(A)$ be the multiset of all columns in A that do not contain a gap symbol. We then define, for each pair (a_i, a_j) of amino acids (symbols), the function value $freq(a_i, a_j)$ as the relative frequency of columns containing the values a_i and a_j, i.e., of the columns (a_i, a_j) and (a_j, a_i), under all columns in $Col(A)$. This means that

[3] The acronym PAM is explained differently in the literature: PAM stands for either *point accepted mutations* or *percent of accepted mutations*.

[4] Two protein sequences are called *homologous* if the corresponding proteins have the same biological function (for example, in different organisms).

$$freq(a_i, a_j) = \frac{\text{number of occurrences of } (a_i, a_j) \text{ and } (a_j, a_i) \text{ in } Col(A)}{2 \cdot |Col(A)|}.$$

This definition in particular ensures that $freq(a_i, a_j) = freq(a_j, a_i)$ holds.

Furthermore, we define for each amino acid a_i the function value $freq(a_i)$ as the relative frequency of the amino acid a_i within all alignments, i.e.,

$$freq(a_i) = \frac{\text{number of occurrences of } a_i \text{ within all alignments}}{\text{total length of all sequences}}.$$

Using the function *freq*, we then define the entry (i, j) of the k-PAM matrix PAM_k as

$$PAM_k(i, j) = \log \frac{freq(a_i, a_j)}{freq(a_i) \cdot freq(a_j)}.$$

This definition can be motivated as follows: The entry (i, j) describes the ratio of the probability with which the symbol a_i is transformed into the symbol a_j by accepted mutations to the probability that this pair of symbols occurs by chance within an alignment. To simplify calculations, the ratio is logarithmized subsequently. This makes it possible to calculate the product of ratios for several columns of a given alignment simply as the sum of the PAM values. In practice, the values $PAM_k(i, j)$ are furthermore multiplied by 10 and rounded to the next integer to accelerate calculations.

But since in practice we do not have ideal data for counting the frequencies of the substitutions, the actual computation of the PAM matrices uses more complicated methods. These methods are based on the following idea. In the first step, one chooses a set of very similar sequences that descend from a common ancestor, and of which one can assume that they are only one PAM unit apart. Since these sequences are very similar, they also have approximately the same length, and it is easily possible to determine the positions of the gaps in an optimal alignment. Thus, the assumptions for the ideal method are satisfied, and one can construct a 1-PAM matrix. Starting with the function values $freq(a_i, a_j)$ and $freq(a_i)$ for all symbols a_i and a_j, calculated for determining this matrix, one can also compute a k-PAM matrix for greater values of k as follows: Let F be a (20×20)-matrix such that $F(i, j)$ is the probability that a_i mutates to a_j within one PAM unit (not depending on the actual occurrence frequency of a_i). Then, the matrix F^k, i.e., k times the product of F with itself, specifies the probability of substitutions in sequences that are k PAM units apart. Using this matrix F^k, one can compute the entry (i, j) of the k-PAM matrix PAM_k as

$$PAM_k(i, j) = \log \frac{freq(a_i) \cdot F^k(i, j)}{freq(a_i) freq(a_j)} = \log \frac{F^k(i, j)}{freq(a_j)}.$$

It is possible to show that the k-PAM matrix computed in this way is symmetric, i.e., that $PAM_k(i, j) = PAM_k(j, i)$ holds, but we will not present the proof here.

Since in practice the PAM distance of the protein sequences one wants to compare is unknown, one normally uses some standard values; the values $k = 40$, $k = 100$, or $k = 250$ are commonly used.

The second type of scoring matrices often used in practical applications are the so-called BLOSUM matrices. As a basis for calculating the BLOSUM matrices, the so-called BLOCKS database was used; it contains information about similar regions in the amino acid sequences of related proteins. These data were obtained from the multiple alignment[5] of related protein sequences. A short, contiguous interval without gaps in such a multiple alignment is called a *block*. In the BLOCKS database, not the complete multiple alignments are stored, but only those blocks that are longer than some threshold length. A block in the BLOCKS database that corresponds to an interval of length n of a multiple alignment of k sequences $r_1, \ldots, r_k$ can be viewed as a matrix $B(i, j)$ with k rows and n columns.

In such a block, some of the rows can be identical or nearly identical, but for the computation of the BLOSUM matrix we are only interested in those pairs of rows which differ in at least a certain percentage of all positions. More precisely, we consider all pairs of matrix entries $(B(i_1, j), B(i_2, j))$, $1 \leqslant j \leqslant n$, such that the rows i_1 and i_2 of B differ at least at $x\%$ of the positions. For the computation of the BLOSUM-x matrix, we determine the set P of all such pairs from all blocks from the BLOCKS database and compute for all amino acids the relative frequency $\mathit{freq}(a, b)$ of the amino acid pair (a, b) in P and the relative frequency $\mathit{freq}(a)$ of the amino acid a in all pairs from P. Then we can compute the entry (i, j) of the BLOSUM-x matrix BLOSUM_x as

$$\mathit{BLOSUM}_x(i, j) = 2 \cdot \log_2 \frac{\mathit{freq}(a_i, a_j)}{\mathit{freq}(a_i) \cdot \mathit{freq}(a_j)},$$

rounded to the next integer value.

In contrast to the PAM matrices, the BLOSUM-x matrices were obtained (at least for high values of x) from the data of protein sequences that are evolutionarily far apart. This is an advantage, since also the protein sequences that one wants to compare using the scoring matrices are often not closely related. The commonly used values for the parameter x lie between 50 and 80; most commonly, the BLOSUM-62 matrix is used.

5.2 Heuristic Methods for Database Search

Although the alignment algorithms presented in the previous section have polynomial running time, they are too slow for searching in very large DNA or protein databases. Thus, in practice often faster heuristic methods are used, but these methods do not guarantee finding the optimal solution.

[5] A *multiple alignment* is an alignment of more than two strings. We present some approaches for computing multiple alignments in Section 5.3.

There are essentially two program systems for database search that are widely used and give useful solutions in most cases. In this section, we will present the algorithmic ideas underlying these programs. For a more detailed discussion of these heuristics we refer the reader to the literature given in Section 5.5.

5.2.1 The FASTA Heuristic

The first heuristic we present here is the program FASTA.[6] The program FASTA is permanently under development, and there exist several variants of it. We only describe the action of the program in principle. In the FASTA method, the pattern (the database query) is subsequently compared to all sequences stored in the database. We call these stored sequences database strings in the following. The comparison of the pattern to one of the database strings can be divided into four steps.

1. First, one chooses a parameter k and searches for all exact matches of length k between the pattern and the database string. Such an exact match is called a *hot spot* and can be unambiguously described by the pair of starting positions in the pattern and the database string. Typical values for the parameter k are $k = 6$ for the comparison of DNA sequences and $k = 2$ for protein sequences. For the search for such short matches one does not need complicated string matching algorithms, but it is sufficient for an efficient implementation to build a table in a preprocessing step that contains for each possible hot spot the positions in the pattern and the database string where the hot spot starts.
2. In the second step of the procedure one tries to cluster the hot spots. For this reason we consider a matrix M, similar to the similarity matrix used in Algorithm 5.1 for the global alignment, where the rows correspond to the symbols of the pattern p and the columns correspond to the symbols of the database string t. The matrix entry at position (i, j) is 1 if $p_i = t_j$, and 0 otherwise. In this matrix, each hot spot (i, j) corresponds to a segment of a diagonal starting at position $M(i, j)$ (see the example in Figure 5.5). Note that it is not necessary for the algorithm to explicitly construct the entire matrix M, since we only consider so-called *diagonal runs*, i.e., segments of diagonals from the matrix starting with a hot spot and ending with a hot spot. Every diagonal run gets scored, every contained hot spot contributes positively to the score, and every gap between two consecutive hot spots contributes negatively to the score, depending on the length of the gap. The algorithm then determines the ten diagonal runs with the best scores. Note that computing the diagonal runs can be done efficiently, in time proportional to the number of hot spots. Each of the ten chosen diagonal

[6] The name FASTA is an abbreviation for “fast all.” Here, “all” refers to the fact that the program FASTA can be used for both DNA and protein sequences, in contrast to its predecessor FASTP, which was suitable for protein sequences only.

runs determines an alignment of a substring of the pattern with a substring of the database string; it can contain matches and mismatches, but not insertions or deletions. For each of these substring pairs, the algorithm now determines an optimal *local* alignment with the algorithm presented in Section 5.1, using one of the scoring matrices presented above. The best partial alignment found in this way is called *init1*, and will serve as a starting point for step 4.

3. Nevertheless, in the third step all of the ten computed partial alignments are considered if their similarity values exceed a certain threshold. The algorithm tries to put these partial alignments together to achieve a longer alignment with a better score. This task can be described as a graph-theoretic problem: The partial alignments are represented as vertices in a directed graph labeled with the scores of the corresponding partial alignments. Let u be such a partial alignment ending at position (i, j), and let v be another partial alignment starting at position (i', j'). Then the graph contains a directed edge from u to v if and only if $i < i'$ and $j < j'$ hold, i.e., if the two partial alignments can be connected to yield a longer alignment. This edge is labeled with a negative weight depending on the distance of the positions (i, j) and (i', j') in the matrix. Then, an optimal alignment composed from the given partial alignments can be determined as a path of optimal score in the graph, where the score of a path is the sum of the vertex and edge labels on the path. The alignment constructed in this way is output as one solution by the FASTA algorithm.
4. In the fourth step, the algorithm computes an alternate solution, based on the optimal partial alignment *init1* computed in Step 2. This solution is computed using the exact algorithm for local alignments, where the computation of the similarity matrix is limited to a band of limited width around the diagonal containing the partial alignment *init1*.

These four steps are carried out for all strings stored in the database. Subsequently, the FASTA algorithm estimates the significance of the computed solutions using statistical methods, i.e., the algorithm tries to estimate the probability with which the similarity of a found solution could have occurred purely by chance. The used statistical methods are beyond the scope of this book, we refer the reader to the bibliography cited in Section 5.5.

Example 5.8. The principle of the work of the FASTA algorithm is shown in Figure 5.5 for the strings `TACCGA` and `ACTGAC`. ◊

5.2.2 The BLAST Heuristic

The second commonly used heuristic for search in sequence databases is the BLAST algorithm.[7] This program exists in many different implementations and variants, optimized, for example, for the search in DNA or protein databases. We just describe the main idea of the BLAST algorithm.

[7] An acronym for “Basic Local Alignment Search Tool.”

	A	C	T	G	A	C
T						
A	●				●	
C		●				●
C						
G				●		
A					●	

Fig. 5.5. The principle of the FASTA heuristic on the strings `TACCGA` and `ACTGAC`. The hot spots for $k = 2$ are shown as pairs of black dots in the matrix; a diagonal run is shaded dark. In this example, the optimal partial alignment coincides with the diagonal run. The lightly shaded area shows a band of width 3 around the partial alignment, within which an optimal local alignment is searched for in step 4

The BLAST algorithm consists of two components, a search algorithm and a statistical evaluation of the computed solutions. The search algorithm can be divided into the following three steps.

1. In the first step, the algorithm looks for so-called *hits*, i.e., similar substrings of a given length w in the pattern and in the database string. Typical values for the parameter w are $w = 11$ for DNA sequences and $w = 3$ for protein sequences. In contrast to the hot spots in the FASTA algorithm, the BLAST algorithm does not only search for exact matches of length w, but for all local alignments without gaps whose score exceeds a certain threshold. This threshold depends on the scoring matrix used. By using appropriate data structures, this step can be implemented in linear time.
2. In the second step, the algorithm searches for all pairs of hits that are at most at distance d from each other. Hits that do not belong to such a pair are not considered by the algorithm. The typically used values for the parameter d depend on the length w of the hits; for example, for hits of length 2 in protein sequences the value $d = 16$ is often used.
3. Now the algorithm tries to extend the pairs of hits by adding further alignment columns at both ends of the hits until the score does not increase any more. The initial BLAST version did not allow gaps in these extensions of the alignment. In more recent versions, the insertion of gaps is possible. If after this extension step the score of a pair of hits is above some threshold S, it is called a *high scoring pair* (*HSP*). The high scoring pairs, in the order of decreasing scores, constitute the output of the BLAST algorithm.

Furthermore, the BLAST algorithm computes a so-called *bit score* from the score of an HSP. This bit score is independent from the scoring matrix used and allows the direct comparison of different computations with different scor-

ing matrices. Finally, as with the FASTA algorithm, the BLAST algorithm also estimates the statistical significance of the computed results.

5.3 Multiple Alignments

In the previous sections we have seen how to compute alignments of two strings. We now want to discuss the question of whether it is possible to find efficient algorithms also for the comparison of more than two strings. We will present some approaches to multiple alignment, i.e., to the alignment of multiple strings, in this section. We will see that the problem of computing a multiple alignment is algorithmically significantly harder than the computation of an alignment of two strings.

5.3.1 Definition and Scoring of Multiple Alignments

Multiple alignment can be formally defined analogously to the alignment of two strings as follows.

Definition 5.10. *Let* $s_1 = s_{11} \dots s_{1m_1}, \ \dots, \ s_k = s_{k1} \dots s_{km_k}$ *be* k *strings over an alphabet* Σ. *Let* $\texttt{-} \notin \Sigma$ *be a gap symbol and let* $\Sigma' = \Sigma \cup \{\texttt{-}\}$. *Let* $h : (\Sigma')^* \to \Sigma^*$ *be a homomorphism, defined by* $h(a) = a$ *for all* $a \in \Sigma$ *and* $h(\texttt{-}) = \lambda$.

A multiple alignment *of* $s_1, \dots, s_k$ *is a* k*-tuple* $(s'_1, \dots, s'_k)$ *of strings of length* $l \geqslant \max\{m_i \mid 1 \leqslant i \leqslant k\}$ *over the alphabet* Σ', *such that the following conditions are satisfied:*

(a) $|s'_1| = |s'_2| = \dots = |s'_k|$,
(b) $h(s'_i) = s_i$ *for all* $i \in \{1, \dots, k\}$, *and*
(c) there does not exist any position where a gap occurs in all s'_i, *i.e., for all* $j \in \{1, \dots, l\}$ *there exists an* $i \in \{1, \dots, k\}$, *such that* $s'_{i,j} \neq \texttt{-}$.

The length $l = |s'_i|$ *is also called the length of the multiple alignment* $(s'_1, \dots, s'_k)$.

Informally speaking, condition (b) in Definition 5.10 says that deleting all gaps from the string s'_i yields the string s_i.

The next question is, how can we define the score of a multiple alignment? We present two different answers to this question.

For the first approach we need the definition of a consensus for a multiple alignment. Informally, a consensus of a multiple alignment can be determined by choosing from each column one of the most frequently occurring symbols. The distance of the alignment to the consensus then can be defined as the number of occurrences of symbols in the alignment that do not agree with the consensus in their columns.

Definition 5.11. *Let* $(s'_1, \ldots, s'_k)$ *be a multiple alignment of the strings* $s_1, \ldots, s_k \in \Sigma^*$ *and let* $l = |s'_1|$ *be the length of the alignment.*

A string $c = c_1 \ldots c_l \in \Sigma^l$ *is called a* consensus *for* $(s'_1, \ldots, s'_k)$ *if*

$$c_j = \operatorname*{argmax}_{a \in \Sigma} |\{s'_{ij} = a \mid 1 \leqslant i \leqslant k\}| \text{ for all } 1 \leqslant j \leqslant l.$$

The distance *of an alignment* $(s'_1, \ldots, s'_k)$ *with* $|s'_1| = l$ *to a consensus* c *is defined as*

$$dist(c, (s'_1, \ldots, s'_k)) = \sum_{j=1}^{l} |\{s'_{ij} \mid 1 \leqslant i \leqslant k, s'_{ij} \neq c_j\}|.$$

Any two consensus strings for a given multiple alignment are related as follows.

Lemma 5.1. *Let* $(s'_1, \ldots, s'_k)$ *be a multiple alignment, and let* c *and* $\overline{c}$ *be two consensus strings for this alignment. Then*

$$dist(c, (s'_1, \ldots, s'_k)) = dist(\overline{c}, (s'_1, \ldots, s'_k)).$$

Proof. For the proof, we consider one column j of the alignment. If $c_j \neq \overline{c}_j$ holds, the symbols c_j and $\overline{c}_j$ have to occur the same number of times in this column j. This implies

$$|\{s'_{ij} \mid 1 \leqslant i \leqslant k, s'_{ij} \neq c_j\}| = |\{s'_{ij} \mid 1 \leqslant i \leqslant k, s'_{ij} \neq \overline{c}_j\}|.$$

Since this holds for any column, summation over all columns of the alignment yields the claim. □

Since the distance to a consensus is independent of the choice of the consensus string, we define the *distance to consensus* for a multiple alignment $(s'_1, \ldots, s'_k)$ as the distance to an arbitrary consensus. This distance to consensus can be used as a measure for scoring a multiple alignment, such that a smaller distance to consensus means a better alignment.

We will illustrate this definition with an example.

Example 5.9. Consider the following alignment of the strings $s_1 =$ `AATGCT`, $s_2 =$ `ATTC`, and $s_3 =$ `TCC`:

$s'_1 =$ `AATGCT`
$s'_2 =$ `A-TTC-`
$s'_3 =$ `---TCC`
$c =$ `AATTCT`

Then c is a consensus of this alignment and $dist(c, (s'_1, s'_2, s'_3)) = 1 + 2 + 1 + 1 + 0 + 2 = 7$. ◊

Now we can formally define the problem of determining a multiple alignment with minimal distance to consensus as follows.

Definition 5.12. *The problem of computing a multiple alignment with minimum distance to consensus, the* MultConsensusAlign *problem, is the following optimization problem:*

Input: A set $S = \{s_1, \ldots, s_k\}$ of strings over an alphabet Σ.
Feasible solutions: All multiple alignments of S.
Costs: The costs of a multiple alignment $(s'_1, \ldots, s'_k)$ with a consensus c are

$$cost((s'_1, \ldots, s'_k)) = dist(c, (s'_1, \ldots, s'_k)).$$

Optimization goal: Minimization.

Note that Definition 5.11 gives only one possible definition of a consensus, often called *majority voting.* In the literature, the notion of consensus is often used in a broader sense for some string that is derived from a multiple alignment by some algorithm.

Another way of scoring a multiple alignment is based on reducing this problem to the computation of the scores of all pairwise alignments of the given strings. This idea leads to the following definition.

Definition 5.13. *Let Σ be an alphabet, let $\texttt{-} \notin \Sigma$ be a gap symbol, and let δ be a scoring function for the alignment of two strings over the alphabet Σ with optimization goal minimization, extended by an appropriate definition of $\delta(\texttt{-},\texttt{-})$.*

The score δ_{SP} of a multiple alignment $(s'_1, \ldots, s'_k)$ of length l as the sum of pairs*, or* SP-score *for short, is first defined column-wise. For all $1 \leqslant j \leqslant l$, let*

$$\delta_{\mathrm{SP}}(s'_{1j}, \ldots, s'_{kj}) = \sum_{i=1}^{k} \sum_{r=i+1}^{k} \delta(s'_{ij}, s'_{rj}).$$

The score of the alignment is then defined as the sum of scores over all columns, i.e.,

$$\delta_{\mathrm{SP}}(s'_1, \ldots, s'_k) = \sum_{j=1}^{l} \delta_{\mathrm{SP}}(s'_{1j}, \ldots, s'_{kj}).$$

We illustrate the SP-score of a multiple alignment with an example.

Example 5.10. Consider the following alignment from Example 5.9:

$$\begin{aligned} s'_1 &= \texttt{AATGCT} \\ s'_2 &= \texttt{A-TTC-} \\ s'_3 &= \texttt{---TCC}. \end{aligned}$$

As an alignment score, we will use the edit distance, i.e., $\delta(a, b) = 0$ if $a = b$, and $\delta(a, b) = 1$ if $a \neq b$, for all $a, b \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}, \texttt{-}\}$. Then the SP-score of this alignment is as follows:

$$\begin{aligned}\delta_{\mathrm{SP}}(s'_1, s'_2, s'_3) &= \sum_{j=1}^{6}\sum_{i=1}^{3}\sum_{r=i+1}^{3} \delta(s'_{ij}, s'_{rj}) \\ &= \sum_{j=1}^{6}(\delta(s'_{1j}, s'_{2j}) + \delta(s'_{1j}, s'_{3j}) + \delta(s'_{2j}, s'_{3j})) \\ &= (0+1+1) + (1+1+0) + (0+1+1) + (1+1+0) \\ &\quad +(0+0+0) + (1+1+1) \\ &= 11.\end{aligned}$$

◊

Now we can formally define the problem of computing a multiple alignment with optimal SP-score as follows. Keep in mind that in this section all considered alignment scoring functions will have minimization as their optimization goal. We will nevertheless use the notion of "similarity" for the score of an optimal alignment, although one could also call this value the degree of "non-similarity" since a similarity value of 0 corresponds to a total agreement of both strings.

Definition 5.14. *The problem of computing a multiple alignment with optimal SP-score, the* MultSPAlign *problem, is the following optimization problem:*

Input: A set $S = \{s_1, \dots, s_k\}$ of strings over an alphabet Σ and an alignment score δ.
Feasible solutions: All multiple alignments of S.
Costs: The costs of a multiple alignment $(s'_1, \dots, s'_k)$ are

$$cost((s'_1, \dots, s'_k)) = \delta_{\mathrm{SP}}(s'_1, \dots, s'_k).$$

Optimization goal: Minimization.

5.3.2 Exact Computation of Multiple Alignments

For computing an optimal multiple alignment, we can generalize the dynamic programming approach used for the pairwise alignment. Consider an input of k strings $s_1, \dots, s_k$, for which we want to compute an optimal multiple alignment. Then, we can use a k-dimensional array M, where the entry $A(i_1, \dots, i_k)$ contains the score of an optimal multiple alignment of the prefixes $s_{11} \dots s_{1i_1}, \dots, s_{k1} \dots s_{ki_k}$.

But this approach raises some difficulties, it is of limited use, especially for high values of k. One of the problems is that the data structure for maintaining a k-dimensional array is quite complex and difficult to handle if the parameter k is not known in advance but is part of the input, i.e., if one wants to construct a program that can compute an optimal multiple alignment for an arbitrary number of strings.

On the other hand, the time and space complexities of this method depend exponentially on the parameter k. To compute one entry of the array, we have to find a minimum of $2^k - 1$ values. In the case of the alignment of two strings, we constructed the minimum of three values corresponding to the three cases of inserting a gap into the first, the second, or none of the given strings. In the case of multiple alignment we now have to consider a special case for each possible combination of gaps in the current column of the alignment. Furthermore, the calculation of these values is more complicated in each of the cases. For example, if we consider the SP-score, computing the score of the current column requires time quadratic in k. If we assume that all given strings have the same length n, the array has n^k entries, and the computation of the array has time complexity in $O(k^2 \cdot 2^k \cdot n^k)$.

Thus, the dynamic programming method is only applicable for multiple alignment if one wants to compare only very few, say, three or four, strings. Essentially, there are no better methods known for exactly computing the optimal multiple alignment. On the contrary, this problem is NP-hard if the number k of strings is considered to be a part of the input.

For showing the NP-hardness of multiple alignment, we will first formally define the decision version of this problem and the problem we want to reduce it to.

Definition 5.15. *The decision version of the multiple alignment problem with SP-score* (DecMultSPAlign problem) *is defined as follows:*

Input: *A positive integer k and a set $S = \{s_1, \ldots, s_k\}$ of strings over an alphabet Σ, a scoring function $\delta : (\Sigma \cup \{\text{-}\})^2 \to \mathbb{Q}$, and a positive integer d.*

Output: Yes *if there exists a multiple alignment of the strings in S that has an SP-score with respect to δ less or equal to d.* No *otherwise.*

Definition 5.16. *A* supersequence *of a set $S = \{s_1, \ldots, s_k\}$ of strings is a string containing each s_i as a subsequence for all $i \in \{1, \ldots, k\}$. The decision version of the* shortest common supersequence *problem over the binary alphabet $\Sigma = \{0, 1\}$* (Dec(0,1)ShortestSuperseq problem) *is defined as follows:*

Input: *A positive integer k and a set $S = \{s_1, \ldots, s_k\}$ of strings over the alphabet $\Sigma = \{0, 1\}$ and a positive integer m.*

Output: Yes *if there exists a common supersequence t of the strings from S, such that $|t| \leqslant m$.* No *otherwise.*

Middendorf [145] has shown by a reduction from the vertex cover problem that this problem is NP-complete; we skip the proof here.

Lemma 5.2. *The* Dec(0,1)ShortestSuperseq *problem is NP-complete.*

□

Using Lemma 5.2, we are now able to prove the hardness of computing the multiple alignment with SP-score.

Theorem 5.2. *The* DecMultSPAlign *problem is computationally hard in the sense that any algorithm solving it can be used to construct an algorithm for the* Dec(0,1)ShortestSuperseq *problem only with a polynomial blow-up in the running time.*

Proof. We will present a polynomial-time reduction from the Dec(0,1)ShortestSuperseq problem to the DecMultSPAlign problem. Consider a set $S = \{s_1, \dots, s_k\}$ of strings over the alphabet $\{0, 1\}$ and a natural number m as an input for the Dec(0,1)ShortestSuperseq problem. Without loss of generality, we assume that $\max\{|s_i| \mid 1 \leqslant i \leqslant k\} \leqslant m$ holds. Furthermore, we may assume without loss of generality that $m \leqslant ||S||$ holds, where $||S|| = \sum_{i=1}^{k} |s_i|$ denotes the total length of all input strings. This assumption ensures that the value of m is polynomially bounded by the input size.

For this input, we now construct $m + 1$ different input instances for the DecMultSPAlign problem and show that there exists a common supersequence for S of length less than or equal to m if and only if for at least one of the constructed input instances for the DecMultSPAlign problem there exists a multiple alignment with SP-score below the given threshold.[8]

The idea behind the reduction is the following: We construct the input instance for the DecMultSPAlign problem by using the same strings that are given as input for the Dec(0,1)ShortestSuperseq problem plus two additional strings made from new symbols. Furthermore, we define the scoring function and the threshold d for the SP-score in such a way that we can guarantee that in every solution with an SP-score below the threshold d no column contains both 1s and 0s. If we can additionally ensure that the length of the multiple alignment has to be $\leqslant m$, the alignment immediately gives us a solution to the Dec(0,1)ShortestSuperseq problem. Since we do not know in advance how many 0-columns and how many 1-columns the DecMultSPAlign solution has, we have to construct $m + 1$ different input instances.

More formally, we can construct the input instances for the DecMultSPAlign problem as follows:

- For all $i, j \in \mathbb{N}$ such that $i + j = m$, let $X_{i,j} = S \cup \{a^i, b^j\}$, where a and b are two new symbols.

[8] Note that this does not imply the NP-hardness of the DecMultSPAlign problem in the usual sense, since we are using a variant of reduction here which constructs polynomially many input instances of the DecMultSPAlign problem for one input instance of the Dec(0,1)ShortestSuperseq problem. The usual reductions used to prove NP-hardness have to transform the given input instance into exactly one input instance of the other problem. For a discussion of the various types of possible reductions and the relations between them see [79, 124].

δ	0	1	a	b	-
0	2	2	0	2	1
1	2	2	2	0	1
a	0	2	0	3	1
b	2	0	3	0	1
-	1	1	1	1	0

Table 5.2. The alignment scoring function in the proof of Theorem 5.2

- Let $d = (k-1) \cdot ||S|| + (k+1) \cdot m$, where $||S|| = \sum_{s \in S} |s|$ denotes the total length of all strings in S.
- Let the alignment scoring function $\delta : (\{0, 1, a, b, \text{-}\})^2 \to \mathbb{Q}$ be defined as shown in Table 5.2.

Now it suffices to prove the following claim:

> The set S has a supersequence t of length m if and only if one of the sets $X_{i,j}$ has a multiple alignment with SP-score $\leqslant d$ according to δ.

To prove this claim we will show the two implications separately.

1. Consider, for an $X_{i,j}$, a multiple alignment $A = (s'_1, \ldots, s'_k, \alpha, \beta)$ with SP-score less or equal to d, where α is the row corresponding to the string a^i and β is the row corresponding to b^j. We first consider the restriction A' of A containing the rows $s'_1, \ldots, s'_k$ only. We show that the score of A' always is $(k-1) \cdot ||S||$, independent of the actual alignment A. To see this, we first examine one column of A'. If this column contains exactly l gaps, the comparison of gaps with non-gaps contributes costs of $l \cdot (k-l)$, and the $\frac{(k-l)(k-l-1)}{2}$ different pairs of non-gaps contribute costs of 2 each. This amounts to a total cost of this column of

$$l \cdot (k-l) + 2 \cdot \frac{(k-l)(k-l-1)}{2} = (k-1) \cdot (k-l).$$

Now let x be the number of columns of A' and let y be the number of gap symbols in A'. Then $||S|| = k \cdot x - y$ holds. Furthermore, let l_p denote the number of gaps in column p of A' for all $1 \leqslant p \leqslant x$. Then the score of A' satisfies

$$\begin{aligned}\delta_{\mathrm{SP}}(A') &= \sum_{p=1}^{x} (k-1) \cdot (k-l_p) = (k-1) \cdot \sum_{p=1}^{x} (k-l_p) \\ &= (k-1) \cdot (k \cdot x - y) = (k-1) \cdot ||S||.\end{aligned}$$

We will now column-wise determine the costs contributed by the comparison of α and β with A'. In a column containing an a (or b) we call any 1 (or 0) a *bad symbol*. For determining the costs, we distinguish three types of columns.

Consider a column where either α contains an a or β contains a b (but not both). If there are no bad symbols inside the column, it contributes costs of $k+1$; in other words, costs of $k-l$ for comparing the gap symbol in α or β with A', where l again denotes the number of gaps within this column of A', costs of l for comparing the a or b to A', and an additional cost of 1 for the comparison of α and β. Any bad symbol gives additional costs of 2.
Now consider a column containing both a and b. It contributes costs of $2k+3$, independent of A': The comparison of a and b contributes 3, the comparison of a 0 or 1 with a and b contributes 2, since the 0 or 1 is a bad symbol with respect to either a or b, and the comparison of a gap symbol with a and b contributes 2.
Finally, consider a column with gaps in both α and β. Such a column contributes costs of $2 \cdot (k-l)$, where l is the number of gaps in A' within this column. Since $l < k$ holds (otherwise the column would consist of only gaps, contradicting the definition of an alignment), the costs of this column are at least 2.
Summarizing the above arguments, we have established the following for a column c with l gaps in the A' rows:

$$cost(c) = \begin{cases} k+1 & \text{gap in either } \alpha \text{ or } \beta, \\ 2k+3 & \text{no gap in } \alpha \text{ or } \beta, \\ \geqslant 2 & \text{gaps in both } \alpha \text{ and } \beta. \end{cases} \tag{5.2}$$

We now show that any alignment of $X_{i,j}$ with an SP-score $\leqslant d$ has a length of m and does not contain any bad symbols.
According to the calculation above, it is clear that any alignment of length exactly m without bad symbols has costs of $(k-1)||S|| + m \cdot (k+1) = d$, since no column with gaps in both α and β can exist in such an alignment. It remains for us to show that any other alignment has higher costs. Obviously, any bad symbol increases the costs. Let us assume that there exists at least one column containing both a and b. Separating this column into two columns, one containing all zeros and the a, the other containing all ones and the b, decreases the cost by at least 1 (from $2k+3$ to $2 \cdot (k+1)$, according to Equation (5.2)). Thus, for every alignment with a and b inside the same column, there exists a cheaper alignment. This implies that no alignment of length $< m$ with costs $\leqslant d$ can exist. We now consider an alignment with $x > m$ columns. As shown above, we may assume that no column contains both a and b. Thus, there are m columns containing either a or b and $x-m$ columns with gaps in both α and β. This amounts to costs of at least $(k-1)||S|| + m \cdot (k+1) + 2(x-m) > d$ according to the above calculations. Thus, no alignment with $x \neq m$ columns can have costs $\leqslant d$.

From such an alignment of length m and without bad symbols, we can obtain a common supersequence t for $s_1, \ldots, s_k$ of length m: We define $t_l = 0$ if $\alpha_l = a$ and $t_l = 1$, if $\beta_l = b$.

2. Consider a common supersequence t for S of length m. Let i denote the number of zeros in t and let j denote the number of ones in t. We will show that the set $X_{i,j}$ has a multiple alignment with SP-score $\leqslant d$. Since t is a supersequence of S, there exists an alignment of p and t without mismatches for every string $p \in S$. To construct a multiple alignment of $X_{i,j}$, we just combine these alignments without mismatches. We assign the symbols a of the string a^i to the 0-columns of this multiple alignment of S, and the symbols b of the string b^j to the 1-columns. Then the SP-score of this multiple alignment (without the string t) is exactly d, as can easily be seen using calculations analogous to those in the first part of this proof. □

5.3.3 Combining Pairwise Alignments

Since we have seen in the previous subsection that determining an optimal multiple alignment exactly is hardly possible in practice, we want to present a method in this subsection that enables us to find at least an approximate solution to the problem. The idea behind this approach is to construct a multiple alignment by combining a set of pairwise alignments of the given strings.

For this approach, we first need the definition of compatibility of multiple alignments.

Definition 5.17. *Let $S = \{s_1, \ldots, s_k\}$ be a set of strings and let $T = \{s_{i_1}, \ldots, s_{i_m}\}$ be a subset of S. Let $A' = (s'_1, \ldots, s'_k)$ be a multiple alignment of S, and let $A'' = (s''_{i_1}, \ldots, s''_{i_m})$ be a multiple alignment of T.*

The alignment A' is called compatible *with the alignment A'' if the restriction of A' to the rows $i_1, \ldots, i_m$, where all columns consisting of gap symbols only are eliminated, equals the alignment A''.*

We will illustrate this definition with an example.

Example 5.11. Let $S = \{\texttt{ACGG}, \texttt{ATG}, \texttt{ATCGG}\}$, $T_1 = \{\texttt{ACGG}, \texttt{ATG}\}$, and $T_2 = \{\texttt{ATG}, \texttt{ATCGG}\}$. The alignment

```
A-CGG
A--TG
ATCGG
```

of S is compatible with the alignment

```
ACGG
A-TG
```

of T_1, since restricting the alignment of S to the first two rows yields

```
A-CGG
A--TG,
```

and eliminating the second column, consisting of gap symbols only, yields the above alignment of T_1. On the other hand, this alignment of S is not compatible with the alignment

```
AT-G-
ATCGG
```

of T_2, since restricting the alignment of S to the second and third row yields another alignment: namely the alignment

```
A--TG
ATCGG.
```

$\diamondsuit$

We will now show how to construct a multiple alignment using a tree of pairwise alignments. For this we will need the following definition.

Definition 5.18. *Let*$S = \{s_1, \ldots, s_k\}$ *be a set of strings over an alphabet* Σ. *A tree* $T = (V, E)$ *with* $V = \{s_1, \ldots, s_k\}$, *where every edge* $\{s_i, s_j\} \in E$ *is labeled with an optimal alignment* (s'_i, s'_j) *for* s_i *and* s_j, *is called an* alignment tree *for* S.

It is now possible to show that, starting from an alignment tree for a given set of strings, one can always compute a compatible multiple alignment.

Theorem 5.3. *Let* $S = \{s_1, \ldots, s_k\}$ *be a set of strings over an alphabet* Σ *and let* $T = (V, E)$ *be an alignment tree for* S. *Then a multiple alignment* $(s''_1, \ldots, s''_k)$ *for* S, *which is compatible with the optimal pairwise alignments* (s'_i, s'_j) *for all* $\{s_i, s_j\} \in E$, *can be efficiently determined.* $\square$

The alignment $(s''_1, \ldots, s''_k)$ is also called *compatible multiple alignment of* S *according to* T. We will not present the proof of Theorem 5.3 here; we refer the reader to the bibliographic notes in Section 5.5.

In the following we consider the special case where the alignment tree is a star, i.e., a tree with a center vertex c and $k-1$ leaves connected to the center. This special case is called *star alignment.* .

The algorithm for the star alignment is based on first choosing one of the given strings as the center of the star and then iteratively combining the optimal pairwise alignments of all other given strings with the center to achieve a compatible multiple alignment. If we have already constructed a multiple alignment of the strings c and $s_1, \ldots, s_i$ and want to combine it with an optimal pairwise alignment of c and s_{i+1}, we proceed using the principle "Once a gap — always a gap." This means that we try to insert as few gap symbols as possible into both alignments such that the resulting extensions

Algorithm 5.3 Star Alignment

Input: A set $S = \{s_1, \ldots, s_k\}$ of strings.

1. Compute the center c of the star:
   for $i := 1$ **to** k **do**
     for $j := i$ **to** k **do**
       Compute an optimal pairwise alignment of s_i and s_j and determine its score $sim(s_i, s_j)$.
   Define c as the string t minimizing the sum $\sum_{s \in S} sim(t, s)$.
   Define T to be the star with center c and leaves $S - \{c\}$.
2. Compute a compatible multiple alignment:
   for $i := 2$ **to** k **do**
     Compute a multiple alignment of c and $s_1, \ldots, s_i$ that is compatible with T from the already known compatible multiple alignment of c and $s_1, \ldots, s_{i-1}$ and the optimal pairwise alignment of c and s_i using the principle "Once a gap — always a gap".

Output: The computed multiple alignment of S, which is compatible with T.

---

of c coincide. This procedure is shown in Algorithm 5.3. We assume that the scoring function is chosen in such a way that an optimal alignment has a *minimum* score.

We illustrate the principle "Once a gap — always a gap" with an example.

Example 5.12. Consider the four strings c = `ATGCATT`, s_1 = `AGTCAAT`, s_2 = `TCTCA`, and s_3 = `ACTGTAATT` with the alignments

c' : `ATG-CATT`
s_1' : `A-GTCAAT`
s_2' : `-TCTCA--`

and

c'' : `A-TGC-ATT`
s_3'' : `ACTGTAATT`.

Inserting the gaps from both alignments into the string c yields the string c''' = `A-TG-C-ATT`, and keeping the columns from both alignments leads to the following multiple alignment of all four strings:

c''' : `A-TG-C-ATT`
s_1''' : `A--GTC-AAT`
s_2''' : `--TCTC-A--`
s_3''' : `ACTG-TAATT`.

◊

Example 5.12 moreover shows that using this method does not always result in an optimal multiple alignment. It would clearly have improved the total alignment to replace the string s_3''' with the string `ACTGT-AATT`.

But we will show in the following that Algorithm 5.3 achieves a good approximation of an optimal multiple alignment, if the scoring function used has some nice properties. The next definition formalizes the properties we have to assume.

Definition 5.19. *A scoring function $\delta : (\Sigma \cup \{\texttt{-}\})^2 \to \mathbb{Q}$ is called* good *if it satisfies the following properties.*

(i) $\delta(a, a) = 0$ holds for all $a \in (\Sigma \cup \{\texttt{-}\})$.
(ii) For all $a, b, c \in (\Sigma \cup \{\texttt{-}\})$ the triangle inequality *holds:*

$$\delta(a, c) \leqslant \delta(a, b) + \delta(b, c).$$

Another property of good scoring functions is given by the following lemma.

Lemma 5.3. *Let δ be a good scoring function. Then $\delta(a, b) \geqslant 0$ holds for all $a, b \in (\Sigma \cup \{\texttt{-}\})$.*

Proof. We know $0 = \delta(a, a) \leqslant \delta(a, b) + \delta(b, a) = 2 \cdot \delta(a, b)$ for all $a, b \in (\Sigma \cup \{\texttt{-}\})$. □

To prove that Algorithm 5.3 achieves a good approximation of the optimal multiple alignment we need the following lemma.

Lemma 5.4. *Let $\delta : (\Sigma \cup \{\texttt{-}\})^2 \to \mathbb{Q}$ be a good scoring function. Let $S = \{c, s_1, \ldots, s_k\}$ be a set of strings. Let $T = (S, E)$ be a star with center c and let $(c', s'_1, \ldots, s'_k)$ be a multiple alignment of S that is compatible with T. Then, for all $i, j \in \{1, \ldots, k\}$,*

$$\delta(s'_i, s'_j) \leqslant \delta(s'_i, c') + \delta(c', s'_j) = sim(s_i, c) + sim(c, s_j).$$

Proof. Since the score $\delta(s'_i, s'_j)$ of the pairwise alignment equals the sum of scores of symbol pairs over all columns, the claimed inequality directly follows from the triangle inequality we have required for δ. The claimed equality holds since the pairwise alignments between s_i and c as well as between c and s_j, induced by the multiple alignment $(c', s'_1, \ldots, s'_k)$, are optimal, and since $\delta(\texttt{-}, \texttt{-}) = 0$ holds. □

Now we have gathered all prerequisites for proving that Algorithm 5.3 computes a good approximation of an optimal multiple alignment.

Theorem 5.4. *Let δ be a good scoring function, and let δ_{SP} be the SP-scoring function induced by δ. Let $S = \{s_1, \ldots, s_k\}$ be a set of strings, and let $sim_{\mathrm{SP}}(S)$ be the SP-score of an optimal multiple alignment for S. Then the multiple alignment $(s'_1, \ldots, s'_k)$ computed by Algorithm 5.3 satisfies*

$$\delta_{\mathrm{SP}}(s'_1, \ldots, s'_k) \leqslant \left(2 - \frac{2}{k}\right) \cdot sim_{\mathrm{SP}}(s_1, \ldots, s_k).$$

Proof. Let $(s''_1, \ldots, s''_k)$ be an optimal multiple alignment of S according to the SP-scoring, i.e., let $\delta_{\mathrm{SP}}(s''_1, \ldots, s''_k) = sim_{\mathrm{SP}}(s_1, \ldots, s_k)$.

We define

$$v(s'_1, \ldots, s'_k) = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s'_i, s'_j) = 2 \cdot \delta_{\mathrm{SP}}(s'_1, \ldots, s'_k)$$

and

$$v(s''_1, \ldots, s''_k) = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s''_i, s''_j) = 2 \cdot \delta_{\mathrm{SP}}(s''_1, \ldots, s''_k) = 2 \cdot sim_{\mathrm{SP}}(S).$$

To prove the claim of the theorem it suffices to show that

$$\frac{v(s'_1, \ldots, s'_k)}{v(s''_1, \ldots, s''_k)} \leqslant 2 - \frac{2}{k}.$$

Let

$$M = \min_{t \in S} \sum_{s \in S} sim(s, t) = \sum_{s \in S} sim(c, s) = \sum_{s \in S - \{c\}} sim(c, s).$$

Without loss of generality we assume that $c = s_k$ holds.

Following Lemma 5.4, we know

$$\begin{aligned} v(s'_1, \ldots, s'_k) = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s'_i, s'_j) \quad &\leqslant \quad \sum_{i=1}^{k} \sum_{j=1}^{k} (sim(s_i, c) + sim(s_j, c)) \\ &= \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} (sim(s_i, c) + sim(s_j, c)) \\ &= \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} sim(s_i, c) + \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} sim(s_j, c) \\ &= 2 \cdot (k-1) \cdot \sum_{i=1}^{k-1} sim(s_i, c) \quad = \quad 2 \cdot (k-1) \cdot M. \end{aligned}$$

On the other hand,

$$\begin{aligned} v(s''_1, \ldots, s''_k) = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s''_i, s''_j) \quad &\geqslant \quad \sum_{i=1}^{k} \sum_{j=1}^{k} sim(s_i, s_j) \\ &\geqslant k \cdot \sum_{j=1}^{k} sim(c, s_j) \quad = \quad k \cdot M \end{aligned}$$

since, due to the choice of $c = \operatorname{argmin}_{t \in S} \sum_{s \in S} sim(t, s)$ we have, for any fixed i,

$$\sum_{j=1}^{k} sim(s_i, s_j) \geqslant \sum_{j=1}^{k} sim(c, s_j).$$

Hence,

$$\frac{v(s'_1, \ldots, s'_k)}{v(s''_1, \ldots, s''_k)} \leqslant \frac{2 \cdot (k-1) \cdot M}{k \cdot M} = 2 - \frac{2}{k}.$$

□

Theorem 5.4 shows that Algorithm 5.3 is a $(2-\frac{2}{k})$-approximation algorithm for the multiple alignment of k strings according to the SP-score for any good scoring function. It is possible to show that it also achieves a 2-approximation for scoring according to the distance to consensus. But the proof for this claim is a little more involved, and we will not present it here.

5.4 Summary

Computing optimal alignments is a method for comparing strings. One tries to align the strings with each other by inserting gaps, such that the aligned strings coincide as much as possible. For two given strings, an optimal alignment can be computed efficiently using the method of dynamic programming. We distinguish between global and semiglobal alignments for comparing the entire strings and local alignments for the search for substrings of maximal similarity.

To compare biological sequences, in particular protein sequences, one needs scoring matrices indicating the similarity of the single symbols. In practice, mainly two kinds of matrices are used, the PAM matrices and the BLOSUM matrices.

For searching biological databases one can use the alignment methods presented. But these exact methods often are still too slow for large databases. Therefore one normally uses heuristic methods. The best-known heuristics are the FASTA and the BLAST algorithms.

For comparing more than two strings simultaneously there are several ways of defining the score of a multiple alignment. Extending the exact algorithms from two strings to more strings is possible in principle, but inefficient. But under certain conditions it is possible to compute a good approximation of an optimal multiple alignment.

5.5 Bibliographic Notes

The textbooks by Gusfield [91], Pevzner [159], and Setubal and Meidanis [180] as well as the lecture notes by Schnitger [177] (in German) present the different approaches and methods for computing alignments in detail. A presentation of the many statistical approaches for the alignment problem, which we have

not discussed here, can be found in the books by Ewens and Grant [68], Clote and Backofen [49], Waterman [201], and Durbin et al. [62].

The algorithm for the global alignment of two strings is due to Needleman and Wunsch [147]; the algorithm for local alignment was designed by Smith and Waterman [184]. The edit distance as an alignment scoring function was first proposed by Levenshtein [132]. The use of affine gap scores was investigated independently by several researchers; for a detailed discussion and the detailed recursion formula see the book by Gusfield [91].

The PAM scoring matrices for the alignment of protein sequences are due to Dayhoff et al. [57], the BLOSUM matrices were developed by Henikoff and Henikoff [102]. The FASTA heuristics goes back to Lipman and Pearson [140, 155]; the description of a more current version of the program, which is always under development can for example be found in the work by Pearson [154]. Altschul et al. [10] developed the original version of the BLAST heuristic. An extended version that also allows the insertion of gaps while extending the hits is decribed in the paper by Altschul et al. [11]. A more detailed discussion of practical applications of these heuristics can be found in the books by Hansen [98] and Rauhut [164] (both in German). The analysis of the statistical significance of the results produced by the BLAST algorithm is described in detail in the book by Ewens and Grant [68].

The SP-score for multiple alignments was introduced by Carillo and Lipman [41]. The scoring by the distance to consensus was investigated by Li et al. [134]. The NP-completeness of the multiple alignment problem with SP-score was proved by Wang and Jiang [199]; the NP-completeness result for the problem of the shortest common supersequence used in their proof is due to Middendorf [145]. The concept of alignment trees and the proof that there exists a compatible multiple alignment for any alignment tree are due to Feng and Doolittle [72]. The approximation algorithm for the star alignment with SP-score was designed by Gusfield [90]. Also, for computing a multiple alignment with distance to consensus score there exists an approximation algorithm, described in the book by Gusfield [91].

Part II

DNA Sequencing

6

Introduction and Overview

DNA sequencing, i.e., determining the sequence of bases within a nucleic acid molecule, is an essential issue in molecular biology leading to a multitude of combinatorial problems. Thus, for solving these problems it is almost inevitable to use computational support. However, before we start describing the particular approaches to accomplish this task, as well as the resulting formal models, we will first discuss the importance of this topic.

For almost all living beings, in particular for humans, DNA provides the molecular structure carrying all information on the construction of other molecules. Thus, it plays a central role in the development of single individuals and in the controlling of vital processes inside organisms. Accordingly, DNA is often called the *blueprint of life*. Moreover, copies of the DNA are passed over from one generation to the other. This process from time to time leads to alterations of the base sequence, and hence also of its encoded information. While most of these alterations are insignificant, some of them may also have an enormous impact. Thus, knowledge about the information encoded in the DNA is crucial for understanding of many genetical diseases and evolutionary relations between species. This information essentially consists of the sequence of bases along the DNA molecule. We sketch some general approaches for deriving this base sequence.

First of all, we recall the chain termination method for sequencing as presented in Section 2.4.3. Using this method, the direct reading of bases is currently limited to about 1 000 bp, since longer DNA molecules will lead to non-tolerable error rates. In contrast, the total length of the human genome is about 3.5 Gbp, i.e., some million times of the molecule length that can be handled by this method. To address this problem, several different approaches have been proposed, two of which we outline in the following.

The first approach uses the following idea. We cut several copies of the investigated DNA molecule into a multitude of overlapping pieces that we will call fragments. These fragments have a length of about 50-300 kbp. During this process, the order of the fragments gets lost and has to be recovered using combinatorial methods. After having reconstructed this order, we are provided

with a so-called physical map of the molecule, which gives us an overview of the order and relative positions of the fragments. At the same time, we obtain a set of short regions along the DNA of which we know the base sequence and the exact position. These regions are called markers of the physical map. They may help us to determine the position of an unknown fragment within the DNA if we know that the marker is inside the fragment.

After we construct a physical map of the investigated DNA, a process we refer to as physical mapping, we aim at determining the base sequence of the single fragments independently. These fragments are not as large as the original DNA molecule, but are nevertheless far too large to be sequenced directly by means of the chain termination method. Thus, we again generate copies of the fragments and cut them into overlapping pieces, which are this time short enough for direct sequencing.[1] Here, the ordering of the subfragments again gets lost, and it has to be restored by clever combinatorial methods.

This clearly is only a rough outline of the first approach for DNA sequencing. It neglects many problems, additional information, and details. Some of these refinements will be discussed in the subsequent chapters. The international Human Genome Project essentially followed this approach. This project was launched in the early 1990s with the goal of sequencing the whole human genome, and it was more or less successfully finished in 2001 [106]. But this obviously does not mean that the task of DNA sequencing with all its problems can be considered to be definitively solved. Also, in the future, there will be an (even greater) need for the information coded in the DNA. A particularly important issue is the identification of gene positions within the now known base sequence, and to further utilize this knowledge, for instance, in the study of genetical defects.

The second approach for DNA sequencing, which we present next, is mainly based on the massive use of computational power and on clever algorithms, and was pursued by the company *Celera Genomics* [197]. The idea is based on simply omitting the costly first step of the previously described approach, i.e., the detour of physical mapping. Instead, the second step, consisting of cutting the DNA into fragments, sequencing these fragments, and recovering their order, is performed directly utilizing massive computing power and some additional information. For example, the length of the fragments is measured and, for each fragment, both prefix and suffix are sequenced. Although this may not cover the whole fragment, one gains two subsequences at a specified distance along the DNA, so-called mate pairs. This is called the *whole genome shotgun approach*, where the shotgun method refers to one possible implementation of the second step, i.e., sequencing the subfragments, in the first approach.

This raises the question, how helpful is it to determine the DNA sequence of one human individual, or even the common sequence of several individuals?

---

[1] If the subfragments are also too long to be totally sequenced, only a prefix or suffix of appropriate length will be sequenced.

In this context, we would like to mention the fact that the genome of all humans is identical up to 99.7%. Thus, a single DNA sequence can also serve as the basis for the solution of more general issues.

An excellent discussion of the issues presented is contained in an article by Myers [146], which is also very readable for computer scientists.

When describing the efforts for the sequencing of the human genome, we also have to critically discuss the results currently achieved. The sequences derived from both of these projects are by no means unambiguous. Moreover, there exist a number of differences that must be resolved by further investigations, such that eventually a more or less complete and unique sequence is achieved. But such a unique sequence itself does not provide us with the crucial structures within the sequence, such as the positions of genes. While the basic data is already available, the analysis requires a lot of additional effort.

Having given these introductory remarks on DNA sequencing, we now describe the approaches presented in more detail. In Chapter 7, we focus on methods for physical mapping. The physical maps obtained may then serve as a prerequisite for DNA sequencing, and also as a source of information for many other biomolecular issues. In Chapter 8, we present some methods for sequencing DNA molecules that are too long for the direct sequencing approach. We also discuss the shotgun approach mentioned above in more detail.

7

Physical Mapping

In the previous chapter we gave an intuition of our understanding of physical mapping and how we can utilize it in the context of DNA sequencing. We now make this more concrete by presenting some methods for the construction of such physical maps as well as by considering the resulting algorithmic problems. Let us start with a definition.

Definition 7.1. *Let $\mathcal{D}$ be a DNA sequence. A* physical map *consists of a set M of* genetic markers *and a function $p : M \to \text{Pot}(\mathbb{N})$ assigning to each marker in M the positions of its occurrence in $\mathcal{D}$. (A marker can typically be seen as a short DNA sequence.) The computation of such a physical map is denoted as* physical mapping.

The idea of a physical map is illustrated in Figure 7.1. Their usage is not restricted to the context of DNA sequencing; physical maps can moreover be utilized as a source of information for numerous other tasks. They may help, for instance, in the search of certain genes within a DNA sequence.

In the next sections we will present two different approaches to physical mapping in more detail. Section 7.1 is devoted to mapping based on restriction site data, while the methods described in Section 7.2 are based on hybridization data. The main statements of this chapter will be summarized in Section 7.3 and followed by some references to the literature for further reading in Section 7.4.

7.1 Restriction Site Mapping

The procedure for physical mapping we describe in the following is based on the application of *restriction enzymes*. These enzymes have the capability to recognize short subsequences within the DNA and to cut the DNA at, or near, this site. The sites are specific to each restriction enzyme and are accordingly called *restriction sites*, i.e., each restriction enzyme corresponds

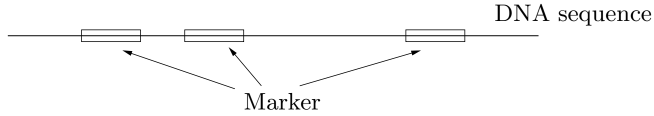


Fig. 7.1. Schematic view of a physical map

to a subsequence that defines the restriction site. The process of cutting the DNA by a restriction enzyme is called *digestion*.

A multitude of restriction enzymes occurring in nature, together with their corresponding restriction sites, are known. These will typically be denoted by a combination of the name of the organism where the enzyme was discovered first and a roman number. For instance, the restriction enzyme *HindIII* was first extracted from the bacterium *Haemophilus influenzae* R_d. Since we do not need such comprehensive information for our purpose, for simplification we denote the enzymes using the characters $\mathcal{A}$, $\mathcal{B}$, ..., and so on. An example for the function of restriction enzymes is depicted in Figure 7.2.

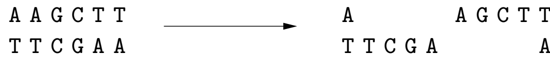


Fig. 7.2. Restriction site of restriction enzyme *HindIII*

We will now use restriction enzymes to cut the considered DNA and then determine the positions of the restriction sites along it from the resulting fragments. In this context, the restriction sites may serve as the markers of a physical map. In this procedure, furthermore, the ordering of the fragments resulting from the digest of the DNA is determined. Thus, it becomes possible to consider these fragments independently from each other in a step of DNA sequencing, i.e., to determine the sequence of the fragments separately and assemble the resulting sequences according to the ordering of the fragments thereafter, to finally obtain the total sequence of the considered DNA (as we described in greater detail in Chapter 6). Two approaches following the rough scheme described above will be presented in more detail in the next subsections.

7.1.1 The Double Digest Approach

Problem Setting

The idea of the *double digest approach* is to digest the considered DNA by two restriction enzymes separately, and then, in an additional experiment, by both of these enzymes. Thus, we have three digest experiments, one with the first enzyme, one with the second, and one that is a combination of the two. The lengths of the fragments resulting from the three setups are determined

and used for the computation of the ordering of the fragments, and hence also of the ordering of the restriction sites and their positions.

A schematic view of this double digest approach is given next.

Method 7.1 The double digest approach

Input: The considered DNA molecule $\mathcal{D}$ and two distinct restriction enzymes $\mathcal{A}$ and $\mathcal{B}$.

1. Generate three copies of $\mathcal{D}$.[1]
2. Apply enzyme $\mathcal{A}$ to the first, enzyme $\mathcal{B}$ to the second, and both enzyme $\mathcal{A}$ and enzyme $\mathcal{B}$ to the third copy. We obtain a set of unordered fragment from each copy.
3. Determine the lengths of the fragments of $\mathcal{D}$ and obtain three multisets:
   - $\Delta(\mathcal{A})$: Contains the lengths of the fragments resulting from the digest of the (first) copy of $\mathcal{D}$ by enzyme $\mathcal{A}$.
   - $\Delta(\mathcal{B})$: Contains the lengths of the fragments resulting from the digest of the (second) copy of $\mathcal{D}$ by enzyme $\mathcal{B}$.
   - $\Delta(\mathcal{AB})$: Contains the lengths of the fragments resulting from the digest of the (third) copy of $\mathcal{D}$ by enzyme $\mathcal{A}$ and enzyme $\mathcal{B}$.

Output: The multisets $\Delta(\mathcal{A})$, $\Delta(\mathcal{B})$, $\Delta(\mathcal{AB})$.

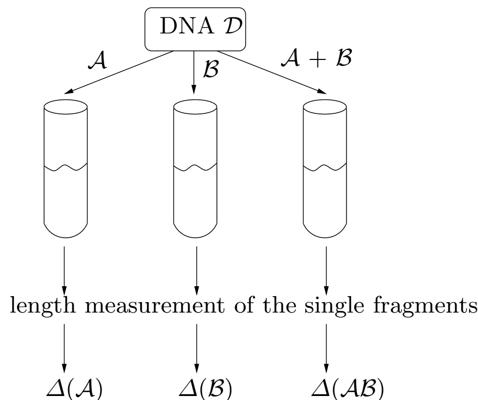


Fig. 7.3. Schematic view of the double digest approach

The principle of the approach is shown in Figure 7.3. Note the following two facts: Ideally a *full digest* is performed in this approach, i.e., the DNA is cut at *each* occurrence of the restriction site of the applied restriction enzyme; and the resulting sets $\Delta(\mathcal{A})$, $\Delta(\mathcal{B})$, and $\Delta(\mathcal{AB})$ actually denote *multisets*, since fragment lengths may occur more than once.

From the multisets $\Delta(\mathcal{A})$, $\Delta(\mathcal{B})$ and $\Delta(\mathcal{AB})$, we want, if possible, to reconstruct the ordering of the derived fragments and thus also of the positions of the restriction sites. This implies the so-called *double digest problem*. Before we give a formal definition, we will describe the underlying idea in an intuitive way.

We look for the orderings of the fragments of the multisets $\Delta(\mathcal{A})$ and $\Delta(\mathcal{B})$; we denote these orderings by π and ϕ, respectively. If we now order the fragments of $\Delta(\mathcal{A})$ according to π and the fragments of $\Delta(\mathcal{B})$ according to ϕ, then the restriction sites resulting from the overlay of the two orderings, i.e., from the boundaries of the fragments, should imply the fragment lengths within $\Delta(\mathcal{AB})$. See Figure 7.4 for a concrete example.

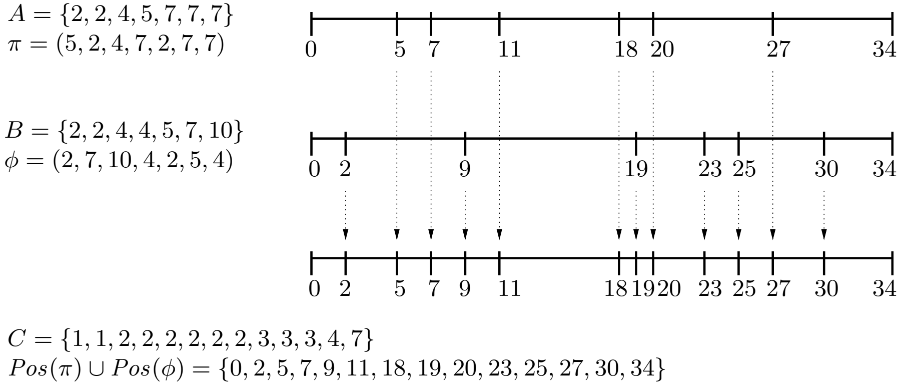


Fig. 7.4. Schematic view of the double digest problem. Find orderings π and ϕ of A and B, such that the multiset C is implied

To formalize this intuitive idea of the desired ordering, we start with some notations. We abstract from the biological data and consider general multisets A, B, and C instead of the multisets $\Delta(\mathcal{A})$, $\Delta(\mathcal{B})$, and $\Delta(\mathcal{AB})$. We refer to a triple of multisets such as A, B, C over positive integers ($\mathbb{N} - \{0\}$) as DD instances in the following.

Definition 7.2. *Let $X = \{x_1, \ldots, x_n\}$ be a multiset with elements from $\mathbb{N} - \{0\}$. Let $\pi = (x_{i_1}, \ldots, x_{i_n})$ be an ordering of the elements from X. By*

$$Pos(\pi) = \left\{ \sum_{j=1}^{k} x_{i_j} \mid 0 \leqslant k \leqslant n \right\}$$

we denote the position set of ordering π.

Conversely, let $Y = \{y_1, \ldots, y_p\}$ with $y_1 < y_2 < \cdots < y_p$ be a set of elements from $\mathbb{N}$. We denote by $Dist(Y)$ the multiset

$$Dist(Y) = \{|y_{i+1} - y_i| \mid i \in \{1, \ldots, p-1\}\}$$

of distances between consecutive elements in Y. We call $Dist(Y)$ the distance set of Y.

In our model the position set corresponds to an ordering of fragment lengths, i.e., to the positions of the endpoints of the fragments in the DNA molecule according to the given order (starting from 0). In Figure 7.4 the position sets are given in terms of intervals along a line. In some sense the distance set of a set denotes the inverse of the construction of a position set; from the positions of the endpoints of fragments along the molecule we can infer the multiset of fragment lengths by applying the function *Dist*. In particular $Dist(Pos(\pi)) = X$ holds for each ordering π of a multiset X.

The following definition specifies when we call a pair of orderings consistent with the data received from a double digest experiment.

Definition 7.3. *Let A, B, and C be multisets with elements from $\mathbb{N} - \{0\}$, where $n = |A|$ and $m = |B|$. Let π and ϕ be orderings of the elements from A and B, respectively. The pair (π, ϕ) is called a* feasible solution *to the DD instance A, B, C, if*

$$Dist(Pos(\pi) \cup Pos(\phi)) = C.$$

We are now ready to define our double digest problem.

Definition 7.4. *The* double digest problem*, DDP for short, is the following computing problem.*

Input: A DD instance A, B, C with elements from $\mathbb{N} - \{0\}$.
Output: An element of the set

$$\mathcal{M} = \{(\pi, \phi) \mid (\pi, \phi) \text{ is a feasible solution to } A, B, C\}$$

or the value 0 if $\mathcal{M} = \emptyset$.

We also denote the set $\mathcal{M}$ as the set of *feasible solutions* for the particular DDP.

Let us bring this definition in line with our intuition. Consider a pair of orderings (π, ϕ) from the set of feasible solutions for DDP. We can imagine arranging the fragments according to the orderings along the number line, starting with 0. We thus obtain all positions along the number line where fragments adjoin each other, which correspond to the sets $Pos(\pi)$ and $Pos(\phi)$, respectively. The set $Pos(\pi) \cup Pos(\phi)$ thus corresponds to the overlay of the two number lines. The positions in $Pos(\pi) \cup Pos(\phi)$ can hence be identified (from a biological point of view) as the restriction sites of $\mathcal{A}$ or $\mathcal{B}$. That is, the distances between adjacent points in $Pos(\pi) \cup Pos(\phi)$ must meet the multiset $C = \Delta(\mathcal{AB})$ to guarantee a useful ordering of the fragments. This issue is clarified in Figure 7.4 by an example.

In principle, the following holds: If a DD instance A, B, C does not obey the equation

$$\sum_{x \in A} x = \sum_{y \in B} y = \sum_{z \in C} z,$$

the set of feasible solutions for this input is necessarily empty. Informally, this equation holds if the multisets A, B and C result from the partition of the same basic distance. On the level of our biological motivation, it hence must hold that the lengths of the digested DNA sequences are all the same. Inputs that do not satisfy this equality need not to be considered and can be rejected directly. On the other hand, one has to note that we rely on ideal data in this context, which does not contain any errors, such as measurement errors.

Problem Complexity

A naive approach to solving the DDP is to test all possible orderings for the multisets A and B. This will result in up to $(|A|)! \cdot (|B|)!$ possible orderings[2] that have to be tested according to our definition. In a realistic setting, where there might be thousands of fragments for each restriction enzyme, this approach is obviously not practical.

We show next that the existence of a polynomial algorithm for DDP directly implies $P = NP$, and thus there is only slight hope that we can come up with an efficient solution for the problem. To prove this, we consider the corresponding decision problem.[3]

Definition 7.5. *The* decision version of the double digest problem, DecDDP *for short, is the following problem.*

Input: A DD instance A, B, C *with elements from* $\mathbb{N} - \{0\}$.
Output: Yes *if* $\mathcal{M} \neq \emptyset$, No *otherwise, where* $\mathcal{M}$ *denotes the set of feasible solutions for the DDP with input* A, B, C.

We are now able to prove the NP-completeness of DecDDP and hence show that there most likely exists no efficient solution. To do so, we use a reduction to the following set partition problem that is well known to be NP-complete [79].

Definition 7.6. *The* set partition problem *is the following decision problem.*

Input: A set $X = \{x_1, \ldots, x_n\}$ *with elements from* $\mathbb{N} - \{0\}$.
Output: Yes *if there exists a partition of* X *into disjoint sets* Y *and* Z, *such that*

$$\sum_{y \in Y} y = \sum_{z \in Z} z.$$

No *otherwise.*

---

[2] The maximum number of possible orderings occurs if neither A nor B includes multiple elements. In this case, the orderings π and ϕ are simply permutations of the elements.

[3] The DDP itself is actually a computing problem (see Section 3.3).

Thus, the set partition problem asks whether there exists a bipartition of the input set X such that the sum of the elements in both parts of the bipartition are equal.

Theorem 7.1. DecDDP *is NP-complete.*

Proof. To prove this we have to show two facts, namely, that DecDDP belongs to the class NP and that each problem in NP can be reduced to DecDDP in polynomial time.

1. Since we are able to check in polynomial time for each given pair of orderings (π, ϕ) whether it is a feasible solution for the DDP or not, DecDDP belongs to NP.
2. To show that all problems within NP can be polynomially reduced to DecDDP, we present a polynomial-time reduction of the set partition problem to DecDDP.
   Let X be an instance of the set partition problem. We can naturally restrict ourselves to sets X, where the overall sum of the included elements is even, since otherwise clearly no partition of the required type can exist. We now transform X into an input $I = (A, B, C)$ for the DecDDP as follows:
   - $A = X$,
   - $B = \{\frac{\alpha}{2}, \frac{\alpha}{2}\}$, where $\alpha = \sum_{x \in X} x$,
   - $C = X$.

   Here, the partition given by B forces a partition of $A = X$ into two parts of exactly the same size. This construction is visualized in Figure 7.5.

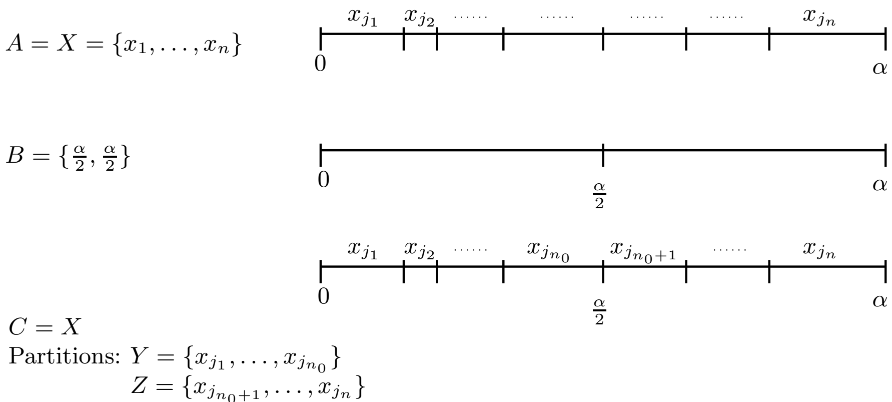


Fig. 7.5. Reduction of the set partition problem to the DecDDP

It remains for us to show that there exists a solution for the set partition problem on input X if and only if there exists at least one feasible solution for the DDP on input I, i.e., if and only if DecDDP gives output Yes.

a) According to the set partition problem, let X be decomposable into two disjoint sets Y and Z with $\sum_{y \in Y} y = \sum_{z \in Z} z$. Then $\pi = (\rho(Y), \rho'(Z))$ and $\phi = (\frac{\alpha}{2}, \frac{\alpha}{2})$ is a feasible solution to the DDP on input I for any arbitrary orderings ρ of Y and ρ' of Z.
b) Let (π, ϕ) be a feasible solution to the DDP on input I and let $\pi = (x_{j_1}, \ldots, x_{j_n})$. Then there exists an index n_0 with

$$\sum_{i=1}^{n_0} x_{j_i} = \sum_{i=n_0+1}^{n} x_{j_i} \ ,$$

since this decomposition has to be possible due to the choice of $B = \{\frac{\alpha}{2}, \frac{\alpha}{2}\}$ and $C = X$. Thus there also exists a feasible solution to the set partition problem on X. □

Hence, under the assumption P≠NP, already the decision version of DDP is not efficiently solvable, and one can thus easily conclude that DDP is also not efficiently solvable. If there were an efficient algorithm which is able to compute a feasible solution for the DDP, or to recognize that there is none, one could also directly infer an efficient method for the decision problem DecDDP.

Multiple Solutions

Another difficulty of the DDP depends on the fact that for a single input there may exist several feasible solutions. A trivial example for this is the reverse ordering of the elements in a feasible solution (π, ϕ). These two possibilities obviously always exist. But there might be also other solutions. For example, consider the input instance given in Figure 7.4. Besides the solution (π, ϕ) depicted there, the feasible solution (π', ϕ') with the ordering $\pi' = (2, 2, 7, 7, 4, 5, 7)$ of elements in A and the ordering $\phi' = (7, 2, 10, 4, 2, 5, 4)$ of elements in B is only one of more than 2 000 other feasible solutions for this rather small instance of the DDP. One approach to handling the large number of solutions is to join several solutions to a single class of solutions and to consider only one representative of each class. We make some bibliographic remarks on this in Section 7.4.

Problem Variants

Due to the fact that DecDDP is NP-complete and, moreover, due to the difficulty of several different feasible solutions described above, usually heuristics are used to address the DDP; they try to find a feasible solution but might fail, even if one exists. Another often considered variant of the DDP is to assign a cost value to each possible pair of orderings describing its distance

from a feasible solution. A pair with cost value 0 would correspond to a feasible solution for the original DDP. In this case, heuristics are also applied to find orderings with as low cost values as possible. Bibliographic notes to this approach are again given in Section 7.4.

In our description of the double digest approach we did not explicitly exclude the case where the restriction enzymes used share some common restriction sites.[4] If we assume this property, we obtain a restricted variant of the DDP known as *disjoint DDP*. For this problem, the proof of Theorem 7.1 is obviously invalid, since there we constructed an instance of the DDP, whose position set is actually not disjoint. Nevertheless, Cieliebak et al. [46] proved the NP-completeness of the disjoint DDP.

7.1.2 The Partial Digest Approach

Problem Setting

In the sequel, we consider a method that in contrast to the double digest approach discussed above uses a single restriction enzyme only. In this approach, the duration of exposure to the enzyme is varied, such that, most likely, the considered DNA is not cut at all restriction sites, and thus fragments appear that still contain some restriction sites of the enzyme used. This is actually the difference between a complete digest and a partial digest. The general idea of the *partial digest approach* is presented in the following.

Method 7.2 The partial digest approach

Input: The considered DNA sequence $\mathcal{D}$ and an restriction enzyme $\mathcal{A}$.

1. Generate several copies of $\mathcal{D}$.
2. Apply the enzyme $\mathcal{A}$ in separated experiments with different duration on the single copies and obtain a set of fragments from each experimental setup.
3. Determine the length of the resulting fragments of $\mathcal{D}$ and combine all these lengths into a single multiset $\Delta_p(\mathcal{A})$.

Output: The multiset $\Delta_p(\mathcal{A})$.

By this procedure we have a *partial digest* only in nearly all experimental set-ups. Therefore, we denote the resulting multiset by $\Delta_p(\mathcal{A})$ to explicitly distinguish it from the multiset $\Delta(\mathcal{A})$ obtained by the double digest approach. The basic principle of this approach is illustrated in Figure 7.6.

We subsequently assume that the partial digest experiment described above provides *ideal data* in the sense of the following definition.

[4] Indeed, there exist restriction enzymes whose restriction sites are not disjoint, but admittedly, a common application of those enzymes does not appear to be meaningful in this context.

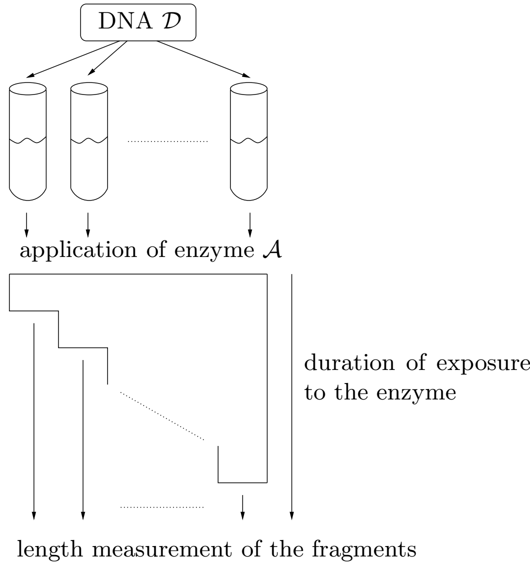


Fig. 7.6. The partial digest approach

Definition 7.7. *The multiset $\Delta_p(\mathcal{A})$ obtained by a partial digest experiment is called* ideal *if the length of each possible fragment occurs exactly once, i.e., if, for $c_1, \ldots, c_k$, $c_1 < c_2 < \ldots < c_k$, the positions of restriction sites of $\mathcal{A}$ along $\mathcal{D}$ (respectively the endpoints of the DNA) satisfy*

$$\Delta_p(\mathcal{A}) = \{c_j - c_i \mid 1 \leqslant i < j \leqslant k\}.$$

That is, each possible combination of two restriction sites of $\mathcal{A}$ (or endpoints of the DNA) leads to exactly one fragment whose length is then included in $\Delta_p(\mathcal{A})$. By an ideal partial digest experiment we thus obtain a multiset with exactly $\binom{k}{2}$ elements, where $k-2$ is the number of restriction sites along the investigated DNA.[5] This view is presented in Figure 7.7; the positions of restriction sites (endpoints) are denoted by $x_1, \ldots, x_k$.

From the data we obtained from an (ideal) partial digest experiment we would now like to reconstruct the ordering of the fragments, or the positions of the restriction sites. To formally define this task, we first describe the solutions that are consistent with the data from a partial digest experiment. Instead of $\Delta_p(\mathcal{A})$, we will now, as in the double digest approach, leave the explicit biological setting and use the term A instead.

Definition 7.8. *Let A be a multiset with $\binom{k}{2}$ elements from $\mathbb{N} - \{0\}$, and let $P = \{x_1, \ldots, x_k\}$ be a set of elements from $\mathbb{N}$, where $x_1 = 0$ and $x_1 < x_2 <$*

[5] Additionally, we have the two endpoints of the DNA.

$x_3 < \cdots < x_k$. *We denote such a set* P *as the* point set. *For such a point set* P*, one can compute the multiset*

$$Dist_p(P) = \{x_j - x_i \mid 1 \leqslant i < j \leqslant k\}$$

of all pairwise distances of elements from P. *A point set* P *is called a* feasible solution for A *if its induced pairwise distances actually meet* A*, i.e., if*

$$Dist_p(P) = A.$$

Definition 7.9. *The* partial digest problem, *PDP for short, is the following computing problem.*

Input: A multiset A *with* $\binom{k}{2}$ *elements from* $\mathbb{N} - \{0\}$.
Output: An element from the set $\mathcal{M} = \{P \mid P$ *is a feasible solution for* $A\}$*, or the value* 0 *if* $\mathcal{M} = \emptyset$.

A point set $P \in \mathcal{M}$ is also called a *feasible solution for the PDP.*

Informally, the partial digest problem is to reconstruct the positions $P = \{x_1, \ldots, x_k\}$ of restriction sites from the multiset A. Note that in contrast to the DDP, we do not search for an *ordering* of the elements from A, since we do not know which fragment lengths correspond to the distances between adjacent restriction sites. Fragments including restriction sites therefore form a structure of overlapping regions (see Figure 7.7).

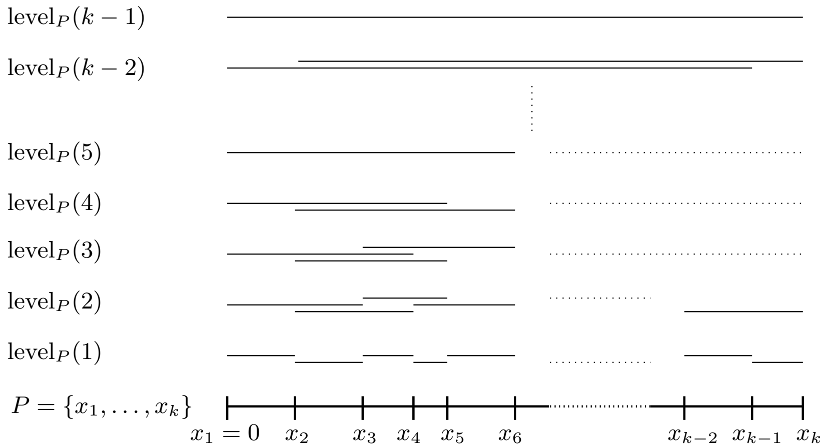


Fig. 7.7. Schematic view of the PDP

To consider the PDP in more detail, we first give a more precise characterization of the structure of feasible solutions, and thus of the structure of A. For this purpose, we introduce the following notation.

Definition 7.10. *Let A be a multiset with $\binom{k}{2}$ elements from $\mathbb{N} - \{0\}$, and let $P = \{x_1, x_2, \ldots, x_k\}$, with $x_i \in \mathbb{N}$ for all $i \in \{1, \ldots, k\}$, $x_1 = 0$, and $x_i < x_{i+1}$ for all $i \in \{1, \ldots, k-1\}$, be a feasible solution for the input instance A of PDP. Then,*

$$\text{level}_P(i) = \{x_{j+i} - x_j \mid j \in \{1, \ldots, k-i\}\} \subseteq A$$

denotes the multiset of distances whose corresponding endpoints have distance i in P for $1 \leqslant i \leqslant k-1$.

As with the point set P, the single levels are primarily unknown. But this definition will be helpful in further analysis of the PDP. Let us now give some remarks on the single levels.

Remark 7.1. Let A be a multiset with $\binom{k}{2}$ elements from $\mathbb{N} - \{0\}$ and let $P = \{x_1, x_2, \ldots, x_k\}$ be a feasible solution for the PDP on input A. Then,

- $|\text{level}_P(i)| = k - i$.
- In our biological motivation, $\text{level}_P(1)$ corresponds to the multiset of lengths whose corresponding fragments are bordered (as in DDP) by *adjacent* restriction sites (or endpoints of the DNA). We denote these elements of $\text{level}_P(1)$ as *atomic distances*, or *atoms* for short.
- $\text{level}_P(k-1) = \{y_{\max}\}$, where $y_{\max} = \max A$.
  $\text{level}_P(k-1)$ thus contains only the length of the DNA sequence itself, i.e., the distance between the two endpoints of the DNA.
- The levels form a kind of disjoint partition,[6]

  $$\text{level}_P(1) \,\dot{\cup}_m\, \text{level}_P(2) \,\dot{\cup}_m \ldots \dot{\cup}_m\, \text{level}_P(k-1) = A,$$

  where $\dot{\cup}_m$ denotes the disjoint union of multisets.

Algorithmic Approaches

Let us first consider the naive approach for solving the PDP, i.e., testing all reasonable point sets. In this case, according to the overlap of the fragments (see Figure 7.7), it is not sufficient to simply check all possible orderings of elements from A. Rather, we have to choose the atomic distances from A first, and to check all possible orderings for each of these choices. It is not surprising that this procedure is, as in the case of the DDP, not applicable in practice. Namely, we would have to choose $k-1$ atomic distances first, which would result in up to $\binom{\binom{k}{2}}{k-1}$ possibilities. Additionally, we would have to check for each of these choices all possible orderings of the $k-1$ distances. This sums

[6] Note that we consider a partition of A into multisets here. Therefore, disjoint in this context means that the sum of occurrences of each element in the single multisets adds up exactly to the number of occurrences in A.

up to $\binom{\binom{k}{2}}{k-1} \cdot (k-1)!$ possibilities we would have to test, i.e., an exponentially large quantity that is not verifiable in reasonable time.

Note that determining $\text{level}_{\Delta_p(\mathcal{A})}(1)$, and thus the atomic distances, is experimentally easy by simply performing a full digest experiment using $\mathcal{A}$. Hence, one could determine $\text{level}_{\Delta_p(\mathcal{A})}(1)$ as $\Delta(\mathcal{A})$, but in this case the number of possibilities is $(k-1)!$, which is far too large.

We now present an algorithm for the PDP that, instead of naively testing all possibilities, develops partial solutions stepwise and only takes into further consideration those that have the chance to be completed to feasible solutions. The running time of this approach will be $O(2^k)$ for an input A with $\binom{k}{2}$ elements, and while still exponentially large in the worst case, smaller than in the previous naive approach. However, as we will see, this algorithm is also suitable in practice, since for many inputs it needs only polynomial time.

The Backtracking Method

This algorithm is based on a widely used algorithmic design method, the so-called *backtracking*. Parts of the solution are specified successively if they can be further refined to a complete solution, or, if this turns out to be impossible, are excluded from the specification of the solution (this is also called the backtracking step). In this way one can examine all possible solutions, while aberrations can be recognized and excluded from further considerations. We will clarify this method in detail with the following algorithm for the PDP. Here, the specification of a partial solution corresponds to fixing a part of the point set that is further completed by adding points step by step. If it is recognized that a part of the point set (partial solution) cannot be extended to a complete solution, the last added position is removed from the subset and not added again. Once the algorithm reaches a feasible solution, it outputs this and stops its calculation.

Before we start describing the details of this algorithm, we introduce the following notation. Let δ be the function that for a given positive integer y and a given set of positive integers $X = \{x_1, \ldots, x_n\}$ computes the multiset of pairwise distances between y and all numbers from X; thus,

$$\delta(y, X) = \{|x - y| \mid x \in X, x \neq y\}.$$

Applied to our problem, the function δ hence computes all distances that newly appear if we introduce the position y as a point into the set X.

To actually implement the backtracking step, we will use a stack[7] to store each added position used to reconstruct the previous configuration in case of a backtracking step. Let us now consider Algorithm 7.1.

[7] A stack denotes a data structure that directly corresponds to an ordinary stack. Using an operation called Push we can file an element on the top of the stack; using an operation called Pop we can remove the topmost element from the stack.

Algorithm 7.1 PDP Backtracking

Input: A multiset A with $\binom{k}{2}$ elements from $\mathbb{N} - \{0\}$.

1. Sort the elements of A according to their size.
2. $X := \emptyset$ {the so far constructed point set}
3. $S :=$ empty stack {stack to control backtracking steps}
4. {Place the largest fragment}
   $y_{\max} := \max A$ {the largest element in A}
   $X := X \cup \{0, y_{\max}\}$ $\{x_1 := 0,\ x_k := y_{\max}\}$
   $A := A - \{y_{\max}\}$
5. Place further fragments (right or left) by a recursive call of procedure Place(X, A, S).

Procedure Place(X, A, S)
  if $A = \emptyset$ **then**
    Output: "feasible solution" X; **halt**
  $y := \max A$ {the largest element in the current set A}
  if $\delta(y, X) \subseteq A$ **then** {placement on left-hand side}
    $A := A - \delta(y, X)$
    $X := X \cup \{y\}$
    Push(y, S)
    Place(X, A, S)
  else if $\delta(y_{\max} - y, X) \subseteq A$ **then** {placement on right-hand side}
    $A := A - \delta(y_{\max} - y, X)$
    $X := X \cup \{y_{\max} - y\}$
    Push$(y_{\max} - y, S)$
    Place(X, A, S)
  else if $S \neq$ empty stack **then** {backtracking is possible}
    $y' :=$ Pop(S) {recall last position from stack}
    $A := A \cup (\delta(y', X) - \{0\})$ {reconstruct previous distance set}
    $X := X - \{y'\}$ {reconstruct previous position set}
    return {backtrack to invoking procedure}
  else {backtracking is impossible}
    $S =$ empty stack
    Output: "There is no feasible solution!"; **halt**

First of all, we can restrict the inputs for Algorithm 7.1 to multisets of $\binom{k}{2}$ elements for some natural number k, since otherwise there cannot exist a solution for the PDP for that input.

Now we have to describe how points are successively added to our partial solution. This is not done by simply checking all points, but in a clever way. Starting from the empty point set as the initial partial solution, the algorithm successively places the largest remaining distance y at the left-hand side, $[0 \ldots y]$, or at the right-hand side, $[y_{\max} - y \ldots y_{\max}]$, of the considered interval $[0 \ldots y_{\max}]$, if such an placement is possible, i.e., if all resulting distances $\delta(y, X)$ [resp. $\delta(y_{\max} - y, X)$] are still included in our set A. If a placement

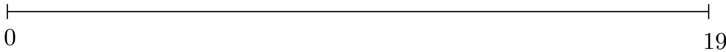
at the left-hand side is possible, this placement will be checked until either a solution is found or a placement of the current element becomes impossible. In the latter case, the previous decision to place the element at the left-hand side is revised and the element is placed at the right-hand side if possible. If both placements turn out to be impossible, because the resulting distances are not included in A, and if additionally no decision remains that can be revised (i.e., if the stack is empty), then there exists no feasible solution for the given input as we will formally show in Theorem 7.2. In this way, all possible solutions are successively built from partial specifications, where partial solutions that provably cannot be extended to complete solutions are rejected as early as possible.

We illustrate this algorithm with the following example.

Example 7.1. Let us consider the multiset

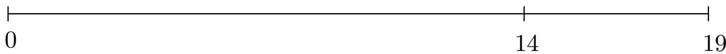
$$A = \{1, 2, 3, 4, 5, 5, 7, 7, 9, 9, 10, 10, 12, 14, 19\}$$

with $\binom{6}{2} = 15$ elements.
After execution of steps 1-4 we have

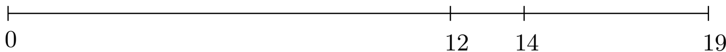


In the sequel, we present the values of the relevant variables after each execution of the procedure Place and illustrate the so far specified subsolution with a figure.

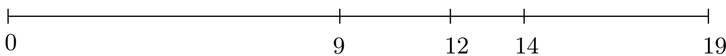
(a) $X = \{0, 19\}$; $A = \{1, 2, 3, 4, 5, 5, 7, 7, 9, 9, 10, 10, 12, 14\}$;
$y = 14$; $S = ()$;
$\delta(14, X) = \{5, 14\} \subseteq A$;
$\Rightarrow$ Placement of the distance 14 at the *left*-hand side.



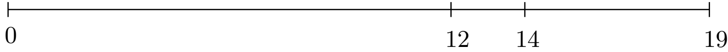
(b) $X = \{0, 14, 19\}$; $A = \{1, 2, 3, 4, 5, 7, 7, 9, 9, 10, 10, 12\}$;
$y = 12$; $S = (14)$;
$\delta(12, X) = \{2, 7, 12\} \subseteq A$;
$\Rightarrow$ Placement of the distance 12 at the *left*-hand side.



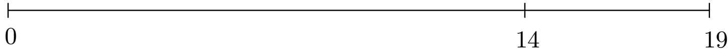
(c) $X = \{0, 12, 14, 19\}$; $A = \{1, 3, 4, 5, 7, 9, 9, 10, 10\}$;
$y = 10$; $S = (14, 12)$;
$\delta(10, X) = \{2, 4, 9, 10\} \not\subseteq A$;
$\delta(19 - 10, X) = \delta(9, X) = \{3, 5, 9, 10\} \subseteq A$;
$\Rightarrow$ Placement of the distance 10 at the *right*-hand side.



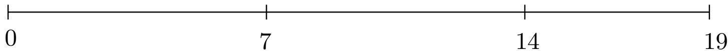
(d) $X = \{0, 9, 12, 14, 19\}$; $A = \{1, 4, 7, 9, 10\}$;
$y = 10$; $S = (14, 12, 9)$;
$\delta(10, X) = \{1, 2, 4, 9, 10\} \not\subseteq A$;
$\delta(19 - 10, X) = \delta(9, X) = \{0, 3, 5, 9, 10\} \not\subseteq A$;
$\Rightarrow$ Execution of a backtracking step. Undo the placement of step (c).



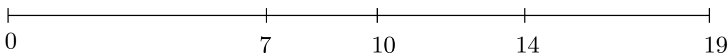
(e) $X = \{0, 12, 14, 19\}$; $A = \{1, 3, 4, 5, 7, 9, 9, 10, 10\}$;
$y = 10$; $S = (14, 12)$;
$\delta(10, X) = \{2, 4, 9, 10\} \not\subseteq A$;
$\delta(19 - 10, X) = \delta(9, X) = \{3, 5, 9, 10\} \subseteq A$; undone by backtracking in step (d).
$\Rightarrow$ Since there is no further possibility to place y, a backtracking step is again performed. Undo the placement of step (b).



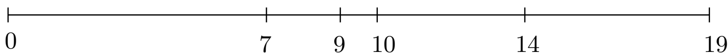
(f) $X = \{0, 14, 19\}$; $A = \{1, 2, 3, 4, 5, 7, 7, 9, 9, 10, 10, 12\}$;
$y = 12$; $S = (14)$;
$\delta(12, X) = \{2, 7, 12\} \subseteq A$; undone by backtracking in step (e).
$\delta(19 - 12, X) = \delta(7, X) = \{7, 7, 12\} \subseteq A$;
$\Rightarrow$ Placement of the distance 12 at the *right*-hand side.



(g) $X = \{0, 7, 14, 19\}$; $A = \{1, 2, 3, 4, 5, 9, 9, 10, 10\}$;
$y = 10$; $S = (14, 7)$;
$\delta(10, X) = \{3, 4, 9, 10\} \subseteq A$;
$\Rightarrow$ Placement of the distance 10 at the *left*-hand side.



(h) $X = \{0, 7, 10, 14, 19\}$; $A = \{1, 2, 5, 9, 10\}$;
$y = 10$; $S = (14, 7, 10)$;
$\delta(10, X) = \{0, 3, 4, 9, 10\} \not\subseteq A$;
$\delta(19 - 10, X) = \delta(9, X) = \{1, 2, 5, 9, 10\} \subseteq A$;
$\Rightarrow$ Placement of the distance 10 at the *right*-hand side.



Hence, the point set $P = \{0, 7, 9, 10, 14, 19\}$ is a feasible solution to the PDP with input $A = \{1, 2, 3, 4, 5, 5, 7, 7, 9, 9, 10, 10, 12, 14, 19\}$. The reverse ordering of the atomic distances gives the point set $\overline{P} = \{0, 5, 9, 10, 12, 19\}$, which obviously is a feasible solution as well. $\Diamond$

Based on this example, we can establish the following two facts.

Remark 7.2.

(i) If $P = \{0, x_2, x_3, \dots, x_k\}$ is a feasible solution of the PDP for a multiset A, $\overline{P} = \{0, x_k - x_{k-1}, x_k - x_{k-2}, \dots, x_k - x_2, x_k\}$ is also a feasible solution. We call $\overline{P}$ the *inverse solution* to P.

(ii) The placement of the first distance in step 5 of Algorithm 7.1 is arbitrary. Both placements (left and right) are always possible if there exists any feasible solution. The choice of the right-hand side instead of the left-hand side may lead to the inverse solution.[8]

Figure 7.8 shows the search tree constructed by our algorithm for the input from Example 7.1.

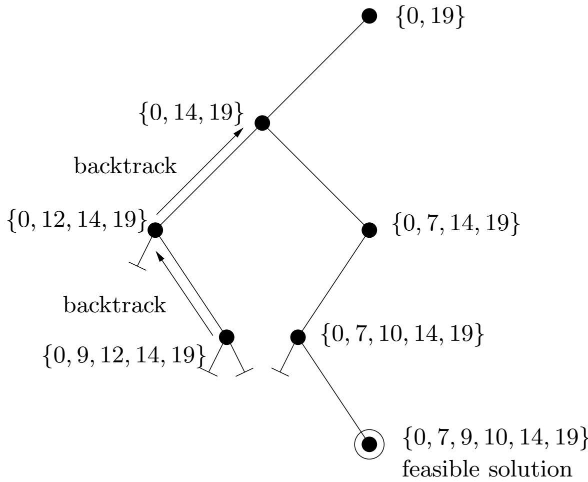


Fig. 7.8. Search tree of Algorithm 7.1 for the input from Example 7.1. Each vertex is labeled with its specified subsolution. Edges of the type $\perp$ denote invalid specifications of solutions

We show next that our algorithm works correctly.

Theorem 7.2. *Let A be an input for the PDP. If there exist feasible solutions for the PDP with input A, then Algorithm 7.1 will compute one of them.*

Proof. Algorithm 7.1 examines all possible solutions in which the longest remaining distance y is placed at either the left-hand or the right-hand side. Thus, it is sufficient to show that we can traverse all possible solutions in this

[8] This is not inevitable. If there exist more than two feasible solutions (P and $\overline{P}$) for the particular input instance, one of these further solutions may be computed by a placement of the distance at the right-hand side, since the preference is to place distances on the left-hand side from that time on.

way, i.e., that a placement of y is only possible at the left-hand side or the right-hand side.

Let $X = \{0, x_2, x_3, \ldots, y_{\max}\}$ be the partial solution constructed so far, A be the multiset of remaining distances originating from the input multiset, and y be the largest distance in A. Note that 0 and $y_{\max}$ were already introduced to the partial solution X in step 4 of Algorithm 7.1. Assume now, in contrast to our claim, that we are able to place the distance y "in the middle" of our position set, that is, within an interval $[z \ldots z + y]$ with $0 < z < y_{\max} - y$ (in particular, $z \notin \{0, y_{\max} - y\}$). At this point we will consider three different cases and show that each of them yields a contradiction.

- If $z+y \notin X$ then $z+y \in A$ must hold, since $0 \in X$. But this contradicts the assumption that y is the largest remaining distance in A, because $y < z+y$ (see Figure 7.9 (a)).
- If $z \notin X$ then $y_{\max} - z \in A$ must hold, since $y_{\max} \in X$. But this again contradicts the assumption that y is the largest remaining distance in A, because $y < y_{\max} - z$ (see Figure 7.9 (b)).
- Finally, in the case that both positions z and $z + y$ are already included in X, a placement of y within this interval is impossible, since otherwise distances of length 0 must appear that are not included in the original multiset, and hence also not in the current multiset A. □

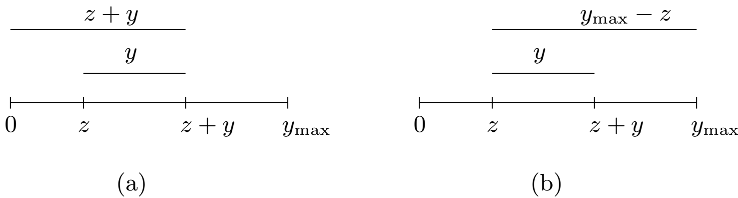


Fig. 7.9. For a placement of y within the interval $[z \ldots z + y]$ either the longer distance $z + y$ occurs in A (a), or the longer distance $y_{\max} - z$ occurs in A (b)

Theorem 7.3. *Algorithm 7.1 has a worst-case running time of* $O(2^k \cdot k \log k)$ *for an input* A *with* $\binom{k}{2}$ *elements.*

Proof. The sorting of the elements of A in step 1 of our algorithm is possible in time $O(k^2 \log k)$.[9] The initializations in steps 2 to 4 can be done in constant time. According to the proof of Theorem 7.2, the algorithm in the worst case checks all possibilities to place the $k - 1$ currently longest distances to either the left-hand side or the right-hand side. Hence, we have at most 2^{k-1} possible placements. For each placement we have to compute the

[9] There are well known algorithms that sort a multiset of n elements in time $O(n \log n)$ (see [51]).

at most $O(k)$ distances, that came along with this placement by the function δ. To check whether these distances occur in A, we may perform a binary search $O(k)$ times. This implies a running time of $O(k \log k)$ for the placement of a distance. If we implement the set A using a data structure that allows us to label chosen distances, the backtrack step can be viewed as the reverse of a placement step, and thus also runs in time $O(k \log k)$. This hence implies a complexity of $O(k \log k)$ for each placement and backtracking step, respectively. Summing up, we obtain the claimed total running time of $O(2^k \cdot k \log k + k^2 \cdot \log k) = O(2^k \cdot k \log k)$. □

Our backtracking algorithm thus has the positive property that the running time, while still being exponential, is significantly reduced with respect to the naive approach. Moreover, this running time concerns the worst case only. Indeed, it is possible to show an expected polynomial running time for inputs resulting from a random decomposition of a basic distance. If we thus assume that in practice, since the restriction sites are more or less randomly distributed over the DNA, only a constant number of backtracking steps is required, i.e., that the number of backtracking steps is independent of k, we are able prove the following theorem.

Theorem 7.4. *Under the assumption that the number of backtracking steps required by Algorithm 7.1 is independent of k for an input A of size $\binom{k}{2}$, the running time of Algorithm 7.1 is in $O(k^2 \log k)$. If this is measured with respect to the input size $n = \binom{k}{2}$, this implies a running time of $O(n \log n)$.*

Proof. Due to our assumption, only a constant number of wrong placements will occur. The algorithm will thus place at most $O(k)$ many distances altogether. To perform the sorting of multiset A in Step 1 of Algorithm 7.1, a running time in $O(k^2 \log k)$ is required. Placements and backtracking steps can, analogously to the proof of Theorem 7.3, be performed in time $O(k \log k)$, which finally implies a total running time of $O(k^2 \log k + k^2 \log k) = O(k^2 \log k)$. □

On the negative side, we refer to a result in the bibliographic notes that shows that inputs for the PDP exist on which Algorithm 7.1 requires an exponential running time.

However, another advantage of our backtracking algorithm is that it can be easily modified to compute not only *one* but *all* feasible solutions. In the biological framework of physical mapping, this may in fact be a crucial aspect that may in practice enable us to further analyze different proposed solutions and to verify their correctness also from the biological point of view.

7.1.3 Comparison of Methods for Restriction Site Mapping

Both methods for restriction site mapping discussed in the previous two sections depend on the decomposition of DNA by means of restriction enzymes.

In this decomposition, the ordering of the resulting fragments is lost. Our desire to reconstruct the order and thus to get the ordering of the restriction sites that provides the physical map led us to the combinatorial tasks presented above.

The double digest approach, where the DNA is fully digested each by two restriction enzymes $\mathcal{A}$ and $\mathcal{B}$ as well as by a combination of both enzymes, has the advantage of being easier to handle from an experimental point of view. On the other hand, there is the provable hardness of the resulting combinatorial problem to reconstruct the ordering of the fragments. We have proved the NP-hardness of the decision variant of this problem and thus shown that our computational problem is NP-hard in the sense that there does not exist any polynomial algorithm for this problem, unless P = NP. To date, essentially heuristic approaches have been used that do not claim any quality of the computed solutions, but obtain quite good results in practice.

The partial digest approach is based on a partial digest of the DNA in different setups, which ideally provides all fragments that are bordered by any pair of restriction sites within the original DNA sequence. The large experimental effort is obvious. To ensure the generation of all these fragments, much care and time are required. On the other hand, the resulting combinatorial problem can be solved by an exact exponential algorithm by applying a backtracking approach; moreover, the resulting algorithm turns out to be applicable in practice, since for practical instances the running time may essentially decrease. Whether there exists an exact algorithm with polynomial running time on all input instances is still unknown, but also the NP-hardness of the problem has not been shown yet.

To recapitulate, one may note:

- double digest: simple experiments — difficult combinatorial problem.
- partial digest: complex experiments — easy combinatorial problem.

At the end of this section, we have to admit that all our considerations were based on idealized data. In the experimental process of collecting data to investigate in our combinatorial tasks, errors are indispensable. In particular, measuring errors may occur, but there might be also other error sources. For instance, the assumption of ideal data in the case of partial digest experiments can be easily invalid, for instance, if a fragment is missing or the same fragment appears more than once.

All these sources of error do not simplify the problems; actually the problems arising become more complex. To account for measurement errors, it is especially necessary to allow for a certain variability with respect to the distance in the tasks, for instance, by introducing a kind of relative variance. Further errors may lead to almost invincible problems. The use of heuristics appears to be the only choice in this case. We will again deal with this task in Section 7.4.

7.2 Fingerprinting and Mapping by Hybridization

First, we recapitulate our main objective again. In essence, our goal is the construction of a physical map, i.e., the derivation of a collection of markers and their positions along a DNA strand. For this purpose we decomposed the DNA under investigation and subsequently tried to order the resulting fragments according to their order in the original DNA.

If we now start with decompositions of the original DNA that lead to overlapping *fragments*, as in the partial digest approach, we may try to use the overlaps of the fragments to determine the ordering.

But how can we derive the overlapping structure of the fragments? Generally speaking, one may try to describe each fragment by specific characteristics that can be compared to a keyword collection and that can, furthermore, be easily inferred. These characteristics of fragments are referred to as their *fingerprints*. The method of *fingerprinting* depends on the assumption that fragments with similar fingerprints will overlap in the original DNA sequence; and, conversely, that overlapping fragments will have similar fingerprints. Thus, it becomes possible to derive conclusions concerning the overlap of the fragments by the fingerprints, and, from that, one may in the next step develop hypotheses on the ordering of fragments in the original DNA.

It remains for us to find suitable candidates for fingerprints. We will now present some useful examples of fingerprints of fragments that have already been applied in practice.

- *Restriction site mapping:* If restriction site maps of the single fragments, as discussed in the previous section, are available, we may assume that two fragments overlap if both have a series of consecutive fragments in the restriction site map in common. This process can be understood as a kind of a hierarchical approach decomposing the fragments whose fingerprints we want to determine into even smaller fragments by means of restriction site mapping. The restriction site maps of these smaller subfragments are subsequently used as fingerprints of the larger fragments.
- *Fragment sizes after digestion by a restriction enzyme:* Instead of fully computing a restriction site map and using it as a fingerprint, one might be content with the lengths of the fragments resulting from a digest of the DNA by a restriction enzyme. If a significant part of these lengths agree with each other, one might presume an overlap of the corresponding fragments.
- *Hybridization data:* Many recent approaches are based on fingerprints that originate from hybridization data. The rest of this section is devoted to the discussion of these approaches and to the detailed modelling of the inferred algorithmic tasks.

First, let us recall the definition of hybridization from Chapter 2. Hybridization is the connection of two complementary nucleic acid chains, as a rule according to the Watson-Crick complement. In this way we have a sort

of substring test available, since we can conclude that a certain string is a substring of a nucleic acid chain, if the chain and the complement of the considered string hybridize. Actually, the DNA chips presented in Section 2.4.4 are suitable to perform this test.

In the following we do not further elaborate on the fact that *complementary* nucleic acid chains pair by hybridization, but we undertake our considerations with respect to an abstract substring test only. We call the known nucleic acid chains, on whose occurrences as substrings we want to test, *probes*, and we call the nucleic acid chains we want to investigate *clones*.[10] The substring test performed in this way we also call a hybridization experiment.

The physical mapping based on hybridization experiments proceeds according to the following scheme.

Method 7.3 Mapping by Hybridization

Given: The DNA sequence $\mathcal{D}$ under investigation.

1. Cut, according to an arbitrary method (restriction enzymes, sonic waves, vibration), several copies of $\mathcal{D}$ into a set of fragments. This set will include overlapping fragments with high probability.
2. For each of the fragments generate copies that will be used within a hybridization experiment. The fragments are called clones.
3. Let $C = \{c_1, \ldots, c_n\}$ be the set of clones (also called clone library), and choose a set of probes $P = \{p_1, \ldots, p_m\}$.
4. Perform all hybridization experiments (c_i, p_j), $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$, (for example, by means of a DNA chip).

Output: An $(n \times m)$-*hybridization matrix* $\mathcal{H}$, where

$$\mathcal{H}(i,j) = \begin{cases} 1 & \text{if } c_i \text{ and } p_j \text{ hybridize, and} \\ 0 & \text{otherwise.} \end{cases}$$

From this hybridization matrix $\mathcal{H}$ we now want to reconstruct the original ordering of the clones along the DNA $\mathcal{D}$. The positions of the probes that will be known afterwards can then serve as markers of the physical map. At this point we have to note that by means of the hybridization matrix we cannot determine *how often* each clone occurs in the considered probe. We are only able to distinguish whether a probe never occurs or occurs at least once. The definition of the resulting task is given next.

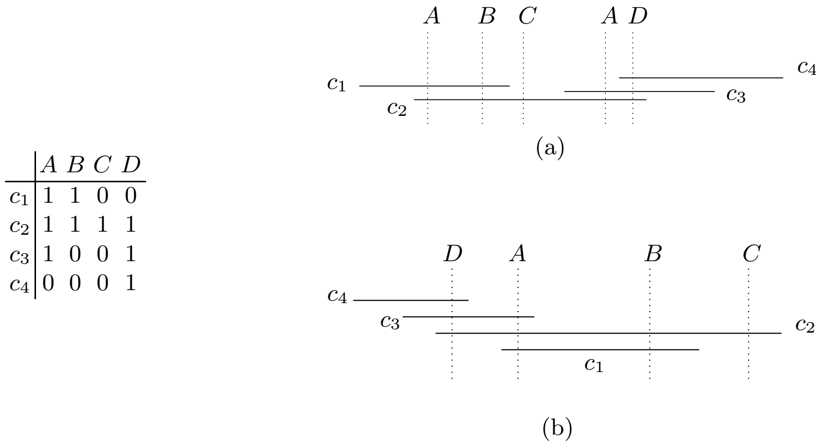
Definition 7.11. *The* problem of Mapping by Hybridization*, MbH for short, is given as follows.*

Input: An $(n \times m)$-hybridization matrix $\mathcal{H}$.

[10] This notation goes back to the duplication of molecules by means of cloning to obtain the large amount of identical molecules required for hybridization experiments.

Output: An ordering of clones and probes, respectively, that explains the data given by $\mathcal{H}$ as well as possible.

At this level, the above definition is, due to the term "explains ... as well as possible", quite informal, and from a mathematical point of view not easy to handle. Therefore, we present an example in Figure 7.10 to get a deeper understanding about the correspondence between an ordering of clones and probes, along with the hybridization matrix.



	A	B	C	D
c_1	1	1	0	0
c_2	1	1	1	1
c_3	1	0	0	1
c_4	0	0	0	1

Fig. 7.10. Hybridization matrix and two derived orderings of clones or probes. Both orderings will lead to the given hybridization matrix. Note that in ordering (a) the probe A occurs twice and in ordering (b) the clone c_1 is completely covered by clone c_2

At this point it is useful to recall that a hybridization matrix is derived from experimental data and we thus have to account for errors in this context. Therefore, in the above definition, we ask for an explanation of the data from $\mathcal{H}$ that is as good as possible, i.e., for a good ordering of clones (or probes). We distinguish between the usual error types for DNA chips, as *false positive* and *false negative*. If a position (i, j) of a hybridization matrix contains the value 1 even though probe p_j is not a substring of clone c_i, i.e., if there is no hybridization between p_j and c_i, then a *false positive* hybridization error has occurred at this position. If, on the other hand, no entry occurs at a certain position of the hybridization matrix, though the corresponding clone contains the corresponding probe as a substring, the error type is called *false negative*.

To evaluate the performance of the following models and algorithms for the mapping problem, we have to keep an eye on the errors in hybridization matrices. Nevertheless, we consider *idealized* hybridization matrices here, whose data is reliable, i.e., error free.

To specify the definition of the problem of mapping by hybridization, we distinguish two cases concerning the frequency of occurrences of probes. In the first case we assume that each probe appears only once within the investigated DNA. In this context, the term *mapping with unique probes* is used. Subsequently, we will also deal with the case where probes may appear several times along the DNA, referred to as *mapping with non-unique probes.*

7.2.1 Mapping with Unique Probes

Before we formally present the problem setting and develop approaches for the mapping with unique probes, we should first justify our assumption of the *uniqueness* of the probes. In experiments following this approach, often so-called *STS probes* are used. The acronym STS derives from the term *sequence-tagged sites.* These special probes are mostly extracted from the clones themselves and are of a length that they can be assumed to occur at most once along the DNA with high probability. We usually do not have a single probe in the sense of a connected nucleotide sequence, but rather a pair of substrings of 18 bp that are separated by 200 to 1000 bp. Since we can consider the resulting special data analogously to our previous setting, we abstract from this pair representation and use our usual model described above.

How can we now envision our problem in the context of unique probes? We are looking for a permutation (remember we are dealing with unique probes) of the probes that corresponds to their real ordering along the DNA. We know that each probe appears exactly once in our DNA sequence. If, for instance, the correct ordering of probes is $p_1, p_2, \ldots, p_m$, this implies that no clone can hybridize with probes p_i and p_{i+2} without hybridizing with probe p_{i+1}. At the level of the hybridization matrix, this corresponds to the fact that, for a correct ordering of the probes, i.e., the columns, only a consecutive interval of 1s can occur for each clone in its corresponding row. This property of the matrices will be formalized by the following definition.

Definition 7.12. *Let A be an $(n \times m)$-matrix with entries from $\{0, 1\}$. We say that A has the* consecutive ones property (C1P), *if there exists a permutation π of the columns of A such that in each row no 0 occurs between two 1s, i.e., for all rows $i = 1, \ldots, n$ the following holds:*

If $A(i, \pi(k)) = 1$ and $A(i, \pi(l)) = 1$ for $\pi(k) < \pi(l)$,
this implies that $A(i, j) = 1$ for all $j \in \{\pi(k) + 1, \ldots, \pi(l) - 1\}$.

If A already shows this special form, i.e., if π is the identical permutation, we say that A is in consecutive ones form (C1F).

Thus, we can formulate our goal as follows. Since, according to our assumption, our input hybridization matrix $\mathcal{H}$ does not contain any errors, we know that a valid ordering of the probes exists. Our goal is to find a corresponding permutation of the columns (probes) that transfers $\mathcal{H}$ into C1F.

How we can finally gain the ordering of the fragments from a C1F representation is demonstrated in Figure 7.10, where the C1F representation of the given hybridization matrix has the following form.

	D	A	B	C
c_1	0	1	1	0
c_2	1	1	1	1
c_3	1	1	0	0
c_4	1	0	0	0

The sequence of probes is thus changed from A, B, C, D to D, A, B, C. This corresponds to a column permutation $(4, 1, 2, 3)$.

It remains for us to actually compute the C1F representation of a given hybridization matrix. So, in what follows, we present an algorithm that determines whether a matrix has the C1P and, if so, computes a permutation of the columns corresponding to a C1F.

If we want to prove that a hybridization matrix satisfies the C1P, a possible approach would be to list all permutations of the columns (i.e., the probes), to apply each of these to the matrix, and to subsequently check whether the resulting matrix is in C1F. This naive approach will obviously not lead us to our goal, since there exist $n!$ many different permutations of n columns, which are impossible to check efficiently for realistic values of n.

Another useful requirement for our algorithm checking for the C1P, other than that it should be efficient, would be for it to compute and output *all* column permutations that transfer the hybridization matrix into C1F. On the other hand, the number of these valid permutations may also be quite large, which would lead to a loss of efficiency.

Nevertheless, despite of all these difficulties, in this case we can actually achieve our goal by the following idea of representing sets of permutations in a clever way. We now introduce a related data structure, the so-called *PQ-trees.*[11]

Definition 7.13. *Let $U = \{u_1, \ldots, u_n\}$ be a finite set of elements. A PQ-*tree *on U is a structure $T = (V, E, r, B, label, type)$ such that*

(i) (V, E) is an ordered tree rooted at $r \in V$,
(ii) $B \subseteq V$ is the set of leaves from (V, E),
(iii) $label: B \to U$ is a bijection from the leaves to U, and
(iv) $type: V - B \to \{P, Q\}$ is a mapping of the inner vertices to either type P or type Q.

Each leaf in a PQ-tree is thus labeled with exactly one element from U and, conversely, each element from U is assigned to exactly one leaf. Each inner vertex of T is either of type P or Q; for simplicity, we call the vertices P-vertices and Q-vertices, respectively. (The exact meaning of P-vertices and Q-vertices is discussed later.)

[11] Recall the general definition of trees in Chapter 3.2.

Graphically, we represent P-vertices as cycles and Q-vertices as rectangles. An example of such a PQ-tree is given in Figure 7.11.

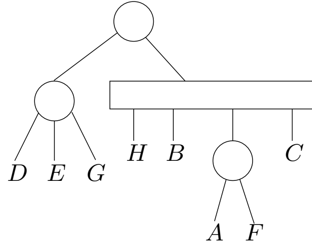


Fig. 7.11. Representation of a PQ-tree with three P-vertices, one Q-vertex, and leaves $A, \ldots, H$

PQ-trees represent a certain subset of permutations of their leaves and thus also of the set U. The restrictions with respect to the set of all permutations is thus given by the order of P- and Q-vertices within the tree. We first define the obvious permutation which we can directly infer from the order of the leaves in the graphical representation.

Definition 7.14. *Let U be a set, T be a PQ-tree on U, and $(v_1, \ldots, v_n)$ be the leaves in T according to their ordering from left to right.*[12] *We define the* front *of T as*

$$\textsc{Front}(T) = (label(v_1), \ldots, label(v_n)),$$

that is, the permutation of the elements of U obtained by reading the labels of the leaves from left to right.

A PQ-tree T now represents all permutations of elements from the set U that are fronts of trees reachable from T, if we are allowed to apply the following two operations to the inner vertices of T.

Definition 7.15.
Let T be a PQ-tree on a set $U = \{u_1, \ldots, u_n\}$. Then, we denote the following operations on T as legal operations*:*

(i) The ordering of the children of a P-vertex may be changed arbitrarily, and
(ii) the ordering of the children of a Q-vertices may be reversed, i.e., if $u_{i_1}, \ldots, u_{i_k}$ denotes the original ordering of the children (reading from left to right), it may be changed to $u_{i_k}, \ldots, u_{i_1}$. Other changes regarding the order of the children of a Q-vertex are not allowed.

[12] Recall that a PQ-tree is an *ordered* tree, i.e., for each vertex the order of its children is given.

We now define the set $\textsc{Consist}(T)$ of permutations induced by T and call them *consistent permutations of* T. Let $T \sim T'$ iff there exists a series of legal operations transforming T into T' for two PQ-trees T and T'; then,

$$\textsc{Consist}(T) = \{\textsc{Front}(T') \mid T \sim T\}.$$

A nice and illustrative way to describe the set of consistent permutations of a PQ-tree T is to think about T as mobile, fixed at the root vertex of T. While from each P-vertex only single threads originate that can be ordered in an arbitrary way, a Q-vertex may be seen as a strip of paperboard on which the single submobiles are fixed according to a certain order; by a rotation of 180° they can only reverse their ordering, and cannot arbitrarily change it.

Two PQ-trees T and T' are said to be *equivalent*, if they represent the same set of consistent permutations, i.e., if $\textsc{Consist}(T) = \textsc{Consist}(T')$.

Let us first consider the PQ-tree shown in Figure 7.12 consisting of only one P-vertex as the root and the elements $u_1, \ldots, u_n$ as leaf vertices. According to the legal operation (i) in Definition 7.15, this tree represents all possible permutations of $u_1, \ldots, u_n$ in a very compact way. We denote this special PQ-tree also as the *universal tree* over the elements $u_1, \ldots, u_n$. Furthermore, we will call the special PQ-tree having no vertices the *empty tree*.

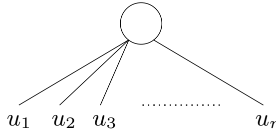


Fig. 7.12. Universal PQ-tree T_U on the set $U = \{u_1, \ldots, u_n\}$

To represent different sets of permutations that depend on restrictions according to the consecutiveness of certain elements using a PQ-tree, we proceed as follows.

Input: A set $U = \{u_1, \ldots, u_n\}$ of pairwise different elements and a set of restrictions $\mathcal{R} \in \text{Pot}(U)$. A restriction $R \in \mathcal{R}$ is thus a subset of U.

Output: All permutations π of elements in U such that for all restrictions $R \in \mathcal{R}$ the elements of R are consecutive in π.

To solve this problem we proceed in two steps.

1. We start with a universal PQ-tree T_U over the elements of U.
2. For each restriction $R \in \mathcal{R}$, we transform the current PQ-tree, such that the set of consistent permutations satisfies the restriction R.

We now describe the second step in more detail and call the transformation of a PQ-tree T according to a single restriction R a *reduction*. For such a reduction, the tree we want to reduce is traversed upwards starting at the

leaves, where certain rules are applied, if possible. Intuitively, these rules will describe the necessary transformations that are required to obtain a tree satisfying the restriction. We discuss this issue in greater detail later. First, we present Algorithm 7.2, which performs the transformation just described. For data management we use a queue, i.e., a special data structure that allows to remove first the elements that are inserted first (the FIFO principle — first in, first out). The operations for insertion and removal of elements in a queue are denoted as Enqueue and Dequeue, respectively. Such a queue thus simply corresponds to a waiting line at a cash register in a supermarket.

Algorithm 7.2 Reduce(T, R)

Input: A PQ-tree T on a set $U = \{u_1, \ldots, u_n\}$ and a restriction $R \subseteq U$.

1. L := queue of all leaves from T.
2. **while** L is not empty **do**

 2.1 x := Dequeue(L) {Extract an element x from queue L.}
 2.2 **if** x is a leaf **then**
  if $x \in R$ **then**
   mark x as *full*
  else
   mark x as *empty*
 2.3 **else if** at the subtree with root x a *rule* may be applied **then**
  apply the rule on T
 2.4 **else**
  output T := empty tree
 2.5 **if** $R \subseteq \{y \mid y \text{ is a descendant of } x \text{ in } T\}$ **then**
  output T
 2.6 **if** each sibling vertex of x was considered **then**
  y := parent of x
  Enqueue(L, y) {Insert parent vertex of x into L.}

Output: The PQ-tree T reduced according to R.

The rules referred to in Algorithm 7.2 always consist of a pattern-replacement pair, with the obvious meaning that if a *pattern* can be matched to the subtree considered, it is substituted with the *replacement*. It will turn out in the construction of the rules that if there exists an applicable rule in step 2.3 of the algorithm, the rule is unique.

If none of the rules can be applied, the algorithm outputs the empty tree to signal that no permutation exists, which, starting from the PQ-tree T, can satisfy the additional restriction R. Step 2.6 takes care that the algorithm traverses further upward to a parent vertex only if all of its children have already been processed.

In the following, we describe the rules applied in step 2.3 in greater detail. For this, we have to give some notations first. With respect to a reduction for a restriction R, we call a vertex x of the PQ-tree T considered

- *full*, if it is or all its descendent vertices are included in R,
- *empty*, if neither it nor any of its descendent vertices is included in R, or
- *partial*, if some, but not all, of its descendent vertices are included in R.[13]

Vertices of T that are either *full* or *partial* are called *pertinent.* The *pertinent subtree of T with respect to R,* $\text{PERTINENT}(T, R)$, is the subtree of minimal height of T, whose front contains all elements from R, i.e., $R \subseteq \text{FRONT}(\text{PERTINENT}(T, R))$. By this, the pertinent subtree as well as its root vertex, $\text{ROOT}(T, R)$, are uniquely determined.

We now describe the single rules graphically. As usual, we will represent P-vertices by circles and Q-vertices by rectangles. Child vertices will be depicted by triangles independent of their types. To distinguish between a *full* and an *empty* vertex, we shade the *full* vertices. The figures show the patterns on their left-hand sides and the corresponding replacements on their right-hand sides.

First, we describe the replacement rules for the case where the current vertex x considered in step 2 of Algorithm 7.2 is of type P. If all its children are *empty* nothing happens; if they are all *full*, we propagate this information to vertex x and mark x as *full* as well (see Figures 7.13 and 7.14).

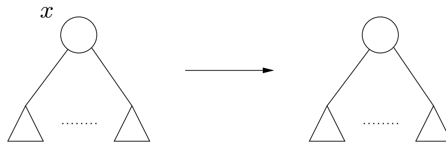


Fig. 7.13. Rule $P.0$: If all children are *empty*, nothing happens

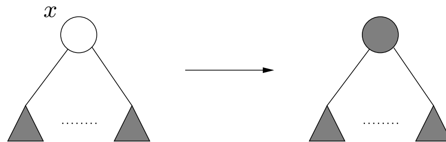


Fig. 7.14. Rule $P.1$: If all children are *full*, the considered root is marked *full*

If only some of the children of the P-vertex considered are *full*, none are *partial*, and the current vertex is also the root of the pertinent tree, i.e.,

[13] The following description of the rules will show that this predicate will be assigned to Q-vertices only.

$x = \text{Root}(T, R)$, then the *full* child vertices are joined together by means of a new P-vertex and thus separated as a group from the *empty* vertices (see Figure 7.15). One should note that, using legal operation (i), all children of x can be ordered in the way shown in the figure. The rule will be applied accordingly.

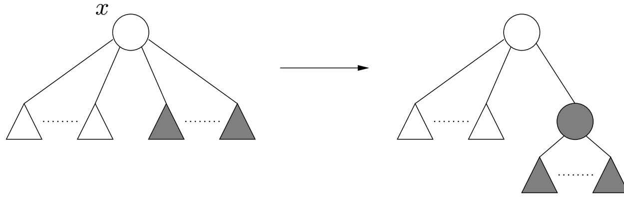


Fig. 7.15. Rule $P.2$: A subset of the child vertices is *full*, none are *partial*, and the considered P-vertex x is the root of the pertinent subtree

In the case where the considered P-vertex x is not the root of the pertinent subtree, the *partial* vertices come into consideration. Graphically, we will shade the *partial* vertices only partly, so that shadows indicate the side where *full* child vertices appear.

If x is not the root of the pertinent subtree, its *full* children must be grouped in such a way that they can, by applying other rules on the other *full* vertices,[14] appear in a consecutive series afterwards. Therefore, we utilize a newly added *partial* Q-vertex (see Figure 7.16).

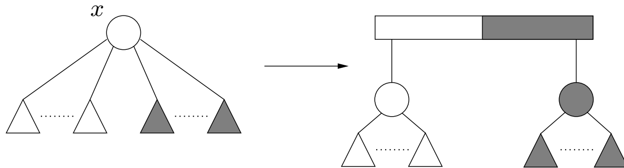


Fig. 7.16. Rule $P.3$: A subset of the children is *full*, none are *partial*, and the considered P-vertex x is not the root of a pertinent subtree

Next, we consider the cases where the P-vertex x under consideration has exactly one *partial* child vertex. Such a situation may directly arise from the application of rule $P.3$. We distinguish again two cases depending on whether x is the root of the pertinent subtree or not. The corresponding replacement rules $P.4$ and $P.5$ are shown in Figures 7.17 and 7.18, respectively. In essence, they correspond to a generalized version of rules $P.2$ and $P.3$.

[14] There must exist other *full* vertices, since otherwise x would be the root of the pertinent subtree.

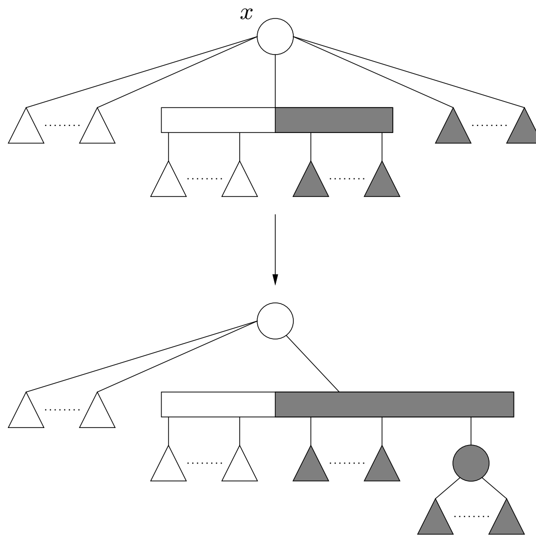


Fig. 7.17. Rule $P.4$: Exactly one child of x is *partial* and x is the root of the pertinent subtree

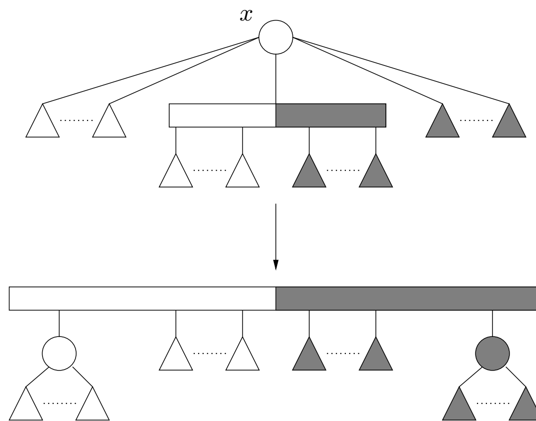


Fig. 7.18. Rule $P.5$: Exactly one child of x is *partial* and x is not the root of the pertinent subtree

Thus, the only remaining case is where the considered P-vertex has exactly two *partial* child vertices (see Figure 7.19). If a vertex has more than two *partial* children, one can easily conclude that the restriction cannot be satisfied, because then *empty* vertices would have to appear between *full* vertices. The same holds for the case where the considered vertex x has two *partial* children and is not the root of the pertinent subtree.

It hence suffices to specify one rule for the handling of two *partial* child vertices of x, where x is the root of the pertinent tree. All other cases, for which no rules exist, will be rejected by the algorithm by giving the empty PQ-tree as output.

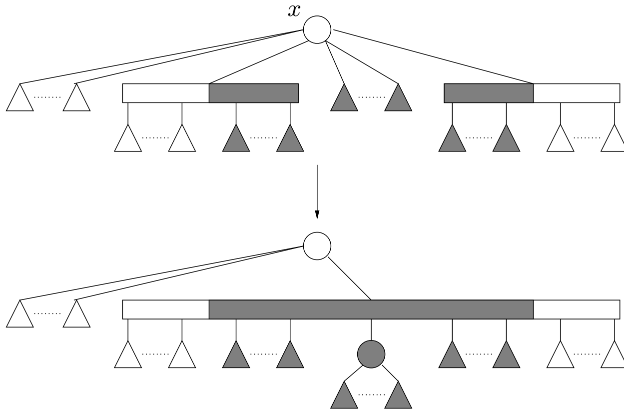


Fig. 7.19. Rule $P.6$: Two children of x are *partial* and x is the root of the pertinent subtree

At this point, we have discussed all cases where the considered vertex was of type P. In all cases not stated explicitly, the demanded restriction is not applicable, since *full* and *empty* vertices would be inevitably nested into one another. This does not satisfy the demand of the restriction.

Next, we consider vertices of type Q. Analogously to the rules for P-vertices with only *empty* or only *full* child vertices, the rules $Q.0$ and $Q.1$ apply for Q-vertices, as shown in Figures 7.20 and 7.21, respectively.

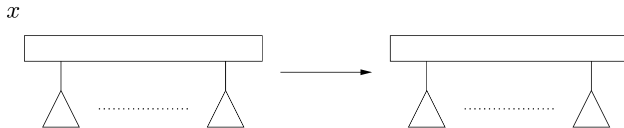


Fig. 7.20. Rule $Q.0$: If all children are *empty*, nothing happens

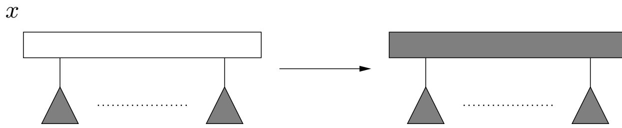


Fig. 7.21. Rule $Q.1$: If all children are *full*, the considered root is marked *full*

Of interest now are the remaining cases, in which the Q-vertex considered has different (with respect to their being *empty, full,* or *partial*) child vertices. In a generalized way, we only consider cases with at most one or two *partial* child vertices. Similarly to the discussion in the context of P-vertices, no other possibilities exist, since otherwise the restriction cannot be satisfied. The resulting rules are shown in Figures 7.22 and 7.23.

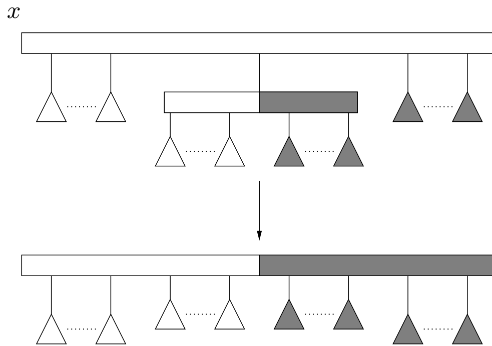


Fig. 7.22. Rule $Q.2$: At most one child of x is *partial*

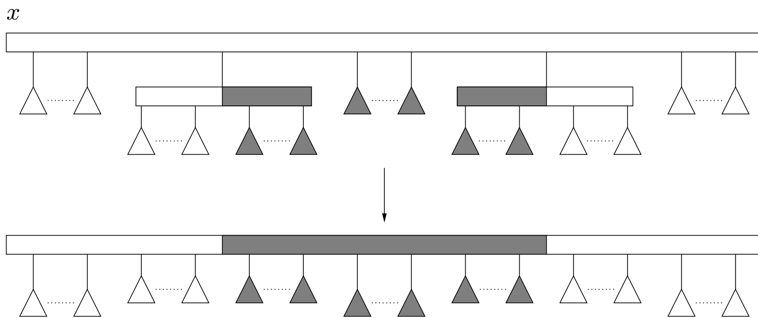


Fig. 7.23. Rule $Q.3$: Two children of x are *partial* and x is the root of the pertinent subtree

With the rules $P.0$ to $P.6$ and $Q.0$ to $Q.3$ we have completed the description of Algorithm 7.2 and are now able to present an algorithm for the consecutive ones problem.

Algorithm 7.3 Consecutive ones

Input: An $(n \times m)$-matrix M with entries from $\{0, 1\}$.

1. $U :=$ the set of column indices of M
2. $T :=$ universal PQ-tree on U
3. **for** $i = 1$ **to** n **do**
   $R := \{u \in U \mid M(i, u) = 1\}$ {Indices of columns that have the entry 1 in row i}
   $T :=$ Reduce(T, R)
   if $T =$ empty PQ-tree **then**
   Output: "M does not satisfy the C1P."
4. **Output:** "M satisfies the C1P and is in C1F for all permutations in Consist(T)."

Theorem 7.5. *Let M be an $(n \times m)$-matrix with entries from $\{0, 1\}$ and let k be the number of 1s in M. Then Algorithm 7.3 solves the consecutive ones problem in time $O(n + m + k)$.* □

The proof of this theorem requires, besides other things, a skillful implementation of the replacement rules, which would be beyond the scope of this chapter. Therefore, we omit the proof of Theorem 7.5 and illustrate the strategy of Algorithm 7.3 by an example instead. To do so, we consider the hybridization matrix given in Figure 7.24 and describe the work of Algorithm 7.3 and the application of the single rules to this matrix in Figures 7.25 to 7.29.

	A	B	C	D	E	F	G	H
I	1	1	0	0	0	1	0	0
II	0	0	0	1	1	0	1	0
III	0	1	0	0	0	0	0	1
IV	1	1	1	0	0	1	0	0
V	0	0	0	1	0	0	0	1

Fig. 7.24. Hybridization matrix $\mathcal{H}$ with set $P = \{A, B, C, D, E, F, G, H\}$ of probes and set $C = \{\text{I}, \text{II}, \text{III}, \text{IV}, \text{V}\}$ of clones

We start with the universal PQ-tree on elements $\{A, \ldots, H\}$ and subsequently consider the restrictions R_i with $i \in \{\text{I}, \ldots, \text{V}\}$ enforced by the rows of $\mathcal{H}$ successively. As in the replacement rules above, we will mark the vertices

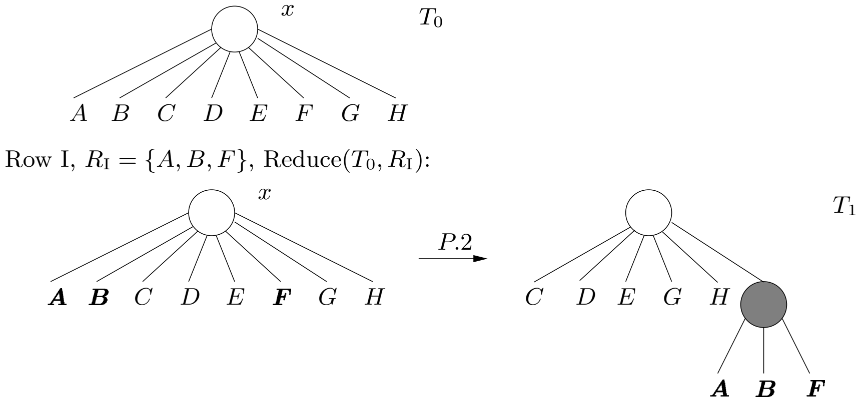


Fig. 7.25. Execution of Algorithm 7.3 on the hybridization matrix from Figure 7.24 (row I)

by different shadings as *empty*, *full*, or *partial*. The leaves, i.e., the elements from $\{A, \ldots, H\}$, are marked as *full* by setting the characters to bold face, and the vertex under consideration is denoted by x. Moreover, one should keep in mind that, in addition to the application of the rules, legal operations on the current PQ-trees may be performed to fit the trees to the pattern. Furthermore, inner vertices with only one successor do not make sense, and that is why we merge such vertices into a single one in this example.

Clearly, if the algorithm leads to several possible orderings of the probes, then we cannot unambiguously reconstruct the original ordering of probes along the DNA molecule on the basis of the hybridization matrix only.

7.2.2 Mapping with Unique Probes and Errors

In the previous section we discussed the consecutive ones problem as a possible model for mapping by hybridization with unique probes on *error-free* data. Here, we present some ideas how to handle the corresponding problem on data that is *error prone*.

For this, we investigate how the entries of the hybridization matrix will change due to different error types. Let us assume we know the ordering of the columns, that is, the ordering of the probes in the original DNA sequence, and that the columns in our given hybridization matrix $\mathcal{H}$ are ordered according to this. If the data represented by $\mathcal{H}$ is error free, then the hybridization matrix $\mathcal{H}$ has consecutive ones form. Now, let us consider the case where errors occur while experimentally determining the hybridization data.

- If a *false negative error* arises, then within the hybridization matrix a 0 occurs at a position where there should be a 1. If this error does not take

Row II, $R_{\mathrm{II}} = \{D, E, G\}$, Reduce$(T_1, R_{\mathrm{II}})$:

Fig. 7.26. Execution of Algorithm 7.3 on the hybridization matrix from Figure 7.24 (row II)

place at the border of a series of consecutive 1s in $\mathcal{H}$, but somewhere in the middle, then the series of 1s is interrupted by a 0 and split into two blocks. We call this break in the contiguous series of 1s a *gap*.

$$0\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 0\ 0$$
$$\uparrow$$

- If, on the other hand, a *false positive error* occurs, and this does not by chance flip a 0 into a 1 at the border of a series of 1s within a row of the hybridization matrix, then a new block of 1s emerges that consists of a single 1 and is separated from the other block of 1s by a gap:

$$0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 0$$
$$\uparrow$$

- Another error not considered so far is caused by so-called *chimeric clones*. By this we mean the joining of two fragments that have not been adjacent to each other in the original DNA into a single fragment. Thus, we have two parts of DNA that appear to be consecutive, although they are actually not. Such chimeric clones may, for instance, occur during the duplication of DNA by means of host organisms as described in Section 2.4.2. At the level of the hybridization matrix, where we again assume a correct ordering of the columns, a chimeric clone may lead to the situation where two blocks of 1s are separated by a gap of multiple 0s, for these blocks actually represent single fragments of the DNA that are melded together and are therefore considered as a single fragment in the hybridization matrix, i.e., as a single row in the matrix:

Row III, $R_{\mathrm{III}} = \{B, H\}$, Reduce$(T_2, R_{\mathrm{III}})$:

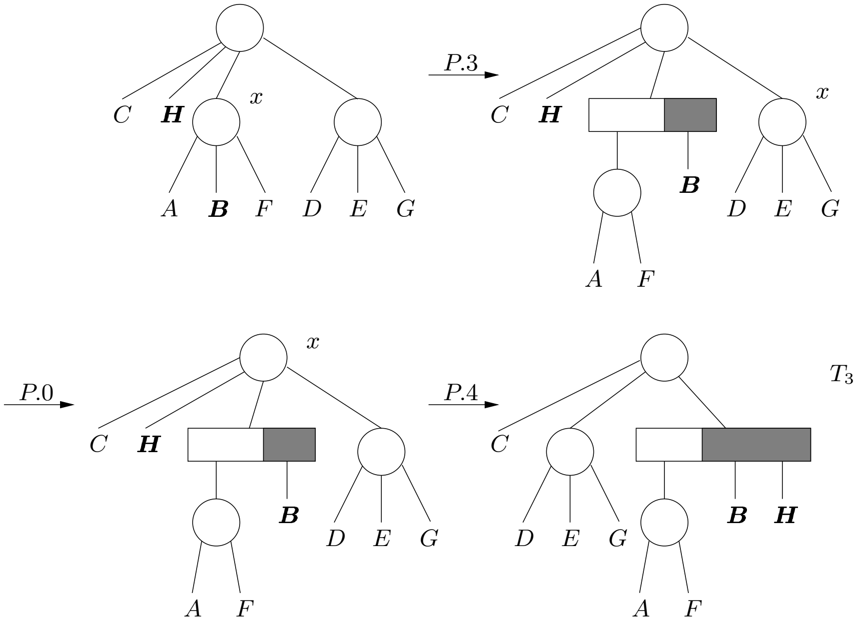


Fig. 7.27. Execution of Algorithm 7.3 on the hybridization matrix from Figure 7.24 (row III)

$$0\;0\;1\;1\;1\;1\;0\;0\;0\;0\;1\;1\;1\;1\;0$$
$$\uparrow\,\uparrow\,\uparrow\,\uparrow$$

Dealing with errors, we usually apply a general parsimony principle. So, even if there might be errors in our hybridization data, we assume that the number of errors with respect to the correctly determined data is relatively small. According to our previous discussion of some different error types and the resulting gaps in the hybridization matrix with a correct ordering of columns (probes), it is meaningful to try to minimize the number of gaps between blocks of consecutive 1s in the hybridization matrix.

Definition 7.16. *The* gap minimization problem, GMINP *for short, is the following optimization problem.*

Input: An $(n \times m)$*-matrix* A *with entries from* $\{0, 1\}$.
Feasible solutions: For all inputs A,

$$\mathcal{M}(A) = \{(i_1, \ldots, i_m) \mid (i_1, \ldots, i_m) \textit{ is a permutation of } (1, \ldots, m)\}.$$

Costs: For all feasible solutions $\pi = (i_1, \ldots, i_m) \in \mathcal{M}(A)$,

$$\mathit{cost}(\pi, A) = \textit{Number of gaps included in the matrix that results from the application of the column permutation } \pi \textit{ on } A,$$

Row IV, $R_{\mathrm{IV}} = \{A, B, C, F\}$, Reduce$(T_3, R_{\mathrm{IV}})$:

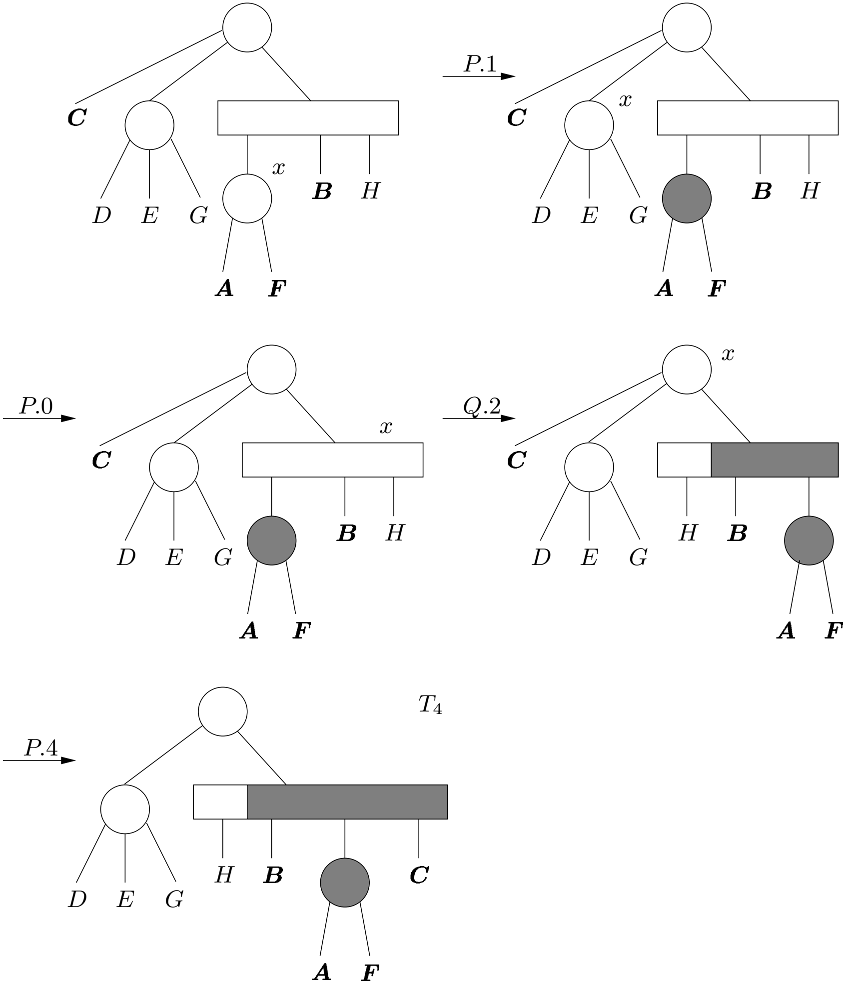


Fig. 7.28. Execution of Algorithm 7.3 on the hybridization matrix from Figure 7.24 (row IV)

where a gap *denotes a block of consecutive* 0*s within one row of the matrix that is bordered by* 1*s.*
Optimization goal: Minimization.

If the the data contains no error, GMinP corresponds to determining a column permutation such that no gaps occur in the resulting matrix. In other words, it is equivalent to determine the C1F of a matrix and hence meets the modelling of the error-free case of the previous section. For convenience, we assume for the rest of this section that the considered hybridization matrix

Row V, $R_{\text{V}} = \{D, H\}$, Reduce(T_4, R_{V}):

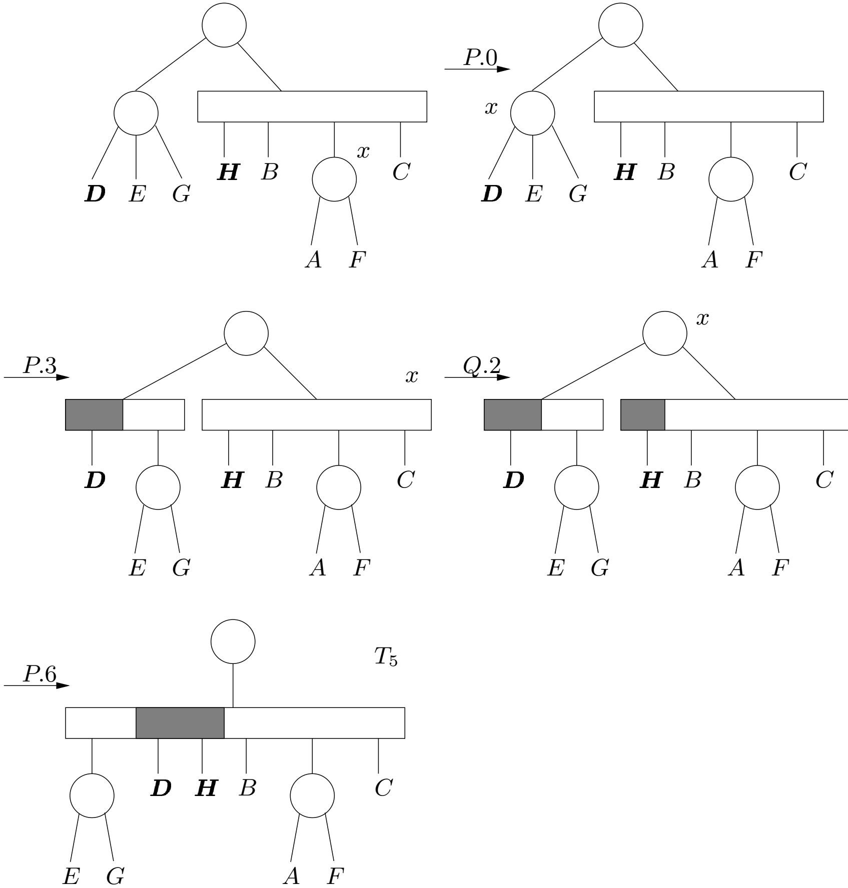


Fig. 7.29. Execution of Algorithm 7.3 on the hybridization matrix from Figure 7.24 (row V)

does not contain any row exclusively consisting of 0s. There would be no data that allows for a meaningful placement of a fragment corresponding to such a row. We can therefore a priori ignore such rows.

To further analyze the GMinP, we now deal with another representation of the problem, and represent parts of the input in terms of a weighted graph.

Definition 7.17. *Let A be a $(n \times m)$-matrix with entries from $\{0, 1\}$. Then the* column distance graph $G_\Delta = (V, E, c)$ *is an undirected edge-weighted graph with*

- $V = \{1, \ldots, m\}$
- $E = \{\{i, j\} \mid 1 \leqslant i, j \leqslant m, i \neq j\}$

- $c : E \to \mathbb{N}$, *where* $c(i,j) = |\{k \mid 1 \leqslant k \leqslant n, A(k,i) \neq A(k,j)\}|$

The column distance graph thus is a complete graph whose vertices correspond to the columns of the matrix and the edge-costs $c(i,j)$ denote the number of rows in which columns i and j differ. The distance given by function c between two columns of matrix A is also known as *Hamming distance*.

Let us now consider the situation where a gap occurs in a row k of matrix A. In this case at least one alternation from 1 to 0, between column i and $i+1$, and one alternation from 0 to 1, between column j and $j+1$, takes place:

$$\begin{array}{ccccccccc} 0\,0\,1\,1\,1\,1 & & 0 & & 0\,0\,0 & & 1 & & 1\,1\,1\,0 \\ & \uparrow & & \uparrow & & \uparrow & & \uparrow & \\ & i & & i+1 & & j & & j+1 & \end{array}$$

Now, let us consider the column distance graph G_Δ corresponding to matrix A and a path within this graph that corresponds to the order of the columns in A. Then, a gap in row k will contribute the value 1 to the edge cost of each of the edges $\{i, i+1\}$ and $\{j, j+1\}$. Accordingly, a gap contributes the value 2 to the total cost of the path[15].

Informally, we are able to rephrase GMinP as follows: Find a path within the column distance graph G_Δ of A that visits each vertex exactly once and has minimum cost.

At this point, a problem occurs. The first alternation from a 0 to a 1 within a row of the matrix contributes to the costs of the path as well, as does the last alternation. But these alternations are not caused by a gap. If we were able to guarantee that exactly two alternations of this kind would occur in each row, we would be able to solve our gap-minimization problem by finding the path of minimal cost as described above. Unfortunately, in general we cannot guarantee this property. Consider

(a) 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1

(b) 0 0 1 1 1 1 0 0 0 0 1 1 1 0 0

Both rows (a) and (b) have exactly *one* gap. But in row (a) only *two* alternations occur while in row (b) *four* such alternations appear.

In what follows, we show how we can enforce exactly two alternations not originating from a gap for each row. For this, we simply extend our matrix by a column that consists of 0s only. Furthermore, instead of considering paths we will switch to cycles in the column distance graph. In this way, we are able to prove the following theorem.

Theorem 7.6. *Let* A *be an* $(n \times m)$*-matrix with entries from* $\{0,1\}$*. Let* A' *be the* $(n \times (m+1))$*-matrix resulting from* A *after adding a specific* 0*-column* p'*.*

[15] As usual, the cost of the path corresponds to the sum of edge costs along the path.

G_Δ denotes the column distance graph belonging to A'. Let π be a permutation of the columns in A and let A_π be the resulting column-permuted matrix. Accordingly, let C_π be the cycle in G_Δ that visits the vertices starting with p', respecting permutation π, and finally coming back to p' again. The edge costs of this cycle now satisfy

$$cost(C_\pi) = 2 \cdot l + 2 \cdot n,$$

where l denotes the number of gaps in A_π.

Proof. The proof of this theorem is based on the idea that by introducing the special column p' we are able to break up the cycle C_π by simply deleting the vertex p'. So we obtain a path that exactly corresponds to permutation π.[16] On the level of the column distance graph, our procedure now guarantees two alternations at the border for each row, one from a 0 to a 1 and one from a 1 to a 0, respectively.

For the costs of the cycle C_π we now derive the following:

$$\begin{aligned} cost(C_\pi) &= \sum_{\{i,j\} \in C_\pi} c(\{i,j\}) = \sum_{\{i,j\} \in C_\pi} \sum_{k=1}^{n} \delta(A(k,i), A(k,j)) \\ &= \sum_{k=1}^{n} \sum_{\{i,j\} \in C_\pi} \delta(A(k,i), A(k,j)), \end{aligned} \tag{7.1}$$

where $\delta(x,y) = 1$ iff $x \neq y$.

According to Equation (7.1), we can sum up the costs for the single rows separately. Let us therefore consider an arbitrary row k with $1 \leqslant k \leqslant n$, and let us additionally denote by $cost(k)$ the number of alternations from a 0 to a 1 and from a 1 to a 0, respectively, within this row, i.e., $cost(k) = \sum_{\{i,j\} \in C_\pi} \delta(A(k,i), A(k,j))$. Let l_k be the number of gaps in row k in matrix A_π. We know that for each gap in row k exactly two alternations occur. We further know that at the borders of row k exactly two alternations occur as well, either because the first column of the k-th row of A_π already contains a 0 or due to the alternation to the special column p'. The same similarly holds for the last column of A_π.[17] Thus, we have for the costs of each row k,

$$cost(k) = 2 \cdot l_k + 2. \tag{7.2}$$

By applying $\sum_{k=1}^{n} l_k = l$ and Equations (7.1) and (7.2), we obtain the claimed result. □

[16] Since a cycle in an undirected graph does not have a direction, the resulting path may also correspond to the reverse permutation of π. We will omit this case here, since it is not essential to the claim of the theorem but would make the presentation of the proof unnecessarily tangled.

[17] Note that we have excluded rows solely consisting of zeros.

By this result we have shown that we can reduce the GMinP to finding a cycle in a complete, undirected, and edge-weighted graph that visits each vertex (except for the start and end vertices) exactly once and has minimum costs. But this is actually the well-known *traveling salesman problem* we already considered in Section 3.3.

Unfortunately, this is an NP-hard optimization problem, and therefore the above procedure does not lead to an efficient algorithm for our original GMinP problem. But GMinP is known to be NP-hard as well [79], which shows that our reasoning has at least not essentially complicated our problem.

So, let us further consider in more detail the TSP and, in particular, the instances arising from our construction. The structure of the inputs of the TSP we obtained by our reduction of GMinP may possibly show some properties that help us simplify the computation of solutions for these instances. Analyzing the structure of input instances is a general technique that might help simplify hard problems and achieve a deeper understanding of the reasons for their hardness.

Indeed, we can recognize in our case that the cost function implied by the column distance graph G_Δ satisfies the *triangle inequality*, i.e., for all vertices v_1, v_2, v_3 in G_Δ,

$$c(\{v_1, v_3\}) \leqslant c(\{v_1, v_2\}) + c(\{v_2, v_3\}).$$

On an intuitive level, the triangle inequality implies that using a detour cannot be cheaper than using the direct path in the graph.

The cost function of the column distance graph corresponds to the Hamming distance between the single columns; therefore, it is easy to ensure that this cost function indeed obeys the triangle inequality.

Let us, at this point, summarize what we have so far undertaken to address the problem of physical mapping with unique probes and errors. By considering different types of errors we motivated our modelling of the problem in terms of the GMinP. Afterwards, we showed that we can solve GMinP by using the TSP. Unfortunately, the TSP in general is not only an NP-hard optimization problem, but it is furthermore impossible to approximate it by any factor, unless $P = NP$. Therefore, we investigated the input instances resulting from our reduction of GMinP to TSP in more detail and found out that the cost function of the constructed column distance graph satisfies the triangle inequality.

We considered the TSP with triangle inequality (Δ-TSP) in Section 3.3 and presented a 2-approximation algorithm (Algorithm 3.1) for it. A $\frac{3}{2}$-approximation algorithm is also known, whose solutions are thus guaranteed to be at most 1.5 times more expensive than the optimal one [43]. Moreover, a huge number of heuristic algorithms exists for the Δ-TSP that cannot give any performance guarantee, but that are used in practice, since they achieve reasonably good results in many cases. References to some of the heuristics can be found in Section 7.4 at the end of this chapter.

With this reference to algorithms for solving the Δ-TSP we conclude our considerations concerning the GMinP and finally discuss some problems in the modelling of physical mapping with unique probes and errors.

For instance, the question arises whether a column permutation, i.e., an ordering of the probes, which obtains a minimal number of gaps in the matrix, but leads to a result where all rows except one consist of a single block of 1s (but where the remaining row has the form $\ldots 01010101010101 \ldots$) is really meaningful. This is clearly not what we desire, since it seems to be rather unlikely that no fragment shows any error, except for one, where instead a huge number of errors occur. Furthermore, thinking it over, it comes to mind that not only should the distribution of gaps within the matrix follow some reasonable assumptions, but the resulting blocks of 1s in the single rows should not become too small.

Again, all models are only approximating the biological problems; their usefulness has to be verified in practice, and their hypotheses must be tested experimentally.

7.2.3 Mapping with Non-unique Probes

After having presented a procedure for obtaining the original ordering of the probes (and thus also of the clones) from hybridization data based on unique probes, we will now discuss some of the issues arising from hybridization data based on *non-unique* probes. That is, we do not assume that each probe occurs only once in the investigated DNA strand.

We present one possible model in this framework. For this, we again assume that the given hybridization data is error free. Then we search for the ordering along our investigated DNA. On the one hand, probes may occur multiple times; on the other hand, the probes hybridizing with a certain clone should occur consecutively in the desired ordering. Let us assume that we have given the hybridization data between clones $c_1, \ldots, c_n$ and probes $p_1, \ldots, p_m$. Let s be a string over the alphabet of probes, i.e., over $\Sigma_P = \{p_1, \ldots, p_m\}$. We say that s *covers* a clone c_i if s contains a substring t that solely consists of probes that hybridize with c_i, where the ordering of probes and their multiplicity in t is irrelevant.

We now assume that probes along the DNA may indeed occur multiple times, but rather infrequently. Thus, we search for the shortest string s covering all clones. This leads to the following optimization problem:

Definition 7.18. *The* shortest covering string problem *is the following optimization problem.*

Input: An $(n \times m)$-matrix A with entries from $\{0, 1\}$.
Feasible solutions: For all inputs A,

$$\mathcal{M}(A) = \{s \in \{1, \ldots, m\}^* \mid \textit{for all } 1 \leqslant i \leqslant n \textit{ there exists a substring } t \textit{ of } s \textit{ such that } \operatorname{char}(t) = \{j \mid A(i,j) = 1\}\},$$

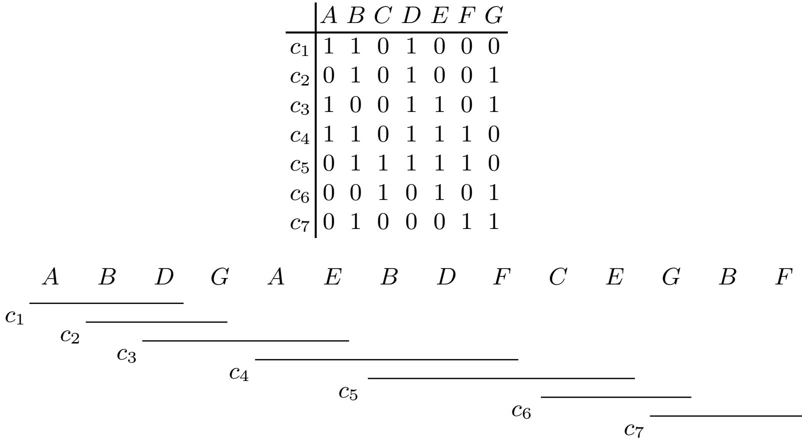
where char(t) *denotes the set of characters occurring in* t. *Hence,* $\mathcal{M}(A)$ *corresponds to the set of all possible strings* s *over the alphabet of probes covering every clone.*

Cost: For all feasible solutions $s \in \mathcal{M}(A)$*:*

$$cost(s, A) = |s|.$$

Optimization goal: Minimization.

This again is a hard combinatorial problem, and we do not consider approaches for addressing this problem here. Instead, we illustrate the problem setting with the example given in Figure 7.30.



	A	B	C	D	E	F	G
c_1	1	1	0	1	0	0	0
c_2	0	1	0	1	0	0	1
c_3	1	0	0	1	1	0	1
c_4	1	1	0	1	1	1	0
c_5	0	1	1	1	1	1	0
c_6	0	0	1	0	1	0	1
c_7	0	1	0	0	0	1	1

Fig. 7.30. Hybridization matrix and a feasible solution of the shortest covering string problem

Finally, at the end of this section we would like to note that also this modelling should be critically questioned, since some probes can indeed occur more frequently than others in the considered DNA, which would question the exclusive search for the shortest string only.

7.3 Summary

A physical map consists of a collection of known (short) sequences of nucleotides, called markers, and their locations along the considered DNA molecule. For constructing these maps several approaches are known, leading to various kinds of combinatorial problems.

In restriction site mapping, restriction enzymes are used to cut the DNA molecule into smaller fragments. In this process, their ordering gets lost and

has to be reconstructed. In doing so, the positions of the restriction sites along the molecule are determined, and hence restriction sites may serve as markers.

One approach in this context is that of the double digest. Here, in three different attempts, copies of the examined DNA are digested (i.e. cut) by two different restriction enzymes as well as by a combination of both. Afterwards, the lengths of the resulting fragments are determined, and the original ordering of the fragments is reconstructed using these lengths. This problem is called the double digest problem (DDP). It turns out that it is algorithmically difficult to handle, since on the one hand, the decision variant of the DDP is NP-hard and, on the other hand, numerous different solutions can often be derived from the given data. Thus, appropriate heuristics are often used to address this problem.

In contrast, the partial digest approach uses only a single restriction enzyme, but tries to derive all fragments that are bordered by either restriction sites or the ends of the molecule, varying the time of exposing copies of the molecule to the restriction enzyme. This means that, in contrast to the double digest approach, the resulting fragments can still contain restriction sites. Therefore, we call this type of action a *partial digest*, distinguishing it from the *full digest* of the double digest approach. In the partial digest approach, the lengths of the fragments are again measured, which ideally leads to a multiset of exactly $\binom{k}{2}$ elements, i.e., the multiset of all pairwise distances between restriction sites (or endpoints) of the molecule, where $k-2$ denotes the number of restriction sites along the molecule. The resulting combinatorial problem is known as the partial digest problem (PDP). There are no results concerning the NP-hardness of PDP so far, but a polynomial-time algorithm has not been discovered yet. Besides the naive approach of testing all possible orderings, there exists a backtracking algorithm, achieving good results in practice, although in the worst case the algorithm was shown to require exponential running time.

A second approach for the construction of physical maps depends on hybridization experiments. For physical mapping in general, the examined DNA is decomposed into fragments and for each of these fragments a most possibly characteristic fingerprint is computed. Fingerprints then serve as the data to reconstruct the ordering of the fragments (which is our desired task). Now, overlapping fragments should have similar fingerprints, and nonoverlapping fragments should have different fingerprints. In mapping by hybridization, the fingerprints consist of information about whether a fragment hybridizes with a certain known nucleotide sequence, called a probe, or not. In this kind of experiments, fragments are also called clones. Essentially, two types are distinguished.

If the probes are unique in the considered DNA molecule, the reconstruction problem can be reduced to the question of whether the hybridization matrix, representing the result of the hybridization experiment for each clone-probe pair, can be transformed by permuting its columns to obtain consecutive blocks of 1s in each of its rows. This problem has been addressed in the

context of some graph-theoretic problems, and a polynomial-time algorithm is known that uses PQ-trees as fundamental data structures. To incorporate experimental data errors, this model can be weakened, and one may want to minimize the number of blocks of consecutive 1s in the permuted matrix. This leads to an NP-hard optimization problem that can be represented as a variant of the well-known traveling salesman problem (TSP), the metric TSP, where edge costs are required to obey the triangle inequality. Efficient approximation algorithms are known to solve this problem.

If one allows for multiple occurrences of probes along the considered DNA molecule, the corresponding problem may be modeled in terms of the shortest covering string problem, where the goal is to minimize the multiplicity of different probes in the reconstruction of their order.

7.4 Bibliographic Notes

The subject of physical mapping is addressed with different focus in the textbooks by Pevzner [159], Setubal and Meidanis [180], and Waterman [201].

The first physical mapping by means of restriction enzymes was published by Danna et al. [56]. In 1987, Goldstein and Waterman examined the DDP, and showed that it is NP-hard and that the number of feasible solutions may increase exponentially with the number of restriction sites [84]. Approaches to join a multitude of feasible solutions to equivalence classes of solutions were considered by Schmitt and Waterman [176] and Pevzner [158]. Heuristics to solve the DDP are proposed in the book by Waterman [201]. Cieliebak et al. [46] investigate the complexity of variants of the DDP with respect to error-prone data.

An analogy of the PDP arises in the area of computational geometry and is known there as the *turnpike reconstruction* problem. The backtracking algorithm presented in Section 7.1.2 was originally presented by Skiena et al. [182] to solve this problem. An example, where this algorithm needs exponential running time was proposed by Zhang [210]. Skiena and Sundaram [183] studied an extension of the backtracking algorithm also to address error-prone data. Furthermore, a pseudopolynomial algorithm for the PDP is known, i.e., an algorithm which runs in polynomial time with respect to the size of the largest distance in the given multiset (see Rosenblatt and Seymour [168]). Cieliebak et al. proved in [47] that for disturbed data PDP is NP-hard. An overview of complexity results concerning these problem variants of restriction site mapping can also be found in the PhD thesis by Cieliebak [45].

A quite recent approach in the area of physical mapping by means of restriction enzymes is *optical mapping*. Here, copies of the investigated DNA are stretched upon an object slide below a microscope and subsequently digested by restriction enzymes. The fragments of the stretched DNA molecules may contract again, which causes gaps in the DNA strand that are visible under the microscope. In this approach, the ordering of fragments remains

unchanged but other combinatorial problems may arise, some of which are described by Karp and Shamir [117].

There is a multitude of papers concerning the subject of physical mapping by means of hybridization data. For an introduction, we refer the reader to the work of Alizadeh et al. [8, 9], Greenberg and Istrail [87], and the books already mentioned above. As early as 1965, Fulkerson and Gross [77] presented an algorithm to decide whether a matrix has the consecutive-ones property or not, which is very well demonstrated in the book by Setubal and Meidanis [180]. The use of PQ-trees to solve this task was proposed in the recommendable article by Booth and Lueker [36]. To model the combinatorial problems arising in this framework, *interval graphs* were often used. An introduction to the theory of interval graphs and their relation to the consecutive-ones property of matrices is given in the textbook by Golumbic [85].

The representation of the problem of physical mapping with unique probes and errors, as well as the connection of this problem to the Δ-TSP, originates from the book by Setubal and Meidanis [180]. The reference to the NP-hardness of GMinP can be found in the book by Garey and Johnson [79] as problem (SR17). The book by Hromkovič [105] includes, besides a very nice overview on algorithmic methods for solving hard optimization problems, also a multitude of approaches to address the TSP with triangle inequality. A more comprehensive overview on techniques applied to the TSP can be found in [130].

The modelling of mapping with non-unique probes in terms of the shortest covering string problem was proposed by Alizadeh et al. [8].

8

DNA Sequencing

In this chapter we will study the process of sequencing DNA and nucleotide sequences in general, i.e., of determining the sequence of bases along a nucleic acid molecule. Throughout this chapter, we will consider the DNA as a standard example of nucleic acids.

As we have already seen in Chapter 2, it is possible to determine the sequence of a DNA molecule by means of the chain termination method, which is also referred to as direct sequencing. By today's methods, this technique is applicable to read about 1 000 consecutive base pairs. Unfortunately, for longer molecules this method becomes too error prone to give reliable statements. On the other hand, our practical applications require the sequencing of molecules of length up to 50-300 kbp. To close this gap between requirements and possibilities enabled by sequencing techniques based on the chain termination method, different experimental methods have been developed, which in turn lead to various combinatorial questions.

This chapter is devoted to the presentation of two of these experimental techniques, known as *shotgun sequencing* and *sequencing by hybridization*. In particular, the first procedure represents a widely used approach for sequencing of DNA and leads to various combinatorial problems we study in more detail in Section 8.1. Afterwards, we take a closer look at the concept of sequencing by hybridization in Section 8.2. To conclude, a summary and some bibliographic notes on the topic of DNA sequencing are given in Section 8.3 and Section 8.4, respectively.

8.1 Shotgun Sequencing

The process of shotgun sequencing depends on the following idea. Since the investigated DNA molecule is too long to be sequenced directly, we generate a large number of copies of the molecule and arbitrarily cut them into random fragments. While doing so, the order of the fragments along the DNA gets lost. Each fragment, or at least each fragment's starting piece (as long as possible),

is now sequenced directly. We thus obtain a set of overlapping fragments of known sequences. The remaining task is to reassemble the sequence of the original molecule from these fragments.

This procedure is summarized below.

Method 8.1 Shotgun Sequencing

Input: A DNA molecule $\mathcal{D}$.

1. Generate a set $C = \{\mathcal{D}_1, \ldots, \mathcal{D}_m\}$ of copies of $\mathcal{D}$.
2. Cut each copy from C randomly into smaller fragments. This results in a set of overlapping fragments $F = \{f_1, \ldots, f_n\}$.
3. Determine the sequence of all the fragments (or their starting sequences) by means of direct sequencing. This gives us a set of strings $S = \{s_1, \ldots, s_n\}$ over the alphabet $\Sigma_{\text{DNA}} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

Output: The set of strings $S = \{s_1, \ldots, s_n\}$ corresponding to the sequences of the fragments in F.

To perform these steps, one has to apply different experimental techniques. Copying DNA sequences may, for instance, be done by cloning them (see Section 2.4.2); random cuts could be obtained using vibration; and sequencing the resulting fragments is typically done using the chain termination method (see Section 2.4.3).

This now leads to the so-called fragment assembly problem.

Definition 8.1. *Let $\mathcal{D}$ be the DNA molecule we want to sequence. Let $S = \{s_1, \ldots, s_n\}$ be the set of strings obtained by shotgun sequencing $\mathcal{D}$. Then the* fragment assembly problem *is defined as follows.*

Input: A set of strings $S = \{s_1, \ldots, s_n\}$.

Output: An ordering of the strings in S corresponding to the ordering of the fragments in the original DNA molecule $\mathcal{D}$.

It is of course clear that we can derive the desired sequence directly from such an ordering.

To give an idea of the sizes that may actually occur in shotgun experiments, we refer to an example given by Myers [146].

Example 8.1. Let $\mathcal{D}$ be the DNA we want to sequence. We denote by l the length of $\mathcal{D}$ and assume an average length of $l = 100$ kbp. We further assume that we sequence $n = 1\,500$ fragments during our shotgun experiment and that each of these fragments have average length $f = 500$ bp. Therefore, we obtain a data set of size $n \cdot f = 750$ kbp that we have to analyze. Altogether, each base pair of the given DNA $\mathcal{D}$ is thus sequenced $c = \frac{f \cdot n}{l} = 7.5$ times on average. The value of c is also referred to as the *coverage*. ♢

To solve the fragment assembly for a given set of strings, in general, one follows an *overlap — layout — consensus* scheme, whose single phases are next described in more detail.

Overlap: In the first step, all possible pairwise overlaps between the strings are computed. It is important to note that overlaps do not necessarily refer to exact suffix-prefix pairs as considered in Section 4.5.3, but a certain number of errors may occur, as described in the context of alignments in Chapter 5.

Layout: Subsequently, the overlap information is used to infer an ordering on the strings that resembles their ordering in the original DNA molecule as much as possible. The structure obtained by this is called the layout.

Consensus: Finally, this layout serves to determine a string that most probably corresponds to the considered DNA molecule.

The phases are illustrated in Figure 8.1.

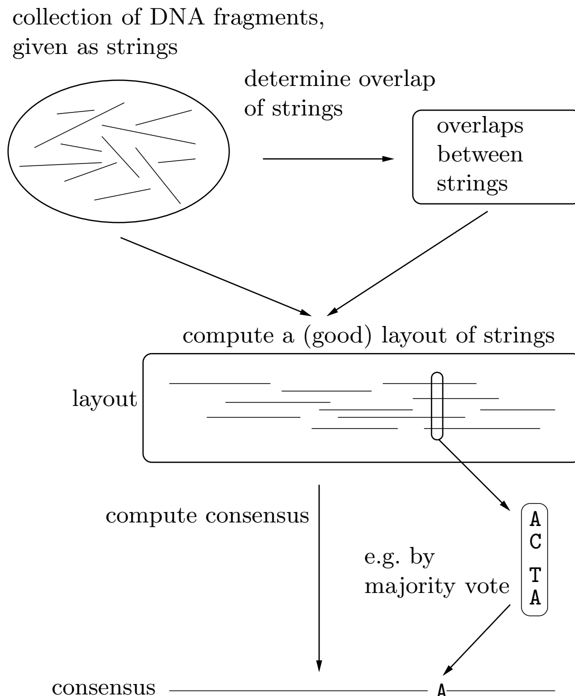


Fig. 8.1. Schematic representation of the overlap — layout — consensus phases for solving the fragment assembly problem

We have already proposed methods in Section 4.5.3 to compute the overlap between strings, and we will not further discuss the subject here. The layout resembles in its structure a semiglobal alignment of multiple strings as we studied it in Section 5.1.3. Nevertheless, due to the enormous number of strings occurring in this framework, typically no alignment methods are applied to actually compute the layout, since they would be inefficient.

The computation of the layout will play a central role in the modeling, and in the algorithmic approaches for the fragment assembly problem as discussed in the following sections.

To elucidate the computation of the consensus, we nevertheless stay with the picture of a multiple alignment to illustrate the structure of a layout. From the layout, we can derive the consensus in different ways. Often, a *majority voting* is used, where the character occurring most frequently in a column of the alignment (layout) is set to be the corresponding entry of the consensus. This also fits to the definition of consensus given in Section 5.3.1.

Before we discuss the modeling process for the fragment assembly problem in more detail, we first address some potential error sources in the underlying data and some biological facts that must be taken into account in evaluating the models with respect to their practical applicability.

8.1.1 Crucial Points to Be Considered in a Suitable Model

Due to the lab techniques required to perform shotgun sequencing, certain errors may occur. In particular, the direct sequencing of DNA (by the chain termination method) may lead to the presence of insertions, deletions, and substitutions as discussed in Section 2.4.3.

Moreover, specific problems may occur in the context of shotgun sequencing, such as *chimeric clones.* When carrying out the shotgun experiment, it may happen that fragments from different regions of the DNA join each other to form a single fragment. Hence, in the resulting data two DNA regions seem to appear consecutively although they are spatially separated in the original DNA (see Figure 8.2). The fragments resulting from the join of the two DNA regions are called *chimeric clones*.

Some addditional biological facts must be regarded when modeling the fragment assembly problem.

Incomplete coverage: In Example 8.1, we have seen how to compute the average coverage c of a shotgun experiment. But a high average coverage by no means guarantees a *complete* coverage of the considered DNA molecule by fragments. That is, there may exist regions of the molecule that are not covered by any fragment (see Figure 8.3). There are different possible reasons for this: one, for instance, may be the toxic effect of a fragment that would normally appear in the region on the host organism used for cloning it.

Unknown orientation: The orientation of fragments gets lost during the shotgun experiment. Thus, the reading direction is unknown and we are not sure whether the resulting strings occur as they are in the DNA sequence or as their reverse complement. Therefore, we have to deal with these two possibilities in fragment assembly.

Repeats: Another particularity of DNA, which we must take into account in fragment assembly, is the appearance of multiple numbers of identical (or nearly identical) substrings at different positions of DNA, called *repeats*. These

repeats may occur in various different types, with different lengths and different levels of similarity. The presence of repeats leads to the necessity in fragment assembly to distinguish whether the overlap of two fragments results from an actual overlap of these fragments in the original DNA or from the fact that these fragments occur only within the same repeats. We will consider this problem in more detail when we analyze the particular models. Furthermore, an unfavorable distribution of the repeats may lead to situations that do no longer allow the unique reconstruction of regions between them (see Figure 8.4).

The next point is neither an error nor a characteristic that may trouble our fragment assembly model. Indeed, we should also care about including all available data into our model to obtain the most realistic one with the most reliable results. One such data is the *length of the original DNA*. This length, or at least a good estimate, can be easily determined by gelelectrophoresis (as discussed in detail in Section 2.4.3). Incorporating this additional knowledge into our model may substantially increase its reliability.

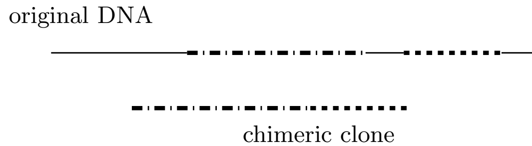


Fig. 8.2. Illustration of a chimeric clone

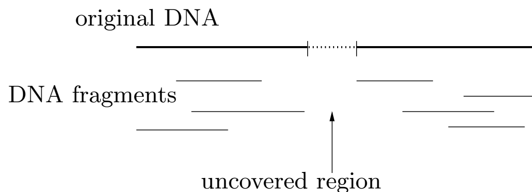


Fig. 8.3. Illustration of an incomplete coverage of DNA by fragments

While all these observations should be considered in the context of modeling the fragment assembly problem, we have to also accept restrictions of the formal model for the sake of its simplicity. Indeed, the solutions obtained based on a simple model can be good enough, such that comprehensive and complex models without any approach to achieve good solutions are no longer advantageous. It may furthermore be possible to successively extend basic models to more complex ones and to cover more and more parameters in doing so. In general, nevertheless, the above observations should be taken into account for

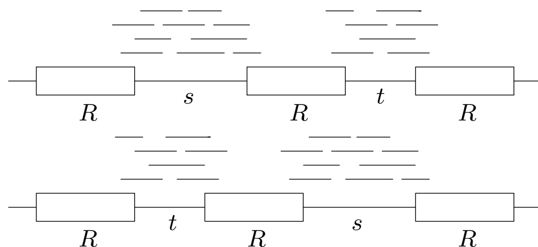


Fig. 8.4. Repeats may imply that different orderings of regions lying in between them are reasonable. Both orderings of substrings s and t are possible. R denotes the copies of the repeat

the evaluation of models for fragment assembly in order to ensure the most reliable modeling.

According to these remarks, the subsequent sections will deal with some approaches to model and solve the fragment assembly problem to a certain extent. Starting with a very abstract model, we will present some possible extensions to make it more realistic.

8.1.2 The Shortest Common Superstring Problem

The Model

A first and rather simple approach to modeling fragment assembly is to cast it as a shortest common superstring problem. According to this, we may understand the following problem setting as one possible model of the fragment assembly problem, where admittedly none of the above described problem parameters are considered yet. The idea behind using the shortest common superstring problem for modeling is based on the assumption that a string containing all strings resulting from the shotgun experiments as substrings that is as short as possible seems to be a fairly good approximation of the desired DNA sequence.

Definition 8.2. *The* shortest common superstring problem, *SCS for short, is the following optimization problem.*

Input: A set $S = \{s_1, \ldots, s_n\}$ *of strings over an alphabet* Σ.
Feasible solutions: Each superstring w *of* S, *i.e., each string* w *that contains all strings* $s_i \in S$ *as substrings.*
Costs: The length of the superstring, $cost(w) = |w|$.
Optimization goal: Minimization.

Accordingly, we call an optimal solution for the SCS a shortest common superstring *of* S.

Thus, for a given set of strings, we want to compute a shortest string that contains all given strings as substrings.

Let us assume that the set S contains two different strings s and t, where t is a proper substring of s. Then, the string t obviously plays no role in computing the superstring, since a superstring w containing s as a substring automatically also contains the substring t. Therefore, we can without loss of generality assume the input set for the SCS to be *substring free*.

Definition 8.3. *A set of strings S is said to be* substring free *if no pair of strings (s, t), $s \neq t$, exists in S such that t is a substring of s.*

In the context of the shortest common superstring problem another optimization problem is often considered, where the goal is to compute a superstring maximizing the *compression*. To explain this in more detail, we first define what we will understand by the term *trivial superstring*.

Definition 8.4. *A* trivial superstring w_T *of a set $S = \{s_1, s_2, \ldots, s_n\}$ is obtained by concatenating all strings in S in an arbitrary order,*

$$w_T = s_1 \cdot s_2 \cdot \ldots \cdot s_n.$$

The length of a trivial superstring is thus the sum of the lengths of the strings in S. We will denote this length by

$$\|S\| = |w_T| = \sum_{i=1}^{n} |s_i|.$$

It is immediately clear that a trivial superstring is a feasible solution to the SCS, but the worst one.

Now, by *compression* we mean the number of characters that are saved by a superstring with respect to the trivial one.

Definition 8.5. *Let w be a superstring of a set $S = \{s_1, s_2, \ldots, s_n\}$. The* compression *of w is defined as*

$$\text{comp}(w, S) = \|S\| - |w|.$$

The relation between the length of a trivial superstring, the length of a superstring w, and the compression of superstring w is illustrated in Figure 8.5. If the set S is clear from the context, we also write $\text{comp}(w)$ for short, instead of $\text{comp}(w, S)$.

At this point, we can define a variant of the shortest common superstring problem, where we seek to maximize the compression. This receives the same type of input and also has the same set of feasible solutions, but differs in the costs and the optimization goal.

Definition 8.6. *The* maximum compression common superstring problem*, MCCS for short, is the following optimization problem.*

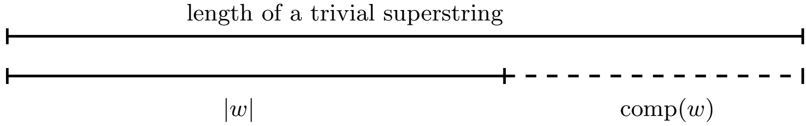


Fig. 8.5. Length of a trivial superstring, length of a superstring w, and compression of superstring w

Input: A set $S = \{s_1, \ldots, s_n\}$ of strings over an alphabet Σ.
Feasible solutions: Each superstring w of S.
Cost: The compression of superstring w, cost$(w) = \text{comp}(w, S)$.
Optimization goal: Maximization.

Often one refers to the shortest common superstring problem with *length measure* in the case of SCS and to the shortest common superstring problem with *compression measure* in case of MCCS. It should be clear from the definition that an optimal solution to the SCS is an optimal solution to the MCCS as well, and vice versa. On the other hand, as we will see in the following, performance ratios of approximation algorithms are in general not transferable between the two optimization problems.

To achieve a deeper understanding of the above definitions, let us consider the following example.

Example 8.2. Let $S = \{ababaa, bab, caba, aaddd, aabca, aacab\}$ be a set of strings over the alphabet $\{a, b, c, d\}$. We observe that the string bab is a substring of the string $ababaa$ and therefore, according to our insight above, we can ignore it. Thus, let us consider the substring-free set

$$S' = \{ababaa, caba, aaddd, aabca, aacab\}.$$

One trivial superstring for S' is, e.g., $w_T = ababaacabaaadddaabcaaacab$, with length $|w_T| = 25$. The superstring $w_1 = aadddcababaabcaacab$, on the other hand, results from the following alignment of strings from S':

```
              a a c a b
          a a b c a
      a b a b a a
     c a b a
a a d d d
---------------------------------------
a a d d d c a b a b a a b c a a c a b
```

The superstring w_1 has length 19, it hence achieves a compression of $\text{comp}(w_1, S') = 6$.

Compared with this, a shortest superstring $w_2 = aacababaabcaaddd$ can be obtained by the following alignment of strings from S':

$$
\begin{array}{l}
\qquad\qquad\quad a\,a\,d\,d\,d \\
\qquad\qquad a\,a\,b\,c\,a \\
\quad\;\; a\,b\,a\,b\,a\,a \\
\quad c\,a\,b\,a \\
\underline{a\,a\,c\,a\,b\qquad\qquad\qquad\quad} \\
a\,a\,c\,a\,b\,a\,b\,a\,a\,b\,c\,a\,a\,d\,d\,d
\end{array}
$$

This superstring has a length of 16 and hence achieves a compression of $\text{comp}(w_2, S') = 9$. ♢

To gain a better intuition on the problems, we will now introduce two graphs that can be viewed as a special representation of the inputs for SCS and MCCS, respectively. Therefore, please first recall the definition of a merge $\langle s, t\rangle$ of two strings $s = uv$ and $t = vw$ as presented in Definition 3.5, which denotes the string uvw resulting from merging s and t with a maximum length region of overlap v. Here, we denote the overlap[1] v by $Ov(s, t)$ and the prefix u of the merge $\langle s, t\rangle$ by $Pref(s, t)$. We call the length $pref(s, t) = |Pref(s, t)|$ of this prefix the *distance* between s and t. Accordingly, let $ov(s, t) = |Ov(s, t)|$. The merge $\langle s, t\rangle$ hence consists of $pref(s, t)$ many characters from s followed by t. It is therefore easy to compute the distance given the length of the overlap, and vice versa (clearly, we also need the length of s):

$$pref(s, t) = |s| - ov(s, t).$$

Definition 8.7. *Let $S = \{s_1, \ldots, s_n\}$ be a set of strings over an alphabet Σ. The* overlap graph *$G_{ov}(S)$ of S is the complete edge-weighted directed graph $G_{ov}(S) = (V, E, c)$, where*

- *$V = S$ and $E = V^2$, and*
- *$c : E \to \mathbb{N}$, with $c(s_i, s_j) = ov(s_i, s_j)$ for all $s_i, s_j \in V$.*
  That is, the edges are labeled with the lengths of the overlaps between the strings connected by them.

Accordingly, the distance graph *$G_{pref}(S)$ of S is the complete edge-weighted directed graph $G_{pref}(S) = (V, E, c)$, where*

- *$V = S$ and $E = V^2$, and*
- *$c : E \to \mathbb{N}$, with $c(s_i, s_j) = pref(s_i, s_j)$ for all $s_i, s_j \in V$.*
  That is, instead of labeling the edges with the lengths of the overlaps, we label them by the corresponding distances.

Note that when we first defined directed graphs in Definition 3.12, we considered them to contain no self-loops.[2] However, in this particular context, self-loops are obviously absolutely meaningful, describing the overlap of a string with itself.

The overlap graph for the input instance from Example 8.2 is shown in Figure 8.6; the distance graph for the same input is shown in Figure 8.7.

[1] As we are dealing with superstrings here, we are looking for exact, and not approximate, overlaps.

[2] A self-loop in a directed graph is an edge (v, v) pointing from vertex v to itself.

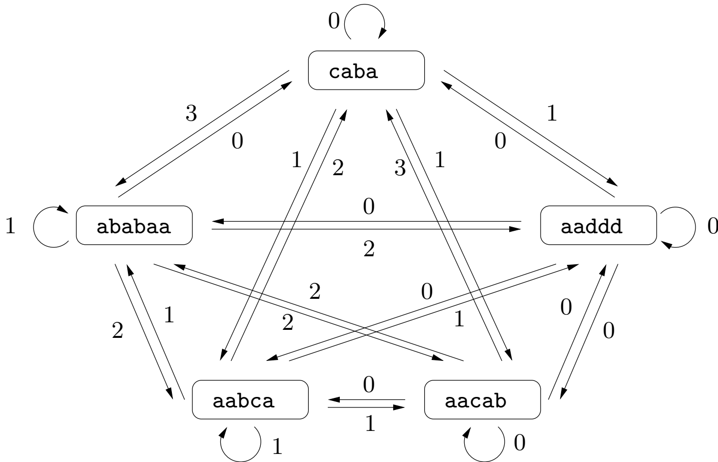


Fig. 8.6. Overlap graph for the strings given in Example 8.2

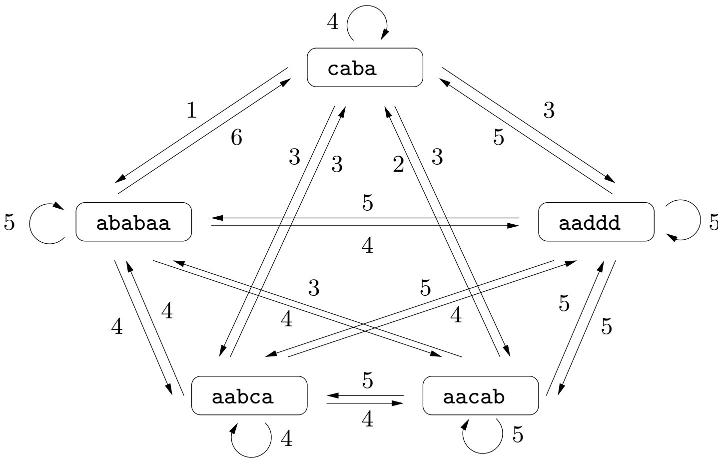


Fig. 8.7. Distance graph for the strings given in Example 8.2

In Section 4.5.3, we proposed a method enabling us to compute all pairwise overlaps of a set S of n strings in time $O(\|S\| \cdot (n + \log \|S\|))$. So, we can directly infer the following theorem.

Theorem 8.1. *Let $S = \{s_1, \ldots, s_n\}$ be a set of strings over an alphabet Σ. The computation of the overlap graph as well as of the distance graph of S can be done in time $O(\|S\| \cdot (n + \log \|S\|))$.* □

Note that an edge (s, t) in the distance graph (overlap graph) corresponds in a natural way to a merge of the strings s and t.

As an edge can be identified with a merge, we can similarly consider a path in the distance graph (overlap graph) as a sequence of several merges. As an example, let $(s_1, s_2, \ldots, s_k)$ be a directed path in a distance graph (overlap graph]); then, we denote the corresponding string by

$$\langle s_1, s_2, \ldots, s_k \rangle = \mathit{Pref}(s_1, s_2)\mathit{Pref}(s_2, s_3) \ldots \mathit{Pref}(s_{k-1}, s_k)s_k.$$

This string thus has length

$$|\langle s_1, s_2, \ldots, s_k \rangle| = \mathit{pref}(s_1, s_2) + \mathit{pref}(s_2, s_3) + \cdots + \mathit{pref}(s_{k-1}, s_k) + |s_k|.$$

Therefore, we may generally define a superstring by an ordering of the strings in the input set.

Definition 8.8. *Let $S = \{s_1, \ldots, s_n\}$ be a set of strings, let $\pi = (s_{i_1}, \ldots, s_{i_n})$ be an ordering of the strings. Then the* superstring w_π induced by π *is*

$$w_\pi = \langle s_{i_1}, \ldots, s_{i_n} \rangle = \mathit{Pref}(s_{i_1}, s_{i_2}) \cdot \mathit{Pref}(s_{i_2}, s_{i_3}) \cdot \ldots \cdot \mathit{Pref}(s_{i_{n-1}}, s_{i_n}) \cdot s_{i_n}.$$

Based on these considerations, we can recognize a relation between the SCS and the traveling salesman problem (TSP).[3] Namely, to solve the SCS, we are looking for an ordering of strings, i.e., a directed tour in the distance graph visiting each vertex exactly once, which has the minimal cost with respect to its induced superstring. Let us assume that we have a solution for the TSP for a given distance graph, i.e., a tour visiting all vertices and of minimal cost. Let this solution be given in terms of a permutation $\pi = (s_{i_1}, \ldots, s_{i_n})$. Its costs with respect to the TSP is thus

$$\mathit{cost}_{\mathrm{TSP}}(\pi) = \mathit{pref}(s_{i_1}, s_{i_2}) + \mathit{pref}(s_{i_2}, s_{i_3}) + \ldots + \mathit{pref}(s_{i_{n-1}}, s_{i_n}) + \mathit{pref}(s_{i_n}, s_{i_1}).$$

From this we can directly obtain a feasible solution for the corresponding SCS by computing the induced superstring w_π of length

$$\mathit{cost}_{\mathrm{SCS}}(w_\pi) = \mathit{pref}(s_{i_1}, s_{i_2}) + \mathit{pref}(s_{i_2}, s_{i_3}) + \ldots + \mathit{pref}(s_{i_{n-1}}, s_{i_n}) + |s_{i_n}|.$$

Knowing that the distance of two strings $\mathit{pref}(s, t)$ is always smaller than the length of the string s itself, we can directly infer that the cost of the solution for the TSP is at least as high as the cost of the corresponding SCS solution. In this way, we obtain a directed path from a solution of the TSP whose induced superstring might be a good candidate for a short superstring. Let $\mathrm{Opt}_{\mathrm{TSP}}$ be the cost of an optimal solution for the TSP on the distance graph for a set of strings S, and let $\mathrm{Opt}_{\mathrm{SCS}}$ be the length of an optimal solution for the SCS on the same set of strings S; then, we have

$$\mathrm{Opt}_{\mathrm{TSP}} \leqslant \mathrm{Opt}_{\mathrm{SCS}} - \mathit{ov}_{\min} \leqslant \mathrm{Opt}_{\mathrm{SCS}},$$

[3] Here we have to deal with the variant of the TSP where the input graph is directed.

where $ov_{\min}$ denotes the minimum length of an overlap between two strings in S.

However, this procedure still does not lead us to an efficient method for solving the SCS, since the TSP is, as we have already pointed out in Section 3.3, an NP-hard optimization problem.

Therefore, let us at this point consider the complexity of the SCS itself and subsequently propose some approaches to solve it. Assume that a set $S = \{s_1, \ldots, s_n\}$ is given. A simple way to determine a shortest superstring for S would consist of examining all different orderings of strings from S and choosing the shortest of the induced superstrings. But as the number of different orderings of n strings is $n!$, this approach is certainly not applicable in practice, where values of n may be quite large.

Indeed, the SCS also turns out to be computationally difficult and therefore the existence of a polynomial-time algorithm solving it is highly unlikely. To make this more precise, we consider the decision version Dec-SCS of the SCS.

Definition 8.9. *The* decision version of the shortest common superstring problems, DecSCS *for short, is defined as follows.*

Input: A set $S = \{s_1, \ldots, s_n\}$ of strings over an alphabet Σ and an integer k.
Output: Yes *if there exists a superstring w of S, where $|w| \leqslant k$, and* No *otherwise.*

This problem was shown to be NP-complete by Gallant et al. in [78].

Theorem 8.2. DecSCS *is NP-complete.* □

We omit the proof of this theorem here and instead refer the reader to the literature referenced in Section 8.4.

As we have seen in Theorem 8.2, DecSCS is NP-complete, and therefore it is highly unlikely that we can design an algorithm for SCS or MCCS which computes an optimal solution in polynomial time. Therefore, we will, as in Section 7.2 (where we examined the mapping with unique probes and errors) reduce our requirements and search for rather good solutions instead of necessarily optimal ones. We hence consider the concept of approximation algorithms and try to find ones that compute solutions deviating from an optimal one as little as possible.

The Greedy Algorithm

An especially simple approach to design an approximation algorithm for the SCS is based on the *greedy method* (see Algorithm 8.1). Here, we successively merge those strings that have the largest overlap and thus determine a superstring step by step. In general, *greedy algorithms* proceed by developing a solution stepwise, using a locally optimal subsolution in each step. They have their name due to their property of acting greedily — "Always take the largest (best) piece of the cake."

Algorithm 8.1 Greedy Superstring

Input: A set of strings $S = \{s_1, \ldots, s_n\}$.

while $|S| > 1$ **do**

1. Find $s_i, s_j \in S$, $s_i \neq s_j$, that have a maximum overlap among all strings in S.
2. Let $s' = \langle s_i, s_j \rangle$ be the merge of strings s_i and s_j.
3. Delete s_i, s_j from S and insert s' into S.

Output: The only remaining string $w_{\text{greedy}} \in S$.

Let us now consider the operation of Algorithm 8.1 from a different point of view.

Let the input S be given in terms of an overlap graph as described in Definition 8.7; then, Algorithm 8.1 always searches for an edge (s_i, s_j) with the highest weight in the graph that is not a self-loop[4], and merges the strings corresponding to the incident vertices with each other. After this, the vertices s_i and s_j are united into one single vertex that now represents the merge $s' = \langle s_i, s_j \rangle$ of the two strings. Furthermore, all outgoing edges of vertex s_i and all incoming edges of vertex s_j are deleted from the graph. This contraction is repeated until only one vertex remains in the graph, representing the final superstring.

On a slightly more abstract level, we can also view Algorithm 8.1 as computing a Hamiltonian path in the overlap graph, i.e., as a directed path visiting each vertex exactly once. For that, let S again be given in terms of an overlap graph. The algorithm now chooses each time an edge with maximal weight that is not a self-loop, such that the set of chosen edges still can be extended to a Hamiltonian path. That is, the new edge must lead neither to the situation where there exists a vertex whose indegree or outdegree with respect to the chosen edges is greater than one, nor to the situation where the newly chosen edge generates a directed cycle together with some of the edges chosen so far. Algorithm 8.1 in this way computes a Hamiltonian path of the overlap graph, and the final superstring corresponds to the one induced by the ordering of strings (vertices) along this path.

We illustrate the work of Algorithm 8.1 with the following example.

Example 8.3. We consider the input instance $S' = \{ababaa, caba, aaddd, aabca, aacab\}$ from Example 8.2. All pairwise overlaps between strings in S' are shown in Table 8.1 (see also the overlap graph in Figure 8.6).

On this input, Algorithm 8.1 now operates as follows: It chooses a pair of strings with maximal overlap. In this case the pair $(caba, ababaa)$ as well as the pair $(aacab, caba)$ are suitable, since both achieve an overlap of 3. Let us

[4] The usage of self-loops is explicitly prohibited, as the algorithm demands $s_i \neq s_j$ in step 1.

	ababaa	*caba*	*aaddd*	*aabca*	*aacab*
ababaa	1	0	**2**	2	2
caba	**3**	0	1	1	1
aaddd	0	0	0	0	0
aabca	1	2	1	1	**1**
aacab	2	**3**	0	0	0

Table 8.1. Pairwise overlaps between strings from Example 8.3

assume that the algorithm chooses the pair $(caba, ababaa)$ and thus computes the merge $\langle caba, ababaa\rangle = cababaa$. After executing this step, the set of remaining strings is

$$\{cababaa, aaddd, aabca, aacab\}.$$

The procedure is now iterated. Accordingly, in the next step the algorithm creates the merge $\langle aacab, cababaa\rangle = aacababaa$ leading to the set

$$\{aacababaa, aaddd, aabca\}.$$

In the following step, the algorithm chooses a pair with overlap 2, for instance, $(aacababaa, aaddd)$, and creates the corresponding merge $\langle aacababaa, aaddd\rangle = aacababaaddd$. Then, the remaining set is

$$\{aacababaaddd, aabca\}.$$

In the last step, the algorithm performs the merge

$$\langle aabca, aacababaaddd\rangle = aabcaacababaaddd,$$

and hence we obtain the superstring

$$w_{\text{greedy}} = aabcaacababaaddd$$

of length 16 and compression $\text{comp}(w_{\text{greedy}}) = 9$. In this particular example, w_{greedy} is in fact a shortest superstring for the input S'. But this need not be the case in general, as we will see in a later example. ◊

Before we examine the running time of Algorithm 8.1, we present the following observation on its work on substring free instances.

Lemma 8.1. *If a substring free set $S = \{s_1, \ldots, s_n\}$ is input to Algorithm 8.1, the set S remains substring free during the computation.*

Proof. Let $S = \{s_1, \ldots, s_n\}$ be a substring free set of strings. A merge $s' = \langle s_i, s_j\rangle$ performed by Algorithm 8.1 cannot become a superstring of strings in S except for s_i and s_j. Assume the contrary, i.e., that there exists a string $t \in S$ such that t is a substring of s'. As S was assumed to be substring free, t is neither a substring of s_i nor of s_j. Hence, the overlap $Ov(s_i, s_j)$ is a proper substring of t. But then, t has a larger overlap with either s_i or s_j than s_i has with s_j, which contradicts the greedy property of Algorithm 8.1. □

We are now ready to consider the running time of Algorithm 8.1.

Theorem 8.3. *For an input $S = \{s_1, \ldots, s_n\}$, Algorithm 8.1 computes a superstring of S in time $O(\|S\| \cdot (n + \log \|S\|))$.*

Proof. By the description of the algorithm and the discussion above, it is clear that the output is actually a superstring. We now analyze its running time.

Before we actually start to consider steps 1 to 3 in the while loop, we first compute all pairwise overlaps between the strings in S and subsequently construct a sorted list of them.

According to Theorem 8.1, we can compute all overlaps in time $O(\|S\| \cdot (n + \log \|S\|))$. As we will ignore self-loops, i.e., merges of type $\langle s_i, s_i \rangle$ here, we may ignore the overlaps and set the corresponding values to -1.

We are now provided with a matrix A containing n^2 entries that we want to sort. As each entry corresponds to the length of an overlap, it is a natural number, and it is definitively smaller than $\|S\|$. Thus, we can sort these elements in time $O(n^2 + \|S\|) \subseteq O(n \cdot \|S\|)$ using counting sort (see Algorithm 4.13).

We may additionally assume that there is a pointer assigned to each element of the sorted array, denoted by L, referring to the corresponding entry in A, i.e., providing us with the pair of strings which corresponds to this overlap.

For these preprocessing steps, we thus require time in $O(\|S\| \cdot (n + \log \|S\|))$. Now, let us consider steps 1 to 3 of the algorithm.

In step 1, we have to determine the pair (s_i, s_j) with the largest overlap. Thanks to our sorted array L and its assigned pointers, we can do so in constant time.

In steps 2 and 3 we have to perform the merge $s' = \langle s_i, s_j \rangle$, remove s_i and s_j from the current set of strings, and insert s' there. With respect to our implementation, we thus have to perform an update on the overlap matrix A and on the sorted array L.

To update A, we have to insert s' into the matrix. The corresponding entries can be easily computed by $A[s', t] = A[s_j, t]$, $A[t, s'] = A[t, s_i]$, and $A[s', s'] = -1$, for all t in the current set S without s_i, s_j, and s'. Here, we set $A[s', s'] = -1$ to guarantee the condition $s_i \neq s_j$ in step 1. The rows and columns corresponding to s_i and s_j are removed from A. This update process can be done in time $O(n)$.

To update L, we simply remove its first entry and rearrange the pointers to A. If there was a pointer to $A[s_j, t]$, set it to $A[s', t]$, and if there was a pointer to $A[t, s_i]$, set it to $A[t, s']$. Observe that by introducing s' we cannot obtain "new" values of overlaps, since this would imply that the current S is not substring free which contradicts our assumption in Lemma 8.1. Moreover, we remove all elements from L that point to entries $A[s_i, t]$ and $A[t, s_j]$. By following the pointers in the opposite direction, this step can be implemented in time $O(n)$. Finally, the first element of L might now be deleted, since it is of type $A[s_i, t]$ or $A[t, s_j]$. To find the new first element of L requires $O(n)$ steps, since at most this number of elements is removed.

Since steps 1 to 3 of the algorithm will be executed $O(n)$ times altogether, we can, with the above considerations, derive a running time of $O(n^2)$. Together with the effort needed for preprocessing the data, we obtain the claimed result, $O(n^2 + \|S\| \cdot (n + \log \|S\|)) = O(\|S\| \cdot (n + \log \|S\|))$. □

Let us now illustrate the work of the algorithm on the level of strings.

Example 8.4. Let $S = \{c(ab)^m, (ba)^m, (ab)^m c\}$ be an input for the SCS over the alphabet $\Sigma = \{a, b, c\}$ for an arbitrary natural number m. Algorithm 8.1 runs on this input as follows. The algorithm starts with computing the pair with a maximal overlap in S. This is the pair $(c(ab)^m, (ab)^m c)$ with an overlap of length $2m$. Then, the two strings are merged together and thus, after the first pass through steps 1 to 3 of the algorithm, we obtain the set $\{(ba)^m, c(ab)^m c\}$. Since the string $c(ab)^m c$ starts and ends with character c and the string $(ba)^m$ consists solely of characters a and b, the only remaining way to generate a superstring is to concatenate both strings. The overlap is 0 in this case. Algorithm 8.1 hence computes either the superstring $(ba)^m c(ab)^m c$ or the superstring $c(ab)^m c(ba)^m$, each having length $4m + 2$.

But, if we take a closer look at our example, we recognize that it is favorable to move the string $(ba)^m$ between the two strings $c(ab)^m$ and $(ab)^m c$. This procedure results in the optimal (shortest) superstring $ca(ba)^m bc$ of length $2m + 4$ for our input S.

The approximation ratio achieved by Algorithm 8.1 in this example is thus about 2, since the fraction $\frac{4m+2}{2m+4}$ converges to 2 for m tending to infinity.

The above example thus shows that Algorithm 8.1 cannot achieve an approximation ratio better than 2 in general. On the positive side, it was shown that Algorithm 8.1 always computes a superstring that is at most 4 times as long as the optimal one [33].

Theorem 8.4. *Algorithm 8.1 (Greedy Superstring) is a 4-approximation algorithm for the SCS.* □

We will not prove this result here, because it is beyond the scope of this chapter. Note that, although a 4-approximation has been shown for Algorithm 8.1, no one has been able so far to actually find an example on which the algorithm performs this badly. Indeed, the example above, showing an approximation ratio of 2, is the worst example known. Thus, there is still a gap according to the exact approximation ratio achieved by the algorithm.[5] In general, it is assumed that Algorithm 8.1 is indeed a 2-approximation algorithm for the SCS; only the corresponding proof is still missing.

In the framework of superstrings, we have defined, besides the SCS, also another optimization problem, the MCCS. Now, let us analyze how Algorithm

[5] Note that for *problems* this is by no means unusual; but for a *certain algorithm* — actually a very simple one — this circumstance is quite unsatisfactory.

8.1 performs with respect to this optimization goal, i.e., to maximizing the compression.

We note here that, although the optimization problems SCS and MCCS have the same optimal solutions, the approximate solutions can vary arbitrarily. That is, an ε-approximation for one problem does not necessarily imply an ε-approximation for the other one. Figure 8.8 illustrates this fact and shows why the approximation ratios cannot be transfered directly.

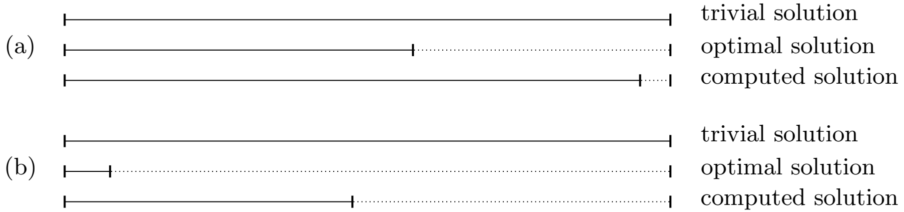


Fig. 8.8. (a) An input instance for which the computed solution is a good approximation with respect to the length, but does not imply a good approximation with respect to the compression. (b) The opposite case. The length of the superstring is represented by a solid line and the size of the compression by a dotted line

We will now examine the approximation performance of Algorithm 8.1 with respect to MCCS.

Theorem 8.5. *Algorithm 8.1 (Greedy Superstring) is a* 2*-approximation algorithm for the MCCS.* □

We will omit the proof of this theorem here. Instead, we will, using a simple argument, present a weaker version of the above statement.

Theorem 8.6. *Algorithm 8.1 (Greedy Superstring) is a* 3*-approximation algorithm for the MCCS.*

Proof. Let w_{opt} be a shortest superstring for an input $S = \{s_1, \ldots, s_n\}$. Thus, each string $s \in S$ occurs on at least one position in w_{opt} as a substring. Let $\pi_{\text{opt}} = (s_{i_1}, \ldots, s_{i_n})$ be the order of strings in S induced by their first occurrence in w_{opt}. As w_{opt} is a *shortest* superstring, we may represent w_{opt} by a sequence of merges $\langle \ldots \langle\langle s_{i_1}, s_{i_2}\rangle, s_{i_3}\rangle \ldots\rangle, s_{i_n}\rangle$ according to π_{opt}.[6] Let $M_{\text{opt}} = \{\langle s_{i_1}, s_{i_2}\rangle, \langle s_{i_2}, s_{i_3}\rangle, \ldots, \langle s_{i_{n-1}}, s_{i_n}\rangle\}$ denote the set of merges performed in the optimal solution on the level of the single strings in S. In our visualization of superstrings in terms of Hamiltonian paths in the overlap graph, M_{opt} corresponds to the set of edges on this path. Now, the compression

[6] It is important that w_{opt} is a shortest superstring, since otherwise a merge would not necessarily represent it. For example, for the strings ab and ba, the string $abba$ is in fact a superstring, but it is different from the merge $\langle ab, ba\rangle = aba$.

provided by the optimal solution is given by $\mathrm{comp}(w_{\mathrm{opt}}) = \sum_{m \in M_{\mathrm{opt}}} ov(m)$, the sum of the lengths of the overlaps of merges in M_{opt}.

In the same way, the superstring w_{greedy} computed by Algorithm 8.1 determines an ordering of the strings in S. Let this one be given by $\pi_{\mathrm{greedy}} = (s_{j_1}, \ldots, s_{j_n})$. Accordingly, denote the set of merges in the greedy solution respecting π_{greedy} by $M_{\mathrm{greedy}} = \{\langle s_{j_1}, s_{j_2}\rangle, \langle s_{j_2}, s_{j_3}\rangle, \ldots, \langle s_{j_{n-1}}, s_{j_n}\rangle\}$.

We now examine the amount of compression of the optimal solution w_{opt} that we can lose when performing a merge of the greedy solution. In particular, we show that each of these merges can rule out at most three merges in the optimal solution, such that they can no longer be performed.

Let $m = \langle s_{j_k}, s_{j_{k+1}}\rangle$ be a merge of the greedy solution.[7] There are three types of merges with respect to the ordering $\pi_{\mathrm{opt}} = (s_{i_1}, \ldots, s_{i_n})$ of strings in the optimal solution w_{opt}.

1. If $m = \langle s_{i_l}, s_{i_{l+1}}\rangle$ for some l ($1 \leqslant l \leqslant n-1$), then the merge performed by Algorithm 8.1 corresponds to a merge of the optimal solution w_{opt}, and thus m does not rule out any other merge of the optimal solution (see Figure 8.9 (a)).
2. If $m = \langle s_{i_l}, s_{i_{l+h}}\rangle$ for appropriate l and h with $1 < h \leqslant n-l$, then m corresponds to a kind of forward merge in w_{opt}, and at most two merges of the optimal solution become forbidden, namely, $\langle s_{i_l}, s_{i_{l+1}}\rangle$ and $\langle s_{i_{l+h-1}}, s_{i_{l+h}}\rangle$ (see Figure 8.9 (b)).
3. If $m = \langle s_{i_{l+h}}, s_{i_l}\rangle$ for appropriate l and h with $1 < h \leqslant n-l$, then m corresponds to a kind of backward merge in w_{opt} and at most three merges of the optimal solution become forbidden, namely, $\langle s_{i_{l+h}}, s_{i_{l+h+1}}\rangle$, $\langle s_{i_{l-1}}, s_{i_l}\rangle$, and one of the merges in $M = \{\langle s_{i_l}, s_{i_{l+1}}\rangle, \ldots, \langle s_{i_{l+h-1}}, s_{i_{l+h}}\rangle\}$, since the use of *all* merges in M would imply a cycle, which is not allowed (see Figure 8.9 (c)).

Hence, any merge in M_{greedy} may rule out at most three merges in M_{opt}. We denote the set of merges in M_{opt} that are prevented by a merge $m \in M_{\mathrm{greedy}}$ by $\mathit{prevent}(m)$.

Now, we are ready to estimate the ratio between the compression provided by the optimal and the greedy solution. Since each merge $m \in M_{\mathrm{greedy}}$ can rule out at most three merges in M_{opt} as shown above, it holds that, for all $m \in M_{\mathrm{greedy}}$,

$$|\mathit{prevent}(m)| \leqslant 3.$$

Since Algorithm 8.1 always chooses a remaining merge with maximal overlap, we obtain, for all $m \in M_{\mathrm{greedy}}$,

$$ov(m) \geqslant \frac{1}{3}\left(\sum_{m' \in \mathit{prevent}(m)} ov(m')\right). \tag{8.1}$$

[7] Note that we may consider any merge from the greedy solution here, since the order how the merges were performed according to π_{greedy} is irrelevant.

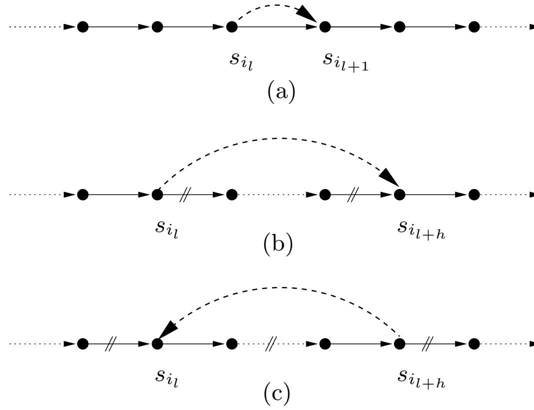


Fig. 8.9. (a) The merge $m = \langle s_{i_l}, s_{i_{l+1}} \rangle$ is part of the optimal solution w_{opt} given as a Hamiltonian path within the overlap graph. (b) How two merges of w_{opt} can be ruled out by one merge of Algorithm 8.1. (c) The case in which one merge of the algorithm rules out three merges of the optimal solution. We depict the merge computed by the algorithm as a dashed arrow (note the different direction in (b) and (c)) and mark the forbidden merges of the optimal solution with a double slash on the edges

Note that Inequality (8.1) holds in particular if $m \in M_{\text{greedy}} \cap M_{\text{opt}}$, i.e., if $\mathit{prevent}(m) = \emptyset$. Additionally, it is easy to recognize that for each merge $m' \in M_{\text{opt}}$ we have

$$m' \in M_{\text{greedy}} \text{ or there exists an } m \in M_{\text{greedy}} \text{ with } m' \in \mathit{prevent}(m). \quad (8.2)$$

The statement (8.2) is immediate, since the existence of a merge m' that does not satisfy this requirement implies that additional merges have to be performed to obtain a connected superstring.

Since the compression is the sum of the achieved overlaps, we can estimate the approximation ratio of Algorithm 8.1 as follows:

$$\begin{aligned}
\frac{\text{comp}(w_{\text{opt}})}{\text{comp}(w_{\text{greedy}})} &= \frac{\sum_{m' \in M_{\text{opt}}} ov(m')}{\sum_{m \in M_{\text{greedy}}} ov(m)} \\
&\overset{(8.1)}{\leqslant} \frac{\sum_{m' \in M_{\text{opt}}} ov(m')}{\sum_{m \in M_{\text{greedy}}} \frac{1}{3}(\sum_{m' \in \mathit{prevent}(m)} ov(m'))} \\
&= \frac{\sum_{m' \in M_{\text{opt}}} ov(m')}{\frac{1}{3} \sum_{m \in M_{\text{greedy}}} \sum_{m' \in \mathit{prevent}(m)} ov(m')} \\
&\overset{(8.2)}{\leqslant} \frac{\sum_{m' \in M_{\text{opt}}} ov(m')}{\frac{1}{3} \sum_{m' \in M_{\text{opt}}} ov(m')} \quad = \quad 3
\end{aligned}$$

This completes our proof. □

The Cycle Cover Algorithm

Besides the above presented approach for approximating the SCS using a greedy method, there exists another common approach that is based on the computation of a minimum cycle cover in the distance graph. Before we consider this approach in detail, we first formally define the term *minimum cycle cover.*

Definition 8.10. *Let* $G = (V, E, w)$ *be a complete directed graph with a weight function* $w : E \to \mathbb{N}$*. A* cycle cover C of G *is a set of directed cycles in* G*,* $C = \{c_1, \dots, c_k\}$*, such that each vertex in* V *is contained in exactly one cycle* $c_i \in C$*,* $1 \leqslant i \leqslant k$*.*

The cost $cost(C)$ of a cycle cover $C = \{c_1, \dots, c_k\}$ *is the sum of all edge-weights of all edges contained in the cycles:*

$$cost(C) = \sum_{i=1}^{k} \sum_{e \in c_i} w(e)$$

A minimum cycle cover $C_{\min}(G)$ of a graph G *is a cycle cover of* G *with minimum cost.*

It is a known fact that one can efficiently compute such a minimum cycle cover of a graph. We will not consider the algorithms here, but refer the reader to the bibliographic notes at the end of this chapter instead.

We have already discussed at the beginning of this section that, if we are able to determine *one* cycle in the distance graph with cost as small as possible, i.e., a solution of the directed TSP, then this would also imply a good solution for the SCS. The idea of Algorithm 8.2 is based on getting close to such a cycle with minimal cost by using an algorithm for computing a minimum cycle cover.[8] In fact, the computation of a minimum cycle cover is done twice. A minimum cycle cover of the distance graph is first computed, followed by the computation of a second minimum cycle cover on representatives of each of the computed cycles. Now, the obtained cycles are concatenated and expanded to finally achieve a superstring.

Note that the expansion of r_c in Step 7 to a superstring on the vertices in cycle c is actually longer than necessary, but has the property of starting and ending with r_c, which is of particular importance, since this expansion has to replace r_c in w'.

Additionally, it is important to note that although we compute cycle covers with respect to the distance between strings, i.e., on the distance graph, we also refer to the overlap between strings, as, for instance, in Step 5; i.e., we are also concerned with the overlap graph.

The work of Algorithm 8.2 is illustrated by the following example.

[8] This algorithm is also called *generic superstring algorithm* in the literature, because it provides the algorithmic idea on which a number of good approximation algorithms for the SCS are based.

Algorithm 8.2 Cycle Cover Superstring Algorithm (CCSA)

Input: A set of strings $S = \{s_1, \ldots, s_n\}$ and the corresponding distance graph $G_{pref}(S)$.

1. Compute a minimum cycle cover C of $G_{pref}(S)$.
2. For each cycle $c \in C$, choose an arbitrary representative r_c. Denote the set of representatives by $R = \{r_c \mid c \in C\}$.
3. Compute the subgraph G' of $G_{pref}(S)$ induced by R.
4. Compute a minimum cycle cover C' of G'.
5. In each cycle $c' = (c'_1, \ldots, c'_{k_{c'}}) \in C'$ delete the edge corresponding to a minimal *overlap*. Let, without loss of generality, $(c'_{k_{c'}}, c'_1)$ be this edge. The strings in the remaining path are now merged in order. Thus, for all $c' \in C'$, a string $u_{c'} = \langle c'_1, \ldots, c'_{k_{c'}} \rangle$ is obtained.
6. Concatenate all the strings $u_{c'}$ for all $c' \in C'$ and denote the resulting string by w'.
7. Let $c = (c_1, \ldots, c_{k_c})$ be a cycle in the cycle cover C. Since c is a cycle, we can, without loss of generality, assume that the chosen representative r_c corresponds to vertex c_1.
   Replace each representative r_c in w' by the concatenation of all prefixes in cycle c followed by r_c itself, i.e., by
   $$Pref(r_c, c_2)Pref(c_2, c_3)Pref(c_3, c_4)\ldots Pref(c_{k_c-1}, c_{k_c})Pref(c_{k_c}, r_c)r_c.$$
   We denote the resulting string by w.

Output: The string $w_{\text{cc}} = w$.

Example 8.5. Let S be an input for Algorithm 8.2. Let us consider the distance graph $G = G_{pref}(S)$. We determine the minimum cycle cover C as shown in Figure 8.10 (i). According to the labels used there, we can write the set S as

$$S = \{a_1, \ldots, a_4, b_1, \ldots, b_6, c_1, \ldots, c_4, d_1, \ldots, d_5, e_1, \ldots, e_5, f_1, f_2, g_1, g_2, g_3\}.$$

From each cycle in C, we choose a representative r_i with $i \in \{a, b, c, d, e, f, g\}$ and consider the subgraph G' of G induced by these representatives. For this subgraph, we again determine a minimum cycle cover C' and obtain in this way the cycles (r_a, r_b, r_c), (r_d, r_e), and (r_f, r_g), as shown in Figure 8.10 (ii).

From each of these cycles we now remove the edge that corresponds to the merge with the smallest overlap. This is illustrated in Figure 8.10 (ii) by a double crossing on the edge. We obtain the strings $u_1 = \langle r_b, r_c, r_a \rangle$, $u_2 = \langle r_e, r_d \rangle$, and $u_3 = \langle r_f, r_g \rangle$, which are now concatenated to obtain the string $w' = u_1 u_2 u_3$. Finally, we replace the representatives in w' by their corresponding cycles; that is, we obtain the following superstring:

$$\begin{aligned} w = {} & \langle \langle b_1, \ldots, b_6 \rangle, \langle c_1, c_2, c_3, c_4 \rangle, \langle a_1, a_2, a_3, a_4 \rangle \rangle \\ & \cdot \langle \langle e_1, \ldots, e_5 \rangle, \langle d_1, \ldots, d_5 \rangle \rangle \\ & \cdot \langle \langle f_1, f_2 \rangle, \langle g_1, g_2, g_3 \rangle \rangle \end{aligned}$$

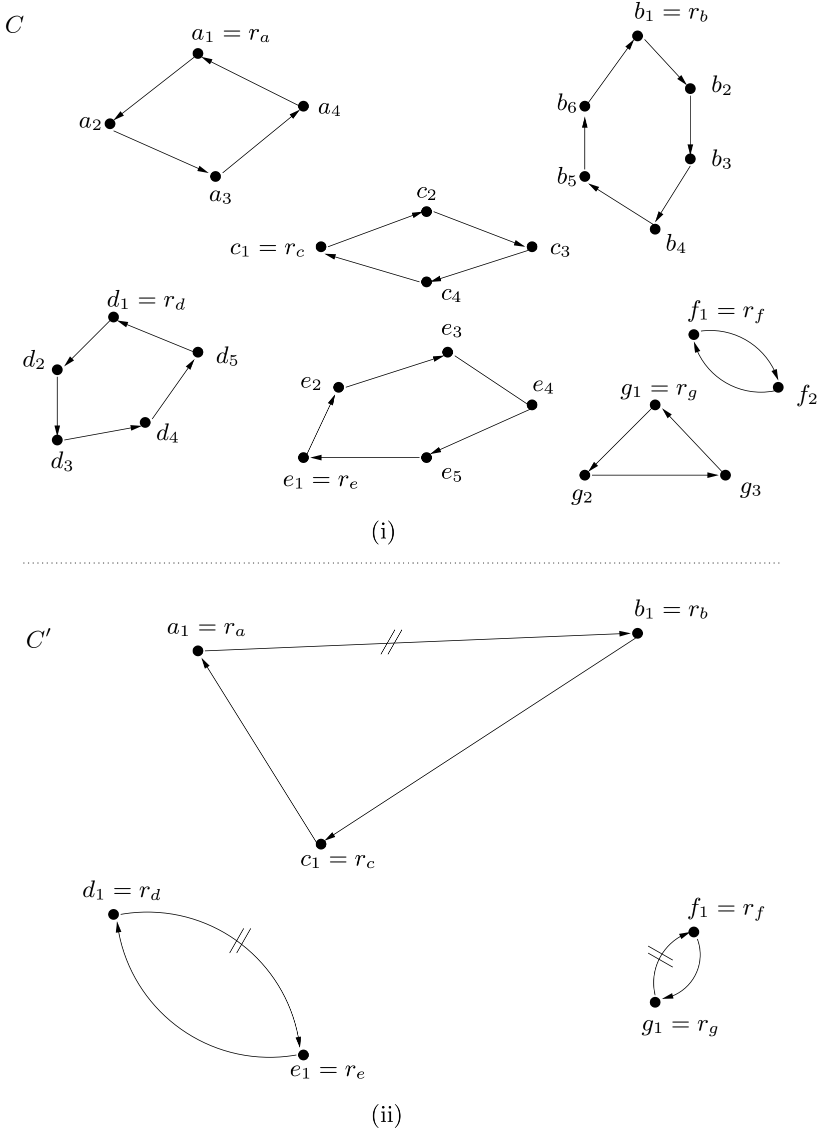


Fig. 8.10. Illustration of the work of Algorithm 8.2 on the instance from Example 8.5

From the procedure of the algorithm, it should be clear that the computed string w_{cc} is a superstring for the given input. Furthermore, we show that Algorithm 8.2 is an approximation algorithm for the SCS.

Theorem 8.7. *Algorithm 8.2 (CCSA) is a* 3*-approximation algorithm for the SCS.*

Proof. We split the proof of this claim into parts. Before discussing the parts in detail, we first sketch the overall structure of the proof.

(i) We start showing that the costs of a minimum cycle cover in a distance graph establish a lower bound on the length of the shortest superstring.
(ii) Then we estimate the length of strings $u_{c'}$ computed in step 5 of Algorithm 8.2.
(iii) This yields an estimate for the length of string w' computed in step 6 based on the cost of the minimum cycle cover C' and the overlap lost during the constructing of strings $u_{c'}$.
(iv) Based on this, we can estimate the length of the computed superstring w with respect to the length of a shortest superstring for S and the lost overlap.
(v) Finally, we estimate the lost overlap in terms of the cost of cycle cover C and obtain the desired result.

Before we actually start considering these claims, we first introduce some notation.

- Let $\mathrm{Opt}_{SCS}(S)$ denote the length of a shortest superstring of a set S, and let $\mathrm{Opt}_{CC}(G)$ denote the cost of a minimum cycle cover for a graph G.
- By $cost(c)$ we mean the overall length of a cycle c.
- As in Algorithm 8.2, we use c to refer to cycles in the cycle cover C of G and c' to refer to those in C'.
- The length of a minimal overlap in a cycle $c' \in C'$ is denoted by $\mathit{min\text{-}ov}(c')$. Accordingly, let $\mathit{min\text{-}ov}(C')$ denote the sum of the lengths of the minimal overlaps over all cycles $c' \in C'$, i.e., $\mathit{min\text{-}ov}(C') = \sum_{c' \in C'} \mathit{min\text{-}ov}(c')$.
- Moreover, we use $\mathit{sum\text{-}ov}(c')$ to denote the sum of overlaps along all edges of cycle $c' \in C'$, and, accordingly, $\mathit{sum\text{-}ov}(C')$ to denote the sum of all overlaps along edges within C', i.e., $\mathit{sum\text{-}ov}(C') = \sum_{c' \in C'} \mathit{sum\text{-}ov}(c')$.

We are now ready to prove the claims (i) to (v) above.

Proof of (i): As already discussed after Definition 8.8, an optimal solution for the TSP on the distance graph G for a set of strings S is cheaper than an optimal solution for the SCS for S. As a solution to the TSP is a special type of cycle cover, consisting of a single cycle, a minimum cycle cover is cheaper than an optimal TSP tour. This implies

$$cost(C) = \mathrm{Opt}_{CC}(G_{pref}(S)) \leqslant \mathrm{Opt}_{SCS}(S) \tag{8.3}$$

and

$$cost(C') = \mathrm{Opt}_{CC}(G_{pref}(R)) \leqslant \mathrm{Opt}_{SCS}(R). \tag{8.4}$$

Since $R \subseteq S$, a shortest superstring for R is clearly at least as short as a shortest superstring for S. Together with (8.4) this implies

$$cost(C') = \mathrm{Opt}_{CC}(G_{pref}(R)) \leqslant \mathrm{Opt}_{SCS}(R) \leqslant \mathrm{Opt}_{SCS}(S). \tag{8.5}$$

Proof of (ii): We now determine the length of string $u_{c'}$ computed in step 5. Let $c' = (c'_1, \ldots, c'_{k_{c'}}) \in C'$ and let $(c'_{k_{c'}}, c'_1)$ be the edge with minimal overlap $\textit{min-ov}(c')$. Then,

$$\begin{aligned} u_{c'} &= \langle c'_1, \ldots, c'_{k_{c'}} \rangle \\ &= \textit{Pref}(c'_1, c'_2) \ldots \textit{Pref}(c'_{k_{c'}-1}, c'_{k_{c'}}) c'_{k_{c'}} \\ &= \textit{Pref}(c'_1, c'_2) \ldots \textit{Pref}(c'_{k_{c'}-1}, c'_{k_{c'}}) \textit{Pref}(c'_{k_{c'}}, c'_1)\, Ov(c'_{k_{c'}}, c'_1). \end{aligned}$$

Thus, the length $|u_{c'}|$ of $u_{c'}$ is

$$\begin{aligned} |u_{c'}| &= \textit{pref}(c'_1, c'_2) + \cdots + \textit{pref}(c'_{k_{c'}-1}, c'_{k_{c'}}) + \textit{pref}(c'_{k_{c'}}, c'_1) + ov(c'_{k_{c'}}, c'_1) \\ &= \textit{cost}(c') + \textit{min-ov}(c'). \end{aligned} \tag{8.6}$$

Proof of (iii): Using (8.6), we can compute the length of w' as computed in step 6 of the algorithm.

$$\begin{aligned} |w'| &= \sum_{c' \in C'} |u_{c'}| = \sum_{c' \in C'} \textit{cost}(c') + \sum_{c' \in C'} \textit{min-ov}(c') \\ &= \textit{cost}(C') + \textit{min-ov}(C'). \end{aligned} \tag{8.7}$$

Proof of (iv): In step 7, we replace each representative r_c in w', for any $c = (c_1, \ldots, c_{k_c}) \in C$, by the string

$$\textit{Pref}(r_c, c_2)\textit{Pref}(c_2, c_3)\textit{Pref}(c_3, c_4) \ldots \textit{Pref}(c_{k_c-1}, c_{k_c})\textit{Pref}(c_{k_c}, r_c) r_c.$$

Thus, the length of w' is increased by $\textit{pref}(r_c, c_2) + \textit{pref}(c_2, c_3) + \cdots + \textit{pref}(c_{k_c}, r_c)$, which is exactly $\textit{cost}(c)$.

Therefore, for the length of the computed superstring w, we obtain

$$\begin{aligned} |w| &= |w'| + \sum_{c \in C} \textit{cost}(c) \\ &= |w'| + \textit{cost}(C) \\ &\overset{(8.7)}{=} \textit{cost}(C') + \textit{min-ov}(C') + \textit{cost}(C) \\ &\overset{(8.3,8.5)}{\leqslant} 2 \cdot \mathrm{Opt}_{SCS}(S) + \textit{min-ov}(C') \end{aligned} \tag{8.8}$$

Proof of (v): It remains for us to estimate the total length $\textit{min-ov}(C')$ of the overlap that was lost in Step 5 of the algorithm. Since each cycle consists of at least two edges and since we removed the one with minimal overlap, we can guarantee that

$$\textit{min-ov}(C') \leqslant \frac{1}{2} \cdot \textit{sum-ov}(C'). \tag{8.9}$$

Combining Inequalities (8.8) and (8.9), we obtain

$$|w| \leqslant 2 \cdot \mathrm{Opt}_{SCS}(S) + \frac{1}{2} \cdot \textit{sum-ov}(C'). \tag{8.10}$$

We now have to show that $\textit{sum-ov}(C') \leqslant 2 \cdot \mathrm{Opt}_{SCS}(S)$. For this, we apply the following lemma, which was proved by Blum et al. [33].[9]

Lemma 8.2. *Let c and c' be two different cycles in a minimum cycle cover C, and let $s \in c$ and $s' \in c'$ be two strings in these cycles; then, the overlap of s and s' satisfies $ov(s, s') < cost(c) + cost(c')$.* □

All vertices in a cycle $c' \in C'$ belong to *different* cycles within C. Thus, we can infer from Lemma 8.2 that

$$\textit{sum-ov}(C') \leqslant 2 \cdot cost(C) \tag{8.11}$$

Combining this with Inequalities (8.10) and (8.3) eventually yields our claimed result:

$$|w| \leqslant 2 \cdot \mathrm{Opt}_{SCS}(S) + \frac{1}{2} \cdot \textit{sum-ov}(C') \leqslant 3 \cdot \mathrm{Opt}_{SCS}(S).$$

□

Although, according to this result, Algorithm 8.2 guarantees a better approximation ratio than the previously presented Algorithm 8.1 (*Greedy Superstring*), in practical applications the latter is preferred. This is supported by the conjecture that the algorithm *Greedy Superstring* actually is a 2-approximation algorithm, but mainly it is used because of its easy implementation and its good adaptability, which lead to good results in practice.

Evaluation of the Shortest Superstring Model

For practical applications it is thought necessary to adapt the algorithms in an appropriate way, since the modeling in terms of the shortest common superstring problem clearly still has some drawbacks with respect to our original fragment assembly problem. According to the overview of error sources given in Section 8.1.1, we will now briefly discuss some of these drawbacks.

The modeling in terms of the shortest common superstring problem takes into account neither the problem setting of sequencing errors, nor incomplete coverage, unknown orientation of the strings, or the occurrence of chimeric clones.

If one wants additionally to account for sequencing errors, one has to consider overlaps between strings that allow for a certain number of errors, as discussed in Subsection 8.1.1. The unknown orientation may be modeled by considering either the string itself or its reverse complement for the SCS.

[9] Although the proof of this lemma is not overly complicated, it requires a lot of additional notation and prerequisites; thus, we omit it here for the sake of simplicity.

But a naive approach to this would directly lead to a combinatorial explosion of the number of possibilities, since for each considered string there exist two possibilities.

The lack of data in the case of incomplete coverage of DNA by fragments or the occurrence of chimeric clones can only be eliminated by additional effort – for instance, by preprocessing or by further experiments.

In particular, with respect to repeats, the modeling of fragment assembly in terms of shortest common superstrings seems to be counterintuitive. It may actually lead to the merge of exact repeats (or even inexact repeats if we allow sequencing errors) that occur at different positions within the DNA, since we cannot distinguish between overlaps resulting from real overlaps in the original DNA or those due to repeats occurring at different positions of the DNA. A possible consequence of this focus on the *shortest* superstring according to repeats is illustrated in Figure 8.11. Problems arising in the presence of repeats are also addressed by clever preprocessing of data based on the knowledge of certain repeat sequences.

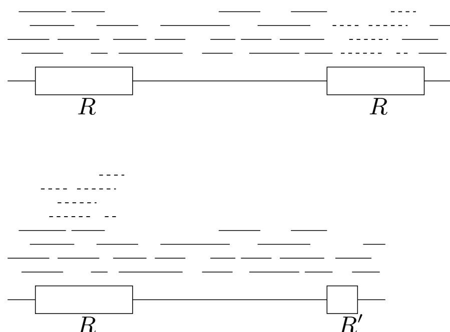


Fig. 8.11. One of the two occurrences of a repeat R may be considerably shortened if fragment assembly is modeled in terms of SCS. Here, the original DNA strand is depicted at the top, and a possible reconstruction based on the SCS at the bottom. Fragments shown as dashed lines will be assigned to only one of the two repeats (here, to the left-hand one) by the SCS algorithm, since they lie completely inside the repeat R

In the following section we will present two models that take into account some of these problems in their formalization of the fragment assembly process.

8.1.3 Refined Models for Fragment Assembly

We will now exemplarily study two refinements of the shortest common superstring approach that will improve applicability in practice.

The Reconstruction Model

In this section we will discuss a model regarding sequencing errors as well as the unknown orientation of the DNA fragments resulting from a shotgun experiment. To model sequencing errors, we have to fix a measure to determine whether two strings are overlapping or not, even if their suffix-prefix pairs do not match exactly, i.e., in the presence of errors. Measures for the similarity of (or the distance between) strings have already been considered in Chapter 5 in the context of the alignment problem. Here, we consider a variant of the *edit distance* that was introduced in Chapter 5. Recall that the edit distance measures the distance between two strings by counting the number of insertions, deletions, and substitutions that are necessary to transform one string into another. We now define the *substring edit distance*, which, in contrast to the original edit distance, does not penalize the deletion of characters at the beginning or end of the second string. Thus, it corresponds to a variant of the semiglobal alignment studied in Section 5.1.3.

Definition 8.11. *Let s and t be two strings over a common alphabet Σ. Then, the* substring edit distance $\mathrm{ed_s}(s,t)$ *between s and t is defined as*

$$\mathrm{ed_s}(s,t) = \min_{x \in \mathrm{SubStr}(t)} \mathrm{ed}(s,x),$$

where $\mathrm{SubStr}(t)$ *denotes the set of all substrings of t, and* ed *denotes the edit distance.*

Due to its relation to semiglobal alignments, the substring edit distance can be efficiently computed.

The idea behind our model is to allow for a certain number of errors, counted according to the previously defined substring edit distance, in the reconstruction of DNA in terms of a "superstring." The threshold on the number of errors will be proportional to the length of the considered string. We refer to this by introducing an *error tolerance value* ε that specifies the average number of errors per character.

Besides sequencing errors, we will also account for the unknown orientation of the fragments by demanding that either the string itself or its reverse complement occurs in the reconstructed DNA sequence. Note that, since we are dealing with the reverse complements of strings, we have to restrict ourselves to strings, where such a reverse complement is properly defined. Therefore we explicitly adhere to the alphabet Σ_{DNA} in the following definitions.

Definition 8.12. *The* reconstruction problem *is the following optimization problem.*

Input: A set of strings $S = \{s_1, \ldots, s_n\}$ over $\Sigma_{\mathrm{DNA}} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and an error tolerance value $\varepsilon \in [0,1]_{\mathbb{R}}$.

Feasible solutions: Each string w, such that, for all $i = 1, \ldots, n$, the following holds:

$$\min\{\mathrm{ed_s}(s_i, w), \mathrm{ed_s}(\overline{s_i}, w)\} \leqslant \varepsilon \cdot |s_i|, \tag{8.12}$$

where $\overline{s_i}$ denotes the reverse complement of s_i.

Costs: The length of w, $cost(w) = |w|$.

Optimization goal: Minimization.

By demanding in Inequality 8.12 the occurrence (with respect to a certain error tolerance) of either the string or its reverse complement, we model the unknown orientation. However, we still do not take into account complications that arise in the presence of chimeric clones, repeats, and incomplete coverage. Before extending the reconstruction problem to additionally account for the issue of incomplete coverage, we will briefly state its complexity. The following theorem was proved by Kececioglu [119].

Theorem 8.8. *The reconstruction problem is NP-hard.* □

A possibility to solve the reconstruction problem is based on an extension of the usual *layout* — *overlap* — *consensus* phases. First, we compute an overlap graph that includes overlaps that respect the above discussed error tolerance but that does not have to be exact. Then, each string gets assigned an orientation and subsequently the phases layout and overlap are performed. For a more detailed discussion of this, we refer the reader to the bibliographic notes in Section 8.4.

The Multicontig Model

The following model will extend the reconstruction problem to account for incomplete coverage. To achieve this, we consider the ordering of strings in a layout. In such a layout, we focus on the overlap between strings and demand that the length of an *important overlap*, one which constitutes the only connection between two blocks of strings, must exceed a certain minimal size. We introduce a threshold T denoting this minimal overlap. At positions where we cannot guarantee the minimal size T for an important overlap, we split the consensus into parts, each called a *contig*. The idea is that, even though incomplete coverage may indeed appear, it does so only rarely. Following this parsimony principle, we aim at minimizing the number of resulting contigs.

We will now formally specify this idea and introduce some additional notation.

Definition 8.13. *Let $S = \{s_1, \ldots, s_n\}$ be a set of strings over an alphabet Σ. Let L be a layout*[10] *of the strings in S, and let c be the corresponding consensus. According to the layout, we can assign a sequence in the consensus*

[10] Recall that we may consider a layout as a semiglobal multiple alignment of the strings.

c to each string $s_i \in S$, which we denote by $c[l(s_i), r(s_i)]$. Here, $l(s_i)$ refers to the start position (left border) and $r(s_i)$ to the end position (right border) of the sequence belonging to s_i in c (as we will account for sequencing errors, s_i is not required to be a substring of c).

We will now specify our notion of an important overlap. If two strings s_i and s_j overlap in L, i.e., if they have a common overlapping region $c[l(s_j), r(s_i)]$, and no further string s_k in L properly covers this overlapping region, i.e., there exists no $k \in \{1, \ldots, n\}$ such that $l(s_k) < l(s_j)$ and $r(s_i) < r(s_k)$, then we call such an overlap a link. *The* size of a link *$c[l(s_j), r(s_i)]$ is defined by its length $|c[l(s_j), r(s_i)]| = r(s_i) - l(s_i) + 1$.*

A link in L with the smallest overlap is called a weakest link.

Let $T \in \mathbb{N}$ be a threshold. If a weakest link in L has at least size T, we call L a T-contig. *We say that the corresponding set S of strings* admits *a T-contig.*

With this definition, we have formalized our demand for reasonably sized important overlaps. It remains for us to include the modeling of sequencing errors and unknown orientation. We have to take into account that, while defining T-contigs, we fixed the position of the appearance of each string in the consensus by fixing its left and right border. Thus, we relinquished the free choice of a potential consensus when comparing the strings to any position, as was the case with the reconstruction problem.

In this context we introduce the following notion.

Definition 8.14. *Let $S = \{s_1, \ldots, s_n\}$ be a set of strings over an alphabet $\Sigma_{\text{DNA}} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Let L be a layout of the strings in S, where either each string s_i or its reverse complement $\overline{s_i}$ occurs, and let c be the corresponding consensus. Moreover, let $\varepsilon \in [0, 1]_{\mathbb{R}}$ be a error tolerance value. We call c an* ε-consensus *if*

$$\text{ed}(s_i, c[l(s_i), r(s_i)]) \leqslant \varepsilon \cdot |s_i|, \text{ if } s_i \in L, \text{and}$$
$$\text{ed}(\overline{s_i}, c[l(\overline{s_i}), r(\overline{s_i})]) \leqslant \varepsilon \cdot |s_i|, \text{ if } \overline{s_i} \in L$$

for all $i = 1, \ldots, n$.

Provided with these auxiliaries, we are ready to define the multicontig problem.

Definition 8.15. *The* multicontig problem *is the following optimization problem.*

Input: A set $S = \{s_1, \ldots, s_n\}$ of strings over an alphabet $\Sigma_{\text{DNA}} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, a threshold $T \in \mathbb{N}$, and an error tolerance value $\varepsilon \in [0, 1]_{\mathbb{R}}$.

Feasible solutions: Each partition of set S into subsets $C_1, \ldots, C_k$ such that, for all $1 \leqslant i \leqslant k$, the following holds:

There exists a layout L_i of C_i, such that

- *L_i is a T-contig, and*

- *L_i induces an ε-consensus.*

Costs: The number of subsets, i.e., $cost(C_1, \ldots, C_k) = k$.
Optimization goal: Minimization.

We are thus looking for a partition of the set of strings such that each of the resulting subsets admits a T-contig, which again induces an ε-consensus. Here, we are aiming at minimizing the number of subsets.

To illustrate this model, we will next present a small example, disregarding the appearance of reverse complements for the sake of convenience.

Example 8.6. Let $S = \{s_1, \ldots, s_8\} = \{$GTAGTA, GCATCC, TCAGTTA, ACGTGA, TGAGC, CTTTGCA, CAGGCA, AGCCATT$\}$ be a set of strings over the alphabet $\Sigma_{\text{DNA}} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Moreover, let $T = 2$ and $\varepsilon = \frac{1}{6}$. A feasible solution for the multicontig problem for the input instance (S, T, ε) is the partition of S into subsets $C_1 = \{\mathtt{GTAGTA}, \mathtt{GCATCC}, \mathtt{TCAGTTA}\}$, $C_2 = \{\mathtt{ACGTGA}, \mathtt{TGAGC}\}$, and $C_3 = \{\mathtt{CTTTGCA}, \mathtt{CAGGCA}, \mathtt{AGCCATT}\}$. To prove this claim, we present a layout L_i for each set C_i, being a T-contig and simultaneously inducing an ε-consensus.

Layout L_1 is shaped as follows:

```
s1  G T A G T A
s2        G C A T C C
s3              T C A G T T A
---------------------------------
c1  G T A G C A T C C G T T A
```

with consensus $c_1 = \mathtt{GTAGCATCCGTTA}$. At positions 5 and 9, the entries in s_1 and s_3 differ from those in c_1. Since s_1 is of length 6 and s_3 is of length 7, both deviations are inside the error tolerance range for $\varepsilon = \frac{1}{6}$. Thus, c_1 is an ε-consensus. The overlap between s_1 and s_2 and the overlap between s_2 and s_3 are links. Both links are of size 3, hence, also weakest links; and because $T = 2$, the layout L_1 is a T-contig.

For the subsequently described layouts L_2 and L_3, one can prove that they are T-contigs and that they induce an ε-consensus in an analogous way.

Layout L_2 is shaped as follows:

```
s4  A C G T G A
s5        T G A G C
---------------------
c2  A C G T T A G C
```

Layout L_3 is shaped as follows:

```
s6  C T T T G C A
s7            C A G G C A
s8              A G C C A T T
---------------------------------
c3  C T T T G C A G G C A T T
```

Note that although the overlap between strings s_6 and s_8 in L_3 has length only 1, the layout L_3 is nevertheless a T-contig, since the overlap region between s_6 and s_8 does not constitute a link as it is completely covered by s_7. ◊

We will not study this model in more detail here, but conclude our discussion of the shotgun sequencing approach. For further information we refer the reader to the literature referenced in Section 8.4.

8.2 Sequencing by Hybridization

In this section, we will present an approach for sequencing DNA that does not depend on the shotgun method, but that tries to reconstruct the DNA sequence from the set of substrings of a certain length that are contained in the sequence.

To test whether certain strings occur as substrings in the considered DNA, we use DNA arrays, as described in Section 7.2. Recall that in this context we use the word *probe* to denote a short DNA sequence that we are testing for being a substring of a larger one.

Method 8.2 Sequencing by Hybridization — SbH

Input: A DNA molecule $\mathcal{D}$ and a natural number l.

1. Generate a DNA chip that contains all different probes of length l, where each probe is assigned to a specific position on the chip.
2. Generate a number of copies of $\mathcal{D}$.
3. Expose the generated copies to the DNA chip and detect the positions where hybridizations between probes and DNA occurred.

Output: The set of strings $S = \{s_1, \ldots, s_n\} \subseteq \Sigma_{\text{DNA}}^l$ of length l that are contained as substrings in $\mathcal{D}$.

The set of strings of length l that we obtain by such a hybridization experiment is called the *l-spectrum* of the DNA; or *spectrum* for short, if l is clear from the context. We note that, ideally, such a spectrum contains exactly those strings of length l that are substrings of the corresponding DNA sequence.[11] In particular, this also means that a string of length l that is not a substring of the DNA is *not* contained in the l-spectrum.

The task is now to reconstruct the original DNA sequence from its spectrum. To formalize this, we first define the compatibility of a string and a spectrum.

Definition 8.16. *Let w be a string and $S = \{s_1, \ldots, s_n\}$ be an l-spectrum. Then, w is* compatible *with S if w contains each string from S as a substring and contains no other substrings of length l. A string w that is compatible with S is* simply compatible *if w contains each string in S exactly once.*

A string is thus compatible with a spectrum if it corresponds to the data given by it. We can now specify our problem setting.

[11] If errors occur during the experiment, this is not the case in general.

Definition 8.17. *The* SbH reconstruction problem *is the following computing problem.*

Input: An l-spectrum $S = \{s_1, \dots, s_n\}$ with strings over an alphabet Σ.
Output: A string $w \in \Sigma^*$ compatible with S.

To solve this problem, we may use the method described in Section 8.1 and consider the spectrum as a set of strings, treating it as an instance of the SCS. With respect to our intended application, this means reconstructing the original DNA sequence using the same approaches as for the SCS. But doing so, we are again faced with an algorithmically hard problem, and we disregard the additional information provided by the l-spectrum, namely, that it contains *all* strings of length l occurring in the DNA.

We will now present an alternative approach to solve the SbH reconstruction problems that, using this information, leads to an efficient algorithm. For this, we first define the representation of a spectrum in terms of a graph, the *spectrum graph.*

Definition 8.18. *Let S be an l-spectrum with strings over an alphabet Σ. The* (l-)spectrum graph $G_{\text{spectrum}}(S) = (V, E, \mathit{label})$ *is defined as follows:*

- *The vertex set corresponds to the set of all strings of length $l-1$ over the alphabet Σ, i.e., $V = \Sigma^{l-1}$.*
- *Two vertices x and y are connected by a directed edge if there exists a string $s \in S$ such that x is a prefix and y is a suffix of s, i.e.,*

  $$E = \{(x, y) \mid x, y \in V, \text{ such that there exists some } s \in S \text{ with } \langle x, y \rangle = s\}.$$

- *Each edge $(x, y) \in E$ is labeled with the last character y_{l-1} of y, i.e., $\mathit{label}((x, y)) = y_{l-1}$.*

For a path $x_1, x_2, \dots, x_k$ in $G_{\text{spectrum}}(S)$, we call pathlabel the string induced by it, i.e., the string corresponding to the first vertex concatenated with the labels along the path,

$$\mathit{pathlabel}(x_1, x_2, \dots, x_k) = x_1 \mathit{label}((x_1, x_2)) \mathit{label}((x_2, x_3)) \dots \mathit{label}((x_{k-1}, x_k)).$$

To illustrate Definition 8.18, we show the spectrum graph for the 3-spectrum $S = \{abc, bbc, bcc, bcd, ccd, cda, cdb, dbb\}$ in Figure 8.12.

For further investigations, we can clearly ignore the vertices of a spectrum graph that are not incident to any edge. Therefore, we omit isolated vertices of the spectrum graph in the following.

We now transfer the SbH reconstruction problem to the problem of finding an Eulerian cycle in a spectrum graph. To do so, we recall some definitions from graph theory. An *Eulerian path* in a directed graph G is a path that traverses each edge in G exactly once. Similarly, an *Eulerian cycle* in a directed

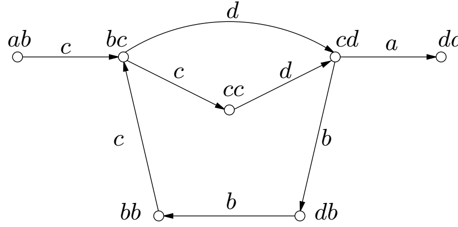


Fig. 8.12. The spectrum graph for the 3-spectrum $S = \{abc, bbc, bcc, bcd, ccd, cda, cdb, dbb\}$. Vertices without incident edges are omitted

graph G is a cycle that traverses each edge of G exactly once.[12] A graph having such an Eulerian cycle is called *Eulerian*.

From the definition of spectrum graph (Definition 8.18) and compatibility (Definition 8.16), we can deduce the following theorem.

Theorem 8.9. *Let S be an l-spectrum with strings over an alphabet Σ. A string w is simply compatible with S if and only if w corresponds to the labeling of an Eulerian path in the spectrum graph $G_{\text{spectrum}}(S)$.*

Proof. To prove this theorem, we first observe that, as a direct consequence of Definition 8.18, for each edge (x, y) in the spectrum graph $G_{\text{spectrum}}(S) = (V, E, \mathit{label})$, $\mathit{pathlabel}(x, y) = x \cdot \mathit{label}((x, y))$ is in the spectrum S.

Now, let w be a simply compatible string to a spectrum S. Then w contains each string in S exactly once as a substring. We can thus construct a path from w by considering the substrings $w[1, l], w[2, l+1], w[3, l+2], \ldots, w[|w| - l+1, |w|]$ of w, where l denotes the length of strings in S, and inserting for each substring $w[i, i+l-1]$ the edge (x, y) into the path, where $\mathit{pathlabel}(x, y) = x \cdot \mathit{label}((x, y)) = w[i, i+l-1]$. Since w is simply compatible with s, each edge of the spectrum graph is traversed by the constructed path exactly once; hence, it is an Eulerian path.

For the reverse direction of the proof, each edge (x, y) of an Eulerian path P in $G_{\text{spectrum}}(S)$ corresponds to a string from the spectrum, namely, to $\mathit{pathlabel}(x, y) = x \cdot \mathit{label}((x, y))$. Merging these strings according to their order in P yields a simply compatible string w. □

Theorem 8.9 implies that, for a spectrum originating from an error-free hybridization experiment and under the assumption that each string in S occurs *exactly once* as a substring in the DNA, the desired DNA sequence corresponds to a labeling of an Eulerian path in the spectrum graph. Each Eulerian path in the spectrum graph hence corresponds to a simply compatible path of the spectrum and vice versa. Our SbH reconstruction problem thus

[12] Note that an Eulerian path/cycle may have to visit the vertices of G more than once.

has an unique solution if and only if the spectrum graph contains a unique Eulerian path.

We illustrate this connection between a simply compatible string for a spectrum and an Eulerian path in the spectrum graph with the following example.

Example 8.7. Let $S = \{abc, bbc, bcc, bcd, ccd, cda, cdb, dbb\}$ be a 3-spectrum. The corresponding spectrum graph is given in Figure 8.12. There exist two simply compatible strings for this spectrum, $w_1 = abcdbbccda$ and $w_2 = abccdbbcda$. The Eulerian paths in $G_{\text{spectrum}}(S)$ are shown in Figures 8.13 (a) and 8.13 (b). $\diamondsuit$

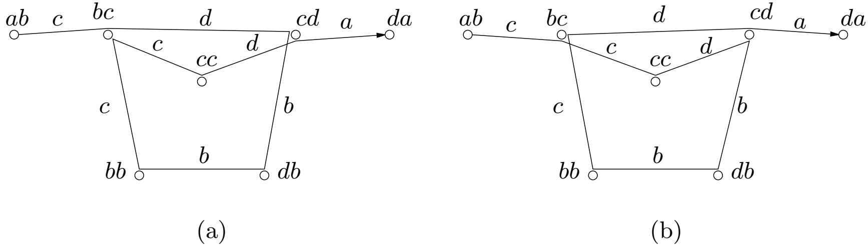


Fig. 8.13. Eulerian paths in the spectrum graph given in Figure 8.12

Before we present an algorithm to compute an Eulerian path in a graph, we first state some relations between Eulerian paths and cycles, and some properties of the considered graphs. We start with the following definition.

Definition 8.19. *Let $G = (V, E)$ be a directed graph and let $v \in V$ be a vertex in G. We call v* balanced *if the indegree of v is the same as the outdegree of v, i.e., if* $\text{indeg}(v) = \text{outdeg}(v)$. *Vertex v is called* semi-balanced *if the indegree and outdegree of v differ by exactly one, i.e., if* $|\text{indeg}(v) - \text{outdeg}(v)| = 1$.

The following theorem presents a well-known result from graph theory. We will therefore omit its proof here and instead refer to standard books on graph theory, such as [58, 81].

Theorem 8.10. *Let $G = (V, E)$ be a connected directed graph. Then,*

(i) G is Eulerian (i.e., G contains an Eulerian cycle) if and only if all vertices in G are balanced, and
(ii) G contains an Eulerian path if and only if G is Eulerian or there exist two semi-balanced vertices a and b such that $\text{indeg}(a) + \text{indeg}(b) = \text{outdeg}(a) + \text{outdeg}(b)$, *and all vertices in $x \in V - \{a, b\}$ are balanced.* $\square$

Algorithm 8.3 Computing an Eulerian cycle

Input: A connected directed graph $G = (V, E)$, whose vertices are all balanced.

Choose a vertex $v_1 \in V$ arbitrarily.
$C := v_1$
while $E \neq \emptyset$ **do**
1. Choose an arbitrary vertex x in C.
2. Starting at x, construct a maximal simple directed path P. This is always a cycle, since all vertices in G are balanced.
3. Set $E := E - \{e \mid e \text{ is an edge in path } P\}$.
4. Merge C and P into a cycle:
   Let $C = x_1, \ldots, x_i, \ldots, x_k, x_1$ and $P = x_i, y_1, \ldots, y_p, x_i$.
   Set $C := x_1, \ldots, x_i, y_1, \ldots, y_p, x_i, x_{i+1}, \ldots, x_k, x_1$.

Output: The Eulerian cycle C.

As a consequence of this theorem, we obtain the following relation. If G is a non-Eulerian graph, but contains an Eulerian path, then we can extend the edge set of G by a directed edge between the semi-balanced vertices a and b, such that all vertices become balanced. In this way, we can obtain an Eulerian graph G' from G.

We now search for an Eulerian path in the spectrum graph according to Theorem 8.9 to obtain a simply compatible string. We may then consider the string as a possible solution to our original DNA reconstruction problem. As we have seen above, we can look for an Eulerian cycle instead of an Eulerian path in an extended spectrum graph. Algorithm 8.3 solves this problem. The idea is to construct a set of arbitrary edge-disjoint cycles, which are subsequently merged into a single cycle in Step 4 of the while loop. We illustrate the merging process in Figure 8.14.

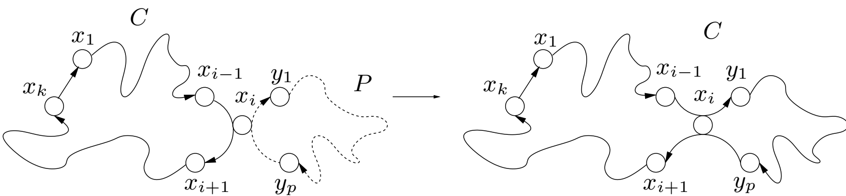


Fig. 8.14. Merging two cycles into a single one according to step 4 of Algorithm 8.3. The cycle C is shown by solid lines and the cycle P by dashed lines

It is easy to recognize that Algorithm 8.3 can be implemented to run in time linear in the number of edges of G. Thus, we will not formally prove this claim, but instead we state the following theorem as a direct consequence of our considerations above.

Theorem 8.11. *There exists a linear-time algorithm to compute a simply compatible string for a given spectrum if possible, i.e., an algorithm that solves the SbH reconstruction problem restricted to* simply *compatible strings.* □

We discuss some shortcomings of this model. To do so, we rephrase our original task. We want to sequence a DNA molecule based on the spectrum of the DNA obtained by a hybridization experiment. This l-spectrum contains all strings of a specific length l that occur as substrings in the DNA sequence. The task is to derive the original DNA sequence from this data.

As we have seen above, a string w compatible with the spectrum corresponds to a DNA sequence consistent with the data. A simply compatible string, i.e., a string containing no string of the spectrum more than once, corresponds to an Eulerian path in the spectrum graph. Eulerian paths in graphs can be easily computed using Algorithm 8.3.

If we have determined such an Eulerian cycle (or the corresponding Eulerian path or simply compatible string), we are equipped with *one* hypothesis for the DNA sequence demanded. However, there might be several admissible hypotheses, since the same graph may contain several different Eulerian cycles.

Moreover, we obtain only *simply* compatible strings, which do not necessarily match the real problem setting, since there may be several positions in the original DNA where a certain substring occurs.

Another problem is caused by the technical realization of this method. In the classical SbH setting, only relatively small probe lengths could be realized on a DNA chip, which allows only sequencing of molecules of about 200 bp length. This is clearly not sufficient, since sequencing the molecules can easily be done using the chain termination method studied in Section 2.4.3. Errors that may occur during the hybridization experiments are also not taken into account. So refined and improved settings are required.

Additionally, the multiplicity of the strings in the spectrum is unknown from the experiment but it is of high interest for the more or less unique reconstruction of the DNA sequence. If the multiplicity were known, we could extend the definition of compatible strings in an appropriate way and thus reduce the set of potential solutions essentially.

Although the above approach seems unsuitable for DNA sequencing today, many approaches for other problems arising in molecular biological research utilize the same basic idea, namely, the usage of hybridization experiments for producing the data. We refer the reader for instance to Section 7.2, where we considered the problem of mapping by hybridization. Furthermore, there exists the possibility to improve the results by applying several different methods to reduce the shortcomings of certain methods. From this point of view, the approach presented appears to be by no means of theoretical interest only.

8.3 Summary

Computing the DNA sequence is a fundamental precondition for further study of biological molecules. But by using the chain termination method we can only sequence relatively short molecules.

For sequencing longer molecules, often the shotgun sequencing method is used, where copies of the DNA molecule, whose sequence we are looking for, are cut into overlapping fragments. The sequence of these (short) fragments is then computed using the chain termination method. Subsequently, we try to reconstruct the original sequence by combinatorial approaches using the fragment sequences. This last step is called the fragment assembly problem, and for its solution often three phases are distinguished. First, one computes the pairwise *overlap* between all fragments; then one determines an overlap structure of all fragments, which is called a *layout*; and, finally, infers a *consensus* from this.

Considering these phases from an algorithmic point of view, one has to keep in mind potential sources of errors and problems arising from the application of the method itself. For instance, sequencing errors, such as insertions, deletions, and substitutions, can occur during sequencing the fragments. Moreover, there is the risk that cloning of the original DNA might lead to contamination with host DNA, that the orientation of the fragments gets lost, that only an incomplete coverage of the original DNA by the fragments occurs, or that repeats in the DNA sequence complicate the reconstruction, and many more things. All this should be taken into account when evaluating the performance of algorithmic approaches to tackle the problem.

A very basic combinatorial model is the *shortest common superstring problem*, SCS for short. Here, one tries to compute the shortest superstring for a given set of strings. Since this problem is known to be NP-hard, one often considers approximation algorithms whose solutions can be evaluated with respect to two different cost measures. These cost measures are on the one hand the length of the computed superstring and, on the other hand, the number of characters saved compared to the trivial solution, i.e., the concatenation of all strings in the given set. We refer to them as the length and compression measures, respectively. Many approximation algorithms for the SCS are based on the representation of the input set in terms of an overlap or distance graph. The complete directed graphs contain, besides the strings themselves modeled as vertices, also the length of the overlap or the distances between the corresponding pairs of strings given as edge weights.

Algorithm 8.1 (Greedy Superstring) chooses a pair of strings with maximal overlap and merges them. This process is iterated until only one superstring remains. It has been shown that this algorithm achieves a 4-approximation with respect to the length measure and a 2-approximation with respect to the compression measure. In contrast, Algorithm 8.2 (CSSA) determines a superstring by computing a cycle cover. The algorithm obtains a 3-approximation.

Two other models of the fragment assembly problem extend this basic concept and take into account sequencing errors, the unknown orientation of the fragments, and a possibly incomplete coverage of the DNA. These models lead to the *reconstruction problem* and the *multicontig problem*.

Besides the shotgun approach, another method for sequencing DNA has been proposed. Here, one tries to reconstruct the DNA utilizing knowledge about the substrings of a specific length that occur in the original DNA. The information about the substrings is obtained by performing hybridization experiments using a DNA chip. For this reason, the method is called *sequencing by hybridization* (SbH). By means of such experiments, substrings of a specific fixed length l that occur in the DNA are derived. The set of these substrings is called the l-spectrum of the DNA and can be represented naturally in terms of a spectrum graph. It is easy to see that an Eulerian path in such a spectrum graph corresponds to a string that is consistent with the spectrum and may thus serve as an admissible hypothesis for the desired DNA sequence. Since Eulerian paths of graphs can be efficiently computed, the resulting combinatorial problem can be solved in reasonable time. Nevertheless, other difficulties arising with this model, lead to the problem that, till now, the application of this method allows only the reliable sequencing of DNA molecules whose length is below the length of molecules that can be directly sequenced using the chain termination method.

8.4 Bibliographic Notes

Several other books on bioinformatics comprehensively deal with the subject of DNA sequencing; we refer the reader to the books by Waterman [201], Setubal and Meidanis [180], Gusfield [91], and Pevzner [159]. A very recommendable presentation of this subject can also be found in the survey by Myers [146], the source of Example 8.1. It also contains further refinements of the models, aiming for better practical applicability.

The shortest common superstring problem was studied by many authors; in particular, Gallant et al. [78] showed the NP-hardness of the problem. Explicit lower bounds on the approximability were given by Vassilevska [195]. Algorithm 8.1 (Greedy Superstring) was proposed and investigated by Tarhio and Ukkonen [190, 192] and Turner [191]. They dealt with efficient implementation of individual steps and proved the 2-approximation with respect to the compression measure. The 3-approximation result goes back to a work by Jenkyns [108] concerning the traveling salesman problem. The proof that Algorithm 8.1 achieves a 4-approximation with respect to the length measure is due to Blum et al. [33]. There, one can also find modified versions of the greedy algorithm and the algorithm based on the cycle cover approach. The particular version of Algorithm 8.2 (CCSA) which we presented here follows the presentation given by Armen and Stein [16, 17]. A multitude of results were obtained by starting from this generic approach to determine a

superstring using a cycle cover, finally resulting in the presentation of a 2.5-approximation algorithm by Sweedyk [187]. The computation of a cycle cover for a given graph is possible in polynomial time, since this problem can be viewed as the assignment problem, which is a special type of matching problem. An efficient algorithm to solve this assignment problem can be found in the book by Papadimitriou and Steiglitz [153]. Note that in the literature cycle covers are often also referred to as 2-factors.

The reconstruction problem, which is more suitable for the intended application of DNA sequencing, was comprehensively studied by Kececioglu and Myers in [120]. The proof of its NP-hardness was given by Kececioglu [119]. Our presentation of the reconstruction and multicontig problem follows the description by Setubal and Meidanis [180].

The method of sequencing by hybridization (SbH) is comprehensively studied in the book by Pevzner [159]. Pevzner [157] moreover showed the relation between solutions to the SbH reconstruction problem and Eulerian paths in the spectrum graph. In the literature, one can also find approaches for refinements of the SbH method that include and utilize knowledge of the approximate position of the substrings or of the usage of universal bases for hybridization experiments; this may lead to a larger number of substrings one can test for. This will yield significant progress in making the SbH more suitable for practice. Examplarily, we refer in this context to the work done by Ben-Dor et al. [25] and Frieze et al. [76].

Part III

Analyzing Biological Data

9

Finding Signals in DNA Sequences

In the previous chapters, we have seen how the sequence of DNA molecules can be determined. Our next question is, how can we find out something about the biological meaning of such DNA sequences. In this chapter, we will deal with some approaches for addressing this problem. To be more concrete, we want to find interesting regions, called *signals*, within a given DNA sequence. Such signals can be restriction sites, binding sites for proteins, or even genes or certain parts of genes.

Most approaches for finding such signals are based on statistical methods. We do not discuss the statistical approaches in detail within this book, but we present some simple examples and try to give an idea of the modeling and the commonly used methods. For a more detailed overview of the statistical methods we refer the reader to the bibliographic notes in Section 9.6.

This chapter is organized as follows. In Section 9.1 we present some methods for finding identical or similar substrings in a given set of DNA sequences, which can be interpreted as signals within the DNA. A special kind of signaling substring, the tandem repeats, is introduced in Section 9.2. The two approaches presented in these sections are of purely combinatorial nature; in contrast, in the following two sections we will deal with two statistical approaches for signal finding, the determination of frequent or rare substrings in Section 9.3 and the use of Hidden Markov Models in Section 9.4. As usual, we will close the chapter with a summary in Section 9.5 and bibliographic remarks in Section 9.6.

9.1 Identical and Similar Substrings

One of the subproblems occurring when attempting to decrypt the biological meaning of a DNA sequence is finding the binding sites. Binding sites are regions within the DNA where a certain protein can attach, for example, for regulating the transcription of a gene. With the currently known experimental methods, it is not possible to determine the binding sites exactly. Such an

experiment rather yields a DNA fragment of length m that contains (with high probability) a binding site of length l, with $l \ll m$.

The execution of several such experiments then leads to the combinatorial problem of finding a (longest) common substring, also called a *magic word* in this context. For this problem, we have presented Algorithm 4.10 in Section 4.5 solving it efficiently using generalized suffix trees.

Unfortunately, this algorithm is not applicable for finding more complicated patterns than simple substrings within the given DNA sequences. But the binding sites of proteins are often made up of several substrings; for example, the restriction site of the restriction enzyme XcmI is of the form $\texttt{CCAN}^9\texttt{TGG}$, where $\texttt{N}^9$ denotes a substring consisting of nine arbitrary nucleotides. There exist even more complicated patterns, where, for example, the length of some part may vary. Currently, there are no approaches known for efficiently and exactly finding such complex signals.

One approach to allow at least for short substrings consisting of arbitrary nucleotides is to allow mismatches within the magic word. Given n DNA sequences, our goal is to find a magic word occurring in all DNA sequences *approximately*, i.e., with a limited number of mismatches. This task can be seen as a special case of a local multiple alignment where no gaps are allowed. To formally describe the problem, we need the following definition.

Definition 9.1. *Let $s = s_1 \dots s_m$ and $t = t_1 \dots t_m$ be two strings of the same length m. The* Hamming distance $d_H(s,t)$ *of s and t is defined as the number of positions $1 \leqslant i \leqslant m$ where $s_i \neq t_i$.*

Using this definition, we can now formalize the problem as follows.

Definition 9.2. *The* consensus string problem *is the following optimization problem:*

Input: A set of n strings $\{s_1, \dots, s_n\} \subseteq \Sigma^m$ and a natural number l.
Feasible solutions: All $(n+1)$-tuples $(t, t_1, \dots, t_n)$, where t_i is a substring of length l of s_i for $1 \leqslant i \leqslant n$. The string $t \in \Sigma^l$ is called the median string.

Costs: The costs of a feasible solution are

$$cost(t, t_1, \dots, t_n) = \sum_{i=1}^{n} d_H(t, t_i).$$

Optimization goal: Minimization.

Unfortunately, the consensus string problem is a hard optimization problem, as shown by the following theorem.

Theorem 9.1. *The consensus string problem is NP-hard.* □

Algorithm 9.1 Consensus string approximation

Input: A set $S = \{s_1, \ldots, s_n\} \subseteq \Sigma^m$ of n strings and two natural numbers l and r.

1. Initialization:
   $c' := \infty$
   $u' := \lambda$
   for $i := 1$ **to** n **do** $v'_i := \lambda$
2. **for all** $(u_1, \ldots, u_r)$, where $u_i \in \Sigma^l$ is a substring of some string in S for all i **do**
   Compute the consensus u of $u_1, \ldots, u_r$ (see Definition 5.11)
   for $i := 1$ **to** n **do**
   Compute a substring v_i of s_i with minimum Hamming distance to u
   $c := \sum_{i=1}^{n} d_H(u, v_i)$
   if $c < c'$ **then**
   $c' := c$
   $u' := u$
   for $i := 1$ **to** n **do** $v'_i := v_i$

Output: $(u', v'_1, \ldots, v'_n)$ with costs c'.

Theorem 9.1 was proved by Li et al. [135]. We do not present the proof here; instead, we describe an approximation algorithm for the consensus string problem. This algorithm, Algorithm 9.1, works as follows. It chooses an r-tuple $(u_1, \ldots, u_r)$ of (not necessarily different) substrings of length l of the given strings $s_1, \ldots, s_n$ and determines via majority voting according to Definition 5.11 the consensus u of these substrings. After that, the algorithm computes for each of the given strings s_i the substring v_i with the smallest Hamming distance to u. The algorithm repeats this procedure for every possible choice of the r-tuple $(u_1, \ldots, u_r)$ and outputs the $v_1, \ldots, v_n, u$ where the sum of the Hamming distances between u and the v_i becomes minimal. The parameter r can be used to influence the approximation ratio of Algorithm 9.1, as shown by the following theorem.

Theorem 9.2. *For each constant $r \geqslant 3$, Algorithm 9.1 is a polynomial approximation algorithm for the consensus string problem with an approximation ratio of*

$$1 + O\left(\sqrt{\frac{\log r}{r}}\right)$$

and a running time in $O(n^{r+1} \cdot (m - l + 1)^{r+1} \cdot l)$.

Theorem 9.2 was also proved by Li et al. [134, 135]. Since the complete proof of the achieved approximation ratio is technically very involved, we just present the idea behind the proof.

Proof idea. We first determine the running time of Algorithm 9.1. The initialization in step 1 can obviously be done in $O(n)$ time. We now count

the number of iterations of step 2. There are n strings in S and $m - l + 1$ possible starting positions in each of the strings from S for each u_i. Hence, there are $(n \cdot (m - l + 1))^r$ ways to choose the r-tuple $(u_1, \ldots, u_r)$. Thus, the algorithm executes $(n \cdot (m - l + 1))^r$ iterations of step 2. In each of these iterations, the algorithm computes the consensus of the strings $u_1, \ldots, u_r$; this is possible in $O(r \cdot l)$ time. The computation of a substring v_i with minimum Hamming distance from u can be done in $O(l \cdot (m - l + 1))$ time, since one has to calculate the sum of l values for $m - l + 1$ possible starting positions. The computation of all v_i is possible in $O(n \cdot l \cdot (m - l + 1))$ time. Computing c clearly can be done in time $O(n \cdot l)$. This yields an overall running time in $O(n \cdot l \cdot (m - l + 1))$ for one iteration of step 2. This estimate follows from the fact that we may assume $r < n$, since for $r \geqslant n$ the algorithm checks all feasible solutions for the consensus string problem and thus computes an exact solution. Hence, the overall running time of the algorithm sums up to $O(n + (n \cdot (m - l + 1))^r \cdot (n \cdot l \cdot (m - l + 1))) = O(n^{r+1}(m - l + 1)^{r+1} l)$.

We will now describe the idea of estimating the approximation ratio of Algorithm 9.1. Consider an input instance consisting of a set $S = \{s_1, \ldots, s_n\} \subseteq \Sigma^m$ of n strings and two natural numbers l and r. Let $(s, t_1, \ldots, t_n)$ be an optimal solution for this input with cost $c_{\text{opt}} = \sum_{i=1}^n d_H(s, t_i)$.

For all $(i_1, \ldots, i_r) \in \{1, \ldots, n\}^r$, let $s_{i_1,\ldots,i_r}$ be a consensus of $t_{i_1}, \ldots, t_{i_r}$, and let $c_{i_1,\ldots,i_r} = \sum_{i=1}^n d_H(s_{i_1,\ldots,i_r}, t_i)$. The idea now is to approximate the optimal consensus with an $s_{i_1,\ldots,i_r}$ for an appropriate r-tuple $(i_1, \ldots, i_r) \in \{1, \ldots, n\}^r$.

It is possible to show that, for r numbers $i_1, \ldots, i_r \in \{1, \ldots, n\}$ chosen independently and uniformly at random, the expected value of $c_{i_1,\ldots,i_r}$ can be estimated as follows:

$$E[c_{i_1,\ldots,i_r}] \leqslant \left(1 + O\left(\sqrt{\frac{\log r}{r}}\right)\right) \cdot c_{\text{opt}}. \tag{9.1}$$

Inequality (9.1) was proved by Li et al. [135]; we skip the technically involved proof here.

Inequality (9.1) implies that there exist $i'_1, \ldots, i'_r \in \{1, \ldots, n\}$ such that

$$c_{i'_1,\ldots,i'_r} \leqslant \left(1 + O\left(\sqrt{\frac{\log r}{r}}\right)\right) \cdot c_{\text{opt}}.$$

Since our algorithm considered all possible r-tuples of substrings, it considered also the r-tuple $(t_{i'_1}, \ldots, t_{i'_r})$. Thus, the solution given by the algorithm is at least as good as the solution $(s_{i'_1,\ldots,i'_r}, t_{i'_1}, \ldots, t_{i'_r})$, which implies the claimed approximation ratio. □

Note that the approximation ratio of Algorithm 9.1 gets better with increasing values of r, and that the algorithm even computes the optimal solution for $r \geqslant n$. An algorithm, for which one can force an arbitrarily good approximation by changing a parameter, such that the running time stays

polynomial in the input length but may increase exponentially in the size of the parameter, is called a *polynomial-time approximation scheme (PTAS).* This means that for a PTAS one can start with a desired prescribed approximation ratio δ and can then compute the mimimum value of the parameter that is sufficient to ensure the ratio δ.

9.2 Tandem Repeats

Another type of interesting structures within the DNA sequence are repeats, i.e., repeated substrings. In Section 4.5.4 we have seen how to efficiently compute all repeats in a given string using suffix trees. In this section we want to investigate a special type of repeats, the *tandem repeats.* In its simplest form, a tandem repeat consists of two consecutive copies of the same substring, i.e., it is of the form xx for some substring x. In a broader sense, sequences of several consecutive copies of the same substring, i.e., substrings of the form x^i for some substring x and some $i \geqslant 2$, are called tandem repeats. Furthermore, DNA sequences often contain *approximate tandem repeats*, i.e., sequences of consecutive similar substrings.

Identifying tandem repeats is of biological interest for several reasons. On the one hand, some genetic diseases are caused by a defect in the DNA where a certain substring in a tandem repeat is repeated ten, or a hundred, times more often than in the corresponding tandem repeat in the DNA of a healthy organism. On the other hand, tandem repeats play an important role in gene regulation. Moreover, tandem repeats are often used as markers for DNA fingerprinting, since the number of repetitions in a tandem repeat often varies from one individual to another.

In the following, we present an efficient method for determining all exact tandem repeats of the form xx in a given string. For this, we first have to solve the following subproblem: For two positions in a given string s, find the longest substring starting at both positions in s.

Definition 9.3. *Let $s = s_1 \dots s_n$ be a string and let $l, r \in \{1, \dots, n\}$ be two positions in s. The* longest common extension $lce(l, r)$ *of l and r in s is defined as the longest substring $t_1 \dots t_k$ of s such that $t_1 \dots t_k = s_l \dots s_{l+k-1} = s_r \dots s_{r+k-1}$.*

Determining the longest common extension for a single pair of positions (l, r) can obviously be done in time $O(n)$ by comparing iteratively, for all $k \geqslant 0$, the symbols s_{l+k} and s_{r+k} until a mismatch occurs. But for the efficient computation of tandem repeats, we have to compute the longest common extension for many pairs of positions in a given string. This problem can be formalized as follows.

Definition 9.4. *The problem of computing the longest common extension for a set of position pairs in a string, the longest common extension problem, is the following computing problem:*

Input: *A string* $s = s_1 \dots s_n$ *of length* n *over an alphabet* Σ *and a set of position pairs* $P = \{(l_1, r_1), \dots, (l_p, r_p)\}$, *where* $(l_i, r_i) \in \{1, \dots, n\}^2$, $l_i < r_i$, *for all* $1 \leqslant i \leqslant p$.
Output: *The longest common extensions* $lce(l_1, r_1), \dots, lce(l_p, r_p)$.

The longest common extension problem is obviously solvable in $O(n \cdot p)$ time. But using suffix trees, we can design a more efficient algorithm such that, after constructing the suffix tree and some further preprocessing in time $O(n \log n)$, computing the longest common extension for two arbitrary positions is possible in constant time. Recall that a suffix tree for a string s is a rooted tree where each suffix of s is assigned to a leaf, and where the labeling of the path from the root to a leaf equals the suffix of s corresponding to the leaf. The label of the path from the root to a vertex x is denoted by $pathlabel(x)$. A detailed description of suffix trees and their applications can be found in Sections 4.4 and 4.5.

The additional preprocessing required is based on the following fact.

Lemma 9.1. *Let* $s = s_1 \dots s_n$ *be a string,* l *and* r *be two positions in* s, *and* T *be a compact suffix tree for* s. *Let* x *be the lowest common ancestor*[1] *of the two leaves in* T *labeled* l *and* r. *Then,*

$$lce(l, r) = pathlabel(x).$$

Proof. Let u be the leaf in T labeled l, and let v be the leaf in T labeled r. Then, $pathlabel(u) = s_l \dots s_n$ and $pathlabel(v) = s_r \dots s_n$ hold. According to the definition of a suffix tree, the label of the path to the lowest common ancestor x of u and v is the longest common prefix of $pathlabel(u)$ and $pathlabel(v)$. Thus, $lce(l, r) = pathlabel(x)$. □

The basic idea of an efficient method for solving the longest common extension problem is shown in Algorithm 9.2.

Theorem 9.3. *The longest common extension problem for a string* s *of length* n *and* p *position pairs is solvable in* $O(n \log n + p + L)$ *time, where* L *denotes the length of the output, i.e., the total length of all longest common extensions.*

Proof idea. To prove the claim of the theorem, we consider Algorithm 9.2. According to Lemma 9.1, it is obvious that the algorithm solves the longest common extension problem.

We now estimate the running time of Algorithm 9.2. Computing the compact suffix tree in step 1 is possible in $O(n \log n)$ time. The computation of the data structure in step 2 can also be done in $O(n \log n)$ time. But since the construction is technically very involved, we will not present the details here. We refer the reader to the bibliographic notes at the end of this chapter.

---

[1] For any two vertices u and v in a suffix tree, the *lowest common ancestor* in T is the root of the smallest subtree of T containing both u and v.

Algorithm 9.2 Computing the longest common extensions

Input: A string $s = s_1 \dots s_n$ of length n over an alphabet Σ and a set of position pairs $P = \{(l_1, r_1), \dots, (l_p, r_p)\}$, where $(l_i, r_i) \in \{1, \dots, n\}^2$, $l_i < r_i$, for all $1 \leqslant i \leqslant p$.

1. Compute a compact suffix tree T for s.
2. Compute an additional data structure with which it is possible to compute the lowest common ancestor x of any pair (u, v) of leaves in T in constant time.
3. **for** $i = 1$ **to** p **do**
   Determine the lowest common ancestor x_i of the two leaves with labels l_i and r_i in T.
   Define $lce(l_i, r_i) := pathlabel(x_i)$.

Output: The longest common extensions $lce(l_1, r_1), \dots, lce(l_p, r_p)$.

The very rough idea of the construction is based on the fact that the regular structure of a binary tree enables us to assign labels to the vertices such that the lowest common ancestor of two vertices can be found by just comparing the labels. With a technically involved construction, it is possible to embed such a binary tree into the given suffix tree and to use it for the computation of the lowest common ancestor.

Due to the preprocessing in steps 1 and 2, one iteration of the loop in step 3 needs constant time for the lookup plus time proportional to the length of the longest common extension for outputting it. Thus, Step 3 can be executed in $O(p+L)$ time. Overall, the algorithm hence has a running time in $O(n \log n + p + L)$. □

In the following we will employ this solution to the longest common extension problem for designing an efficient algorithm for computing all exact tandem repeats of the form xx in a given string. We start by defining this problem more precisely.

Definition 9.5. *The problem of computing all exact tandem repeats in a string, the* tandem repeat problem, *is the following computing problem:*

Input: A string $s = s_1 \dots s_n$.
Output: The set of all pairs (i, k) *with* $i, k \in \{1, \dots, n\}$, *such that*

$$s_i \dots s_{i+k-1} = s_{i+k} \dots s_{i+2k-1}.$$

The idea behind the algorithm for computing all tandem repeats is based on the principle of "divide and conquer." Here, the given problem is divided into smaller subproblems that are easier to solve, and subsequently the solutions for the subproblems are merged together to yield a solution for the original problem. In our case, the problem can be divided in the following way. Let $s = s_1 \dots s_n$ be the given string and let $h = \lfloor \frac{n}{2} \rfloor$. Then, the problem

of determining all tandem repeats in s is equivalent to solving the following four subproblems:

1. Compute all tandem repeats that are completely included in the first half of s, i.e., in $s_1 \ldots s_h$.
2. Compute all tandem repeats that are completely included in the second half of s, i.e., in $s_{h+1} \ldots s_n$.
3. Compute all tandem repeats whose first half includes the position h, i.e., all tandem repeats (i, k) with $i \leqslant h \leqslant i + k - 1$.
4. Compute all tandem repeats whose second half includes the position h, i.e., all tandem repeats (i, k) with $i + k \leqslant h \leqslant i + 2k - 1$.

The first two subproblems are of the same form as the original problem; they can be solved recursively using the same method. The third and the fourth subproblems are obviously symmetric to each other; thus, it suffices to present a solution to the third subproblem in the following.

To do so, we introduce a method for determining all tandem repeats of fixed length whose first half includes the position h. Iterating over all possible lengths then solves the third subproblem. First, we need another definition.

Definition 9.6. *For a string $s = s_1 \ldots s_n$ and two positions $i, j \in \{1, \ldots, n\}$, we denote by $\overline{lce}(i, j)$ the longest substring $t = t_1 \ldots t_k$ such that $t_1 \ldots t_k = s_{i-k+1} \ldots s_i = s_{j-k+1} \ldots s_j$, i.e., the longest common extension from the positions i and j to the left.*

Using this definition we can formulate the following property, valid for all tandem repeats of fixed length $2l$ whose first half includes the position h.

Lemma 9.2. *Let $1 \leqslant l \leqslant \lfloor \frac{n-h}{2} \rfloor$ and let $q = h + l$. Let $l_1 = |lce(h, q)|$ and let $l_2 = |\overline{lce}(h-1, q-1)|$. Then, the following hold:*

(i) If there exists a tandem repeat of length $2l$ whose first half contains the position h, then $l_1 + l_2 \geqslant l$.
(ii) If $l_1 + l_2 \geqslant l$ holds, then those tandem repeats of length $2l$ whose first half contains the position h start at positions $h - l_2, \ldots, h + l_1 - l$.

Proof. The preconditions of the lemma are depicted graphically in Figure 9.1. Based on this figure, we can now prove the claims of the lemma.

(i) If there exists a tandem repeat of length $2l$ whose first half contains the position h, then $q = h+l$ is the corresponding position in the second half of the tandem repeat. Hence, there exists a common substring in s starting at positions h and q that corresponds to the suffix of the two halves within the tandem repeat. This substring has length at most $l_1 = |lce(h, q)|$. Furthermore, there exists a common substring in s ending at positions $h - 1$ and $q - 1$ that correspond to the prefixes of the two halves within the tandem repeat. This substring has length at most $l_2 = |\overline{lce}(h-1, q-1)|$.

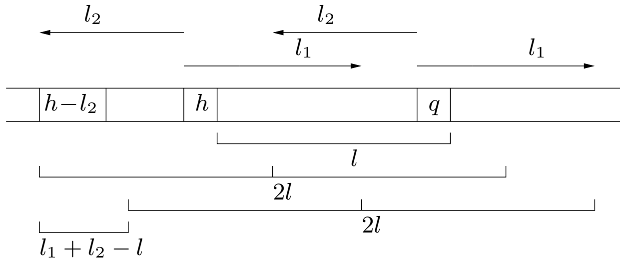


Fig. 9.1. The situation in the proof of Lemma 9.2

Since every position between h and q belongs to one of the two halves within the tandem repeat, the substring between h and q has to be covered by the suffix of the first half and the prefix of the second half; this implies $l_1 + l_2 \geqslant l$.

(ii) On the other hand, if we assume that $l_1 + l_2 \geqslant l$ holds, then we have the situation shown in Figure 9.1. From the figure it is apparent that the substrings $s_{h-l_2} \dots s_{h+l_1-1}$ and $s_{q-l_2} \dots s_{q+l_1-1}$ are equal. This results in $l_1 + l_2 - l$ different tandem repeats of length $2l$ starting at positions $h - l_2, \dots, h - l_1 + l$. □

Algorithm 9.3 employs Lemma 9.2 to solve the third subproblem for computing the tandem repeats.

Algorithm 9.3 Solving the third subproblem for the computation of tandem repeats

---

Input: A string $s = s_1 \dots s_n$.

1. Compute a compact suffix tree and a data structure for efficiently computing the longest common extensions for s and for $\overline{s} = s_n \dots s_1$ (needed for efficiently computing $\overline{lce}$).
2. $R := \emptyset$
   $h := \lfloor \frac{n}{2} \rfloor$
   for $l = 1$ **to** $\lfloor \frac{n-h}{2} \rfloor$ **do**
       $q := h + l$
       $l_1 := lce(h, q)$
       $l_2 := \overline{lce}(h-1, q-1)$
       if $l_1 + l_2 \geqslant l$ **then**
           for $j := h - l_2$ **to** $h + l_1 - l$ **do**
               $R := R \cup \{(j, l)\}$

Output: The set R of all tandem repeats whose first half contains position h in s.

Algorithm 9.4 Computing all tandem repeats

Input: A string $s = s_1 \dots s_n$.

- $R := \emptyset$
- Call the recursive procedure $tandemrepeat(s_1 \dots s_n)$.

Output: The set R of all tandem repeats in s.

Procedure $tandemrepeat(s_1 \dots s_n)$:

if $n > 1$ **then**

- $h := \lfloor \frac{n}{2} \rfloor$
- $tandemrepeat(s_1 \dots s_h)$
- $tandemrepeat(s_{h+1} \dots s_n)$
- Compute all tandem repeats in $s_1 \dots s_n$ whose first half contains the position h using Algorithm 9.3 and add them to the set R.
- Compute all tandem repeats in $s_1 \dots s_n$ whose second half contains the position h using Algorithm 9.3 on the input instance $s_n \dots s_1$, and add them to the set R.

Theorem 9.4. *Algorithm 9.3 computes all tandem repeats in a string* $s = s_1 \dots s_n$ *whose first half contains the position* $h = \lfloor \frac{n}{2} \rfloor$ *in* s *in* $O(n \log n + p)$ *time, where* p *is the number of such tandem repeats.*

Proof. The correctness of Algorithm 9.3 directly follows from Lemma 9.2. Step 1 of the algorithm is possible in $O(n \log n)$ time according to Theorem 9.3. Since every tandem repeat is found and output exactly once by Algorithm 9.3, the execution of all inner for loops in step 2 needs a time in $O(p)$. Using the data structures constructed in step 1, computing l_1 and l_2 is possible in constant time within each execution of the loop in step 2. This implies that all operations within one execution of the outer for loop in step 2 can be done in constant time, excluding the inner for loop for inserting the found tandem repeats into the set R. Hence, the second step of the algorithm needs time in $O(\lfloor \frac{n-h}{2} \rfloor + p) = O(n + p)$. This directly implies the claimed overall running time for Algorithm 9.3. □

Algorithm 9.4 now summarizes the complete procedure for computing all tandem repeats.

Theorem 9.5. *Algorithm 9.4 solves the tandem repeat problem for a string of length* n *containing* p *tandem repeats in* $O(n \cdot (\log n)^2 + p)$ *time.*

Proof. Following the above discussion it is obvious that Algorithm 9.4 solves the tandem repeat problem. In the following, we analyze its time complexity on an input instance $s = s_1 \dots s_n$ containing p tandem repeats. Since the algorithm computes each tandem repeat of s exactly once, it needs $O(p)$ time for updating the set R. We now estimate the running time of all other operations of the algorithm.

One call of the procedure *tandemrepeat* for a string of length m, excluding the recursive procedure calls, needs additional time in $O(m \log m)$, as shown in the proof of Theorem 9.4. The length of the considered string is divided by two with each recursive call of the procedure *tandemrepeat*; thus, the algorithm has a recursion depth of $\lceil \log_2 n \rceil$. On the i-th level of the recursion, there are 2^i subproblems of size $\leqslant \lceil \frac{n}{2^i} \rceil$ that have to be solved. This implies that, for some constant $c \geqslant 1$, the overall complexity of the algorithm can be estimated by

$$\begin{aligned} p + \sum_{i=0}^{\lceil \log_2 n \rceil} 2^i \cdot \left(c \cdot \frac{n}{2^i} \cdot \log_2 \left(\frac{n}{2^i} \right) \right) &= p + c \cdot \sum_{i=0}^{\lceil \log_2 n \rceil} n \cdot \log_2 \left(\frac{n}{2^i} \right) \\ &\leqslant p + c \cdot \lceil \log_2 n \rceil \cdot n \log_2 n \\ &\in O(n \cdot (\log n)^2 + p). \end{aligned}$$

□

Theorem 9.5 shows that Algorithm 9.4 achieves a significant improvement over a naive algorithm for computing all tandem repeats. Such a naive algorithm would check for the existence of a tandem repeat for each length and each starting position. This would lead to a running time in $O(n^3)$, since every single test needs $O(l)$ time, and l can be as large as $\frac{n}{2}$.

Computing approximate tandem repeats can be done using a similar divide and conquer approach. In this case, computing the longest common extensions has to be replaced by computing the longest similar extensions, for example, with a variant of the alignment methods presented in Section 5.1. Details of this approach can be found in the literature cited in Section 9.6.

9.3 Frequent and Infrequent Substrings

Another way to find interesting sections in a DNA sequence is by analyzing the frequencies of the occurring substrings. This approach is based on the idea that substrings occurring significantly more frequently or significantly less frequently in the DNA sequence than would be expected in a random string of the same length could point to coding regions of the DNA. One example for the biological relevance of infrequent substrings occurs in the DNA of many bacteriophages. In this DNA, substrings of length four or six corresponding to binding sites of restriction enzymes occur significantly less frequently than would be expected in random strings. On the other hand, a significantly higher frequency can point to biologically meaningful regions, for example, to regulatory regions involved in gene expression.

Our goal in this section is to compare, for every substring t of length l in a given string s of length m, the number of occurrences of t in s with the expected number of occurrences of t in a random string of length m. To decide

Algorithm 9.5 Frequency Analysis for Substrings

Input: A string s of length m over an alphabet Σ and a natural number l.

for all $t \in \Sigma^l$ **do**

- Compute the number $h(t)$ of occurrences of t in s.
- Compute the expected number of occurrences of t in a random string of length m and its variance.

Output: All strings t of length l, whose actual frequency in s differs significantly from the expected frequency in a random string.

---

if the actual frequency significantly differs from the expected frequency in a random string, we calculate not only the expected value of the frequency in a random string but also its variance. This method is shown schematically in Algorithm 9.5.

Recall that the variance of a random variable X is defined as

$$Var(X) = E[X^2] - E[X]^2,$$

where $E[X]$ denotes the expectation of X. This means that the variance measures how much the value of the random variable oscillates around the expected value. If the variance is large, it might happen that the value of the random variable differs a lot from the expected value. If we want to compare the measured value of a random variable with its expected value, the variance gives us a measure for the significance of this difference.

Although the expected frequency of a substring t of length l in a random string[2] of length m depends on the lengths l and m only, the variance depends on the string t itself. The reason for this is that the number of possible self-overlaps differs for different strings. This implies that the variance for the frequency of self-overlapping substrings like *aaa* differs completely from the variance for non-self-overlapping substrings like *abc*. To describe this more formally, we need the following definition.

Definition 9.7. *Let $t = t_1 \dots t_l$ be a string of length l. The* autocorrelation *of t is a binary string $c(t) = c_0^{(t)} \dots c_{l-1}^{(t)}$, where*

$$c_i^{(t)} = \begin{cases} 1 & \textit{if } t_1 \dots t_{l-i} = t_{i+1} \dots t_l, \\ 0 & \textit{otherwise.} \end{cases}$$

Intuitively, $c_i^{(t)}$ indicates whether the prefix of t of length $l - i$ coincides with the suffix of t of length $l - i$.

[2] By a random string of length m over an alphabet Σ, we mean a string of length m where at each position one character from Σ is chosen uniformly and independently at random. Such a string is also called a *Bernoulli string*.

i	$t_1 \dots t_{l-i}$	$t_{i+1} \dots t_l$	$c_i^{(t)}$
0	*ababa*	*ababa*	1
1	*abab*	*baba*	0
2	*aba*	*aba*	1
3	*ab*	*ba*	0
4	*a*	*a*	1

Table 9.1. The autocorrelation for the string $t = ababa$ from Example 9.1

The autocorrelation polynomial *of t is defined as*

$$corr_t(x) = \sum_{i=0}^{l-1} c_i^{(t)} \cdot x^i.$$

Note that $c_0^{(t)} = 1$ holds for any string t. We illustrate this definition with an example.

Example 9.1. Consider $t = ababa$. The values $c_i^{(t)}$ and the corresponding prefixes and suffixes of t are shown in Table 9.1. The corresponding autocorrelation polynomial for t is $corr_t(x) = 1 + x^2 + x^4$. ♢

The following theorem describes how to compute the expected value and the variance of the number of occurrences of a pattern $t = t_1 \dots t_l$ in a random string $s = s_1 \dots s_m$. For simplicity, we assume that the random string s is cyclic, i.e., we also count the pattern occurrences of the form $t_1 \dots t_l = s_{m-j+1} \dots s_m s_1 \dots s_{l-j}$.

Theorem 9.6. *Let Σ be an alphabet of size k and let $s = s_1 \dots s_m$ be a cyclic Bernoulli string of length m, i.e., let every s_i be chosen uniformly and independently at random from Σ. Let $t \in \Sigma^l$ be a pattern we want to search for in s.*

Consider, for $1 \leqslant i \leqslant m$, the random variable X_i defined by

$$X_i = \begin{cases} 1 & \text{if } t \text{ starts at position } i \text{ in } s, \\ 0 & \text{otherwise.} \end{cases}$$

Then the number of occurrences of t in s is given by the random variable $X = \sum_{i=1}^{m} X_i$, and we can compute the expectation and the variance of X as follows:

(a) $E[X] = m \cdot p$, where $p = E[X_i] = \frac{1}{k^l}$, and
(b) $Var[X] = pm \cdot \left(2corr_t\left(\frac{1}{k}\right) - (2l-1)p - 1\right)$.

Proof. (a) Since s is a Bernoulli string over an alphabet of size k and $|t| = l$ holds, it is immediately clear that $p = E[X_i] = \frac{1}{k^l}$. Due to the linearity of expectation, we know $E[X] = \sum_{i=1}^{m} E[X_i] = m \cdot p$.

(b) The variance $Var[X]$ of the random variable X can be computed as $Var[X] = E[X^2] - E[X]^2$, where

$$E[X^2] = E\left[\sum_{1\leqslant i,j\leqslant m} X_i \cdot X_j\right] = \sum_{1\leqslant i,j\leqslant m} E[X_i \cdot X_j]$$

and

$$E[X]^2 = \left(\sum_{i=1}^{m} E[X_i]\right)^2 = \sum_{1\leqslant i,j\leqslant m} E[X_i]\cdot E[X_j].$$

Hence,

$$Var[X] = E[X^2] - E[X]^2 = \sum_{1\leqslant i,j\leqslant m} (E[X_i \cdot X_j] - E[X_i]\cdot E[X_j]).$$

Let $d(i,j)$ be the shortest distance between the positions i and j in the cyclic string s. Then the following holds:

$$\begin{aligned} Var[X] = & \sum_{\substack{1\leqslant i,j\leqslant m \\ d(i,j)\geqslant l}} (E[X_i \cdot X_j] - E[X_i]\cdot E[X_j]) \\ & + \sum_{\substack{1\leqslant i,j\leqslant m \\ d(i,j)=0}} (E[X_i \cdot X_j] - E[X_i]\cdot E[X_j]) \\ & + \sum_{\substack{1\leqslant i,j\leqslant m \\ 0<d(i,j)<l}} (E[X_i \cdot X_j] - E[X_i]\cdot E[X_j]) \end{aligned}$$

We will now determine the values of these three sums separately. For the first sum we have

$$\sum_{\substack{1\leqslant i,j\leqslant m \\ d(i,j)\geqslant l}} (E[X_i \cdot X_j] - E[X_i]\cdot E[X_j]) = 0,$$

since the random variables X_i and X_j are independent in this case. Moreover,

$$\begin{aligned} & \sum_{\substack{1\leqslant i,j\leqslant m \\ d(i,j)=0}} (E[X_i \cdot X_j] - E[X_i]\cdot E[X_j]) \\ & \qquad = \sum_{1\leqslant i\leqslant m} (E[X_i \cdot X_i] - E[X_i]\cdot E[X_i]) = m(p - p^2). \end{aligned}$$

Now it remains for us to calculate the last sum. We know that

$$\begin{aligned} & \sum_{\substack{1\leqslant i,j\leqslant m \\ 0<d(i,j)<l}} (E[X_i \cdot X_j] - E[X_i]\cdot E[X_j]) \\ & \qquad = \sum_{i=1}^{m}\sum_{r=1}^{l-1}\sum_{|i-j|=r} (E[X_i \cdot X_j] - E[X_i]\cdot E[X_j]). \end{aligned}$$

If $c_r^{(t)} = 0$, then $X_i \cdot X_{i+r} = 0$ holds for all i. But if $c_r^{(t)} = 1$ holds, then it is possible that t starts in s at positions i and $i + r$. In this case, the expected value $E[X_i \cdot X_{i+r}]$ is the product of the probability p that t starts at position $i + r$ and the probability that $s_i \dots s_{i+r-1} = t_1 \dots t_r$ holds. Hence,

$$E[X_i \cdot X_{i+r}] = c_r^{(t)} \cdot p \cdot \frac{1}{k^r}.$$

For every i there are exactly two positions j with $d(i, j) = r$; hence,

$$\sum_{i=1}^{m} \sum_{r=1}^{l-1} \sum_{|i-j|=r} E[X_i \cdot X_j] = \sum_{i=1}^{m} \sum_{r=1}^{l-1} 2p \cdot c_r^{(t)} \cdot \frac{1}{k^r} = \sum_{i=1}^{m} 2p \cdot \left(corr_t \left(\frac{1}{k} \right) - 1 \right).$$

Furthermore,

$$\sum_{r=1}^{l-1} \sum_{|i-j|=r} E[X_i] \cdot E[X_j] = \sum_{r=1}^{l-1} \sum_{|i-j|=r} p^2 = 2(l-1)p^2.$$

This implies

$$\begin{aligned}
&\sum_{i=1}^{m} \sum_{r=1}^{l-1} \sum_{|i-j|=r} (E[X_i \cdot X_j] - E[X_i] \cdot E[X_j]) \\
&= \sum_{i=1}^{m} \left(2p \cdot \left(corr_t \left(\frac{1}{k} \right) - 1 \right) - 2(l-1)p^2 \right) \\
&= pm \cdot \left(2\,corr_t \left(\frac{1}{k} \right) - 2 - 2(l-1)p \right).
\end{aligned}$$

Overall, we get

$$\begin{aligned}
Var[X] &= pm \cdot \left(2\,corr_t \left(\frac{1}{k} \right) - 2 - 2(l-1)p \right) + m(p - p^2) \\
&= pm \cdot \left(2\,corr_t \left(\frac{1}{k} \right) - (2l-1)p - 1 \right).
\end{aligned}$$

□

Using the formulas proved in Theorem 9.6, we can now execute the necessary calculations for Algorithm 9.5, and we obtain a method for choosing the substrings of a given (short) length in a given string, that occur significantly more frequently or significantly less frequently than expected in a random string.

9.4 Hidden Markov Models

In the previous section we analyzed the frequency of short substrings within random strings, and we used this analysis to find highly non-random regions within an entire DNA sequence. But it might also happen that a certain substring occurs rarely within some regions of the DNA sequence, but significantly more frequently in other regions. Aa an example, we consider the so-called CG-islands. The string CG is the rarest dinucleotide in general within many genomes, but in some regions of the DNA the substring CG occurs very frequently. The regions containing many CG-dinucleotides are called CG-*islands*. CG-islands are of biological interest, since in many genomes they occur in proximity to the coding regions, and thus can be used for predicting the locations of genes within the DNA sequence.

In this section we present a method that allows us to find the CG-islands in a given DNA sequence. As a tool, we will use *Hidden Markov Models*, which can informally be described as a kind of probabilistic finite automata that, in contrast to common finite automata, do not read an input but output a symbol at each computation step. More formally, a Hidden Markov Model can be defined as follows.

Definition 9.8. *A* Hidden Markov Model (HMM) *is a quintuple* $\mathcal{M} = (\Sigma, Q, q_0, \delta, \eta)$*, where*

- Σ *is an alphabet,*
- Q *is a finite set of states,*
- $q_0 \in Q$ *is the initial state,*
- δ *is a* $(|Q| \times |Q|)$*-matrix of transition probabilities, and*
- η *is a* $((|Q|-1) \times |\Sigma|)$*-matrix of emission probabilities.*

The matrix entry $\delta(p, q)$ *indicates the probability of a transition from state* p *to state* q*. Here,* $\delta(p, q_0) = 0$ *and* $\sum_{q \in Q} \delta(p, q) = 1$ *hold for all* $p \in Q$*.*

The matrix entry $\eta(q, a)$ *indicates for all states* $q \in Q - \{q_0\}$ *and all* $a \in \Sigma$ *the probability that the symbol* a *is output in state* q*. Here,* $\sum_{a \in \Sigma} \eta(q, a) = 1$ *for all* $q \in Q - \{q_0\}$*.*

A path *in* $\mathcal{M}$ *is a sequence* $\pi = q_0, q_1, \ldots, q_n$ *of states, starting with the initial state.*

We will first illustrate the work of an HMM with a simple example that is not motivated by any biological application.

Example 9.2. We consider the following experiment: For a series of dice rolls, sometimes a fair die is used, i.e., a die returning each of the numbers from $\{1, \ldots, 6\}$ with probability $\frac{1}{6}$, and sometimes an unfair die is used, returning the result 6 with probability $\frac{1}{2}$ and all other numbers with probability $\frac{1}{10}$. The die is changed after each roll with probability $\frac{1}{20}$. At the beginning, one of the two dice is chosen with probability $\frac{1}{2}$.

This experiment can be described by the HMM $\mathcal{M} = (\Sigma, Q, q_0, \delta, \eta)$, where the components are defined as follows:

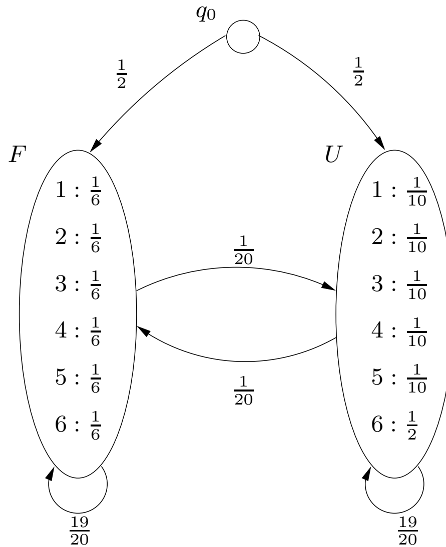


Fig. 9.2. An HMM for the dice experiment from Example 9.2

- $\Sigma = \{1, 2, 3, 4, 5, 6\}$,
- $Q = \{q_0, F, U\}$,
-

δ	q_0	F	U
q_0	0	$\frac{1}{2}$	$\frac{1}{2}$
F	0	$\frac{19}{20}$	$\frac{1}{20}$
U	0	$\frac{1}{20}$	$\frac{19}{20}$

-

η	1	2	3	4	5	6
F	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$
U	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{10}$	$\frac{1}{2}$

This HMM is shown graphically in Figure 9.2; the transitions are labeled with the transition probabilities and in the states the emission probabilities for all possible outputs are given. The state F corresponds to rolling the fair die and the state U corresponds to rolling the unfair die. $\Diamond$

For a given HMM $\mathcal{M}$, a path π in $\mathcal{M}$, and a string x, we are now able to compute the probability that the path π is followed and that the string x is output on this path.

Lemma 9.3. *Let $\mathcal{M} = (\Sigma, Q, q_0, \delta, \eta)$ be an HMM, $\pi = q_0, q_1, \ldots, q_n$ be a path in $\mathcal{M}$, and $x = x_1 \ldots x_n \in \Sigma^n$ be a string. Then, the probability that the HMM follows the path π and outputs the string x can be computed as*

$$\text{Prob}[x \wedge \pi] = \prod_{i=1}^{n} (\delta(q_{i-1}, q_i) \cdot \eta(q_i, x_i)).$$

Proof. The probability that the HMM follows the path π equals the product of the transition probabilities, i.e.,

$$\text{Prob}[\pi] = \prod_{i=1}^{n} \delta(q_{i-1}, q_i).$$

For the path π, the conditional probability that x is output, if the HMM follows π, is

$$\text{Prob}[x \mid \pi] = \prod_{i=1}^{n} \eta(q_i, x_i).$$

Thus, the probability $\text{Prob}[x \wedge \pi]$ of the HMM following π and outputting x is given by the product of the two probabilities. □

Example 9.3. In the dice experiment from Example 9.2, the probability of rolling a 6 within three consecutive rolls with the unfair die chosen at the beginning and replaced by the fair die after two rolls corresponds to the probability of following the path $\pi = q_0, U, U, F$ in the corresponding HMM and outputting the string $x = 666$. This probability can be computed using Lemma 9.3 as

$$\begin{aligned}
\text{Prob}[x \wedge \pi] &= \prod_{i=1}^{3} (\delta(q_{i-1}, q_i) \cdot \eta(q_i, x_i)) \\
&= \delta(q_0, U) \cdot \eta(U, 6) \cdot \delta(U, U) \cdot \eta(U, 6) \cdot \delta(U, F) \cdot \eta(F, 6) \\
&= \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{19}{20} \cdot \frac{1}{2} \cdot \frac{1}{20} \cdot \frac{1}{6} \\
&= \frac{19}{19200} \approx 0,00099.
\end{aligned}$$

◊

We have seen above how to compute the emission probability of a string on a given path in an HMM. But the applications often raise another question. For example, in the dice experiment from Example 9.2, we would like to know, if we can determine, from a given series of die rolls, for which rolls the fair die was used and for which rolls the unfair die was used. This corresponds to computing, for a given HMM $\mathcal{M}$ and a given string x, the path π in $\mathcal{M}$ that generated the string x.[3] Since this path π generally cannot be determined exactly, we are looking for the most probable path. We can formally describe this problem as follows.

[3] This task also gives us an explanation for the name *Hidden Markov Model*. The notion *Markov Model* is used for random processes where the current state depends on the immediate predecessor state only, and the word *Hidden* points out that the current state is normally unknown (or hidden) in typical applications.

Definition 9.9. *The* HMM decoding problem *is the following optimization problem:*

Input: An HMM $\mathcal{M} = (\Sigma, Q, q_0, \delta, \eta)$ *and a string* $x = x_1 \dots x_n \in \Sigma^n$.
Feasible solutions: All paths $\pi = q_0, \dots, q_n$ *of length* n *in the HMM* $\mathcal{M}$.
Costs: For a feasible solution π, *the costs are given by the probability that* $\mathcal{M}$ *outputs the string* x *on the path* π, *i.e.,*

$$cost(\pi) = \text{Prob}[x \mid \pi].$$

Optimization goal: Maximization.

Fortunately, the HMM decoding problem is solvable in polynomial time. We can use a dynamic programming approach in the following way: We compute, for all states of the HMM and for all prefixes of the given string, the most probable path ending in this state and emitting this prefix of the string. The following lemma shows how to compute these most probable paths.

Lemma 9.4. *Let* $\mathcal{M} = (\Sigma, Q, q_0, \delta, \eta)$ *be an HMM and let* $x = x_1 \dots x_n \in \Sigma^n$ *be a string. Let* $\sigma_q(i)$ *be the probability of the most probable path ending in the state* q *for the prefix* $x_1 \dots x_i$ *for all* $q \in Q$ *and for all* $0 \leqslant i \leqslant n$. *Then,*

$$\begin{aligned} \sigma_{q_0}(0) &= 1 \quad \textit{and} \\ \sigma_q(0) &= 0 \quad \textit{for all } q \in Q - \{q_0\} \end{aligned} \tag{9.2}$$

holds, and, for all $q \in Q - \{q_0\}$ *and for all* $1 \leqslant i \leqslant n$,

$$\sigma_q(i) = \eta(q, x_i) \cdot \max_{p \in Q}(\sigma_p(i-1) \cdot \delta(p, q)). \tag{9.3}$$

Proof. The first state in any path is always q_0; thus, the HMM is always in the state q_0 after reading the empty prefix of x. Hence, $\sigma_{q_0}(0) = 1$ and $\sigma_q(0) = 0$ hold for all $q \in Q - \{q_0\}$. This proves Equation (9.2).

The path with maximum emission probability for the prefix $x_1 \dots x_i$ ending in state q can be computed, for all $1 \leqslant i \leqslant n$ and for all $q \in Q - \{q_0\}$, as follows: Let $\pi = q_0, q_1, \dots, q_i$ be the path with the highest emission probability for $x_1 \dots x_i$ ending in state $q = q_i$. Then the probability $\sigma_{q_i}(i)$ of this path can be calculated as

$$\sigma_{q_i}(i) = \sigma_{q_{i-1}}(i-1) \cdot \delta(q_{i-1}, q_i) \cdot \eta(q_i, x_i).$$

Since π is the path with highest emission probability for $x_1 \dots x_i$ ending in q, no path containing a second to last state other than q_{i-1} can have a higher emission probability; hence,

$$\sigma_q(i) = \eta(q_i, x_i) \cdot \max_{p \in Q}(\sigma_p(i-1) \cdot \delta(p, q_i)).$$

This proves Equation (9.3). □

Algorithm 9.6 Viterbi algorithm

Input: An HMM $\mathcal{M} = (\Sigma, Q, q_0, \delta, \eta)$ and a string $x = x_1 \dots x_n \in \Sigma^n$.

1. Initialization:
   $\sigma_{q_0}(0) := 1$
   for all $q \in Q - \{q_0\}$ **do**
     $\sigma_q(0) := 0$
2. Computing $\sigma_q(i)$:
   for $i = 1$ **to** n **do**
     for all $q \in Q - \{q_0\}$ **do**
       $\sigma_q(i) := \eta(q, x_i) \cdot \max_{p \in Q}(\sigma_p(i-1) \cdot \delta(p, q))$
       $ptr_q(i) := \text{argmax}_{p \in Q}(\sigma_p(i-1) \cdot \delta(p, q))$
3. Traceback for computing a most probable path π^*:
   $\text{Prob}[x \mid \pi^*] := \max_{p \in Q - \{q_0\}}(\sigma_p(n))$
   $\pi_n^* := \text{argmax}_{p \in Q - \{q_0\}}(\sigma_p(n))$
   for $i := n - 1$ **downto** 0 **do**
     $\pi_i^* := ptr_{\pi_{i+1}^*}(i + 1)$

Output: $\pi^* = \pi_0^*, \dots, \pi_n^*$ as a most probable path for x in $\mathcal{M}$ with the probability $\text{Prob}[x \mid \pi^*]$.

Algorithm 9.6 uses Equations (9.2) and (9.3) from Lemma 9.4 to solve the HMM decoding problem. This algorithm is called *Viterbi algorithm*, after its inventor. Using Equations (9.2) and (9.3), it computes the probability of a most probable path for a given string $x = x_1 \dots x_n$ in a given HMM $\mathcal{M}$. This probability can in general be reached on several different paths. To compute one of the most probable paths, the Viterbi algorithm uses a traceback approach similar to the one we used for computing optimal pairwise alignments in Section 5.1.

Theorem 9.7. *The Viterbi algorithm solves the HMM decoding problem for an HMM with k states and a string of length n in $O(n \cdot k^2)$ time.*

Proof. Let $\mathcal{M} = (\Sigma, Q, q_0, \delta, \eta)$ be an HMM and let $x = x_1 \dots x_n \in \Sigma^n$. We first show that the Viterbi algorithm solves the HMM decoding problem. According to Lemma 9.4, the algorithm computes the correct values of $\sigma_q(i)$ for all $q \in Q$ and for all $0 \leqslant i \leqslant n$. For all $1 \leqslant i \leqslant n$ and for all $q \in Q - \{q_0\}$, the algorithm saves in the variable $ptr_q(i)$ the second to last state on a most probable path for the prefix $x_1 \dots x_i$ ending in state q.[4] Using these values, the algorithm can trace back the most probable path for x in step 3.

Now, we analyze the time complexity of the Viterbi algorithm. The initialization in step 1 clearly is possible in $O(k)$ time. In step 2, the algorithm

[4] This means that we interpret argmax in such a way that an arbitrary argument delivering the maximum value is chosen.

has to execute two nested for loops which cause the inner block of statements to be executed $O(n \cdot k)$ times. In each of these executions, the maximum over $O(k)$ values has to be calculated; this results in an overall time in $O(n \cdot k^2)$ for step 2. The traceback in step 3 then needs time in $O(n)$, and hence the total time complexity of the Viterbi algorithm is in $O(n \cdot k^2)$. □

Note that, in practice, often all calculations are done using the logarithms of the probabilities. This has two advantages: On the one hand, the emission probabilities get very small for longer strings, as we have already seen in Example 9.3, which can lead to significant rounding errors in the calculations. On the other hand, the recursion formula (9.3) can be replaced by

$$\sigma_q(i) = \log \eta(q, x_i) + \max_{p \in Q}(\sigma_p(i-1) + \log \delta(p, q)) \tag{9.4}$$

when using the logarithms. This replaces a multiplication operation by a less expensive addition operation.

In the following we show how to use the HMM method and the Viterbi algorithm to solve the problem of the CG-islands. We note that the problem of determining the CG-islands in a DNA sequence is very similar to computing a solution to the dice experiment from Example 9.2. In the dice experiment, we are given a sequence of outcomes of die rolls and we want to decide which results were obtained using the fair die and which were obtained using the unfair one. In the CG-island problem, we want to decide which regions of a given DNA sequence belong to a CG-island and which do not. In both cases, the regions in the sequence we want to separate from each other differ by the emission probabilities. But while we are looking for the frequent occurrence of a single symbol, the 6, in the dice example, we are looking for the frequent occurrence of a substring of length 2 in the DNA sequence. Thus, the HMM for the CG-island problem needs a larger number of states.

The CG-island problem can be modeled by the HMM $\mathcal{M}_{\texttt{CG}} = (\Sigma_{\text{DNA}}, Q, q_0, \delta, \eta)$, where the components are defined as follows:

- $\Sigma_{\text{DNA}} = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$.
- $Q = \{q_0, A^+, C^+, G^+, T^+, A^-, C^-, G^-, T^-\}$. Here the states marked $+$ model CG-islands, and the states marked $-$ model the rest of the DNA sequence.
- The transition probabilities can be estimated from testing data, i.e., by counting DNA sequences with already known CG-islands. We know that the probability of going into a state marked $+$ or of staying inside states marked $+$ is significantly lower than for states marked $-$. This models the fact that the CG-islands only make up a small part of the entire DNA sequence. The special property of CG-islands is modeled by

$$\delta(C^+, G^+) \gg \delta(C^-, G^-).$$

- For the emission probabilities, we have

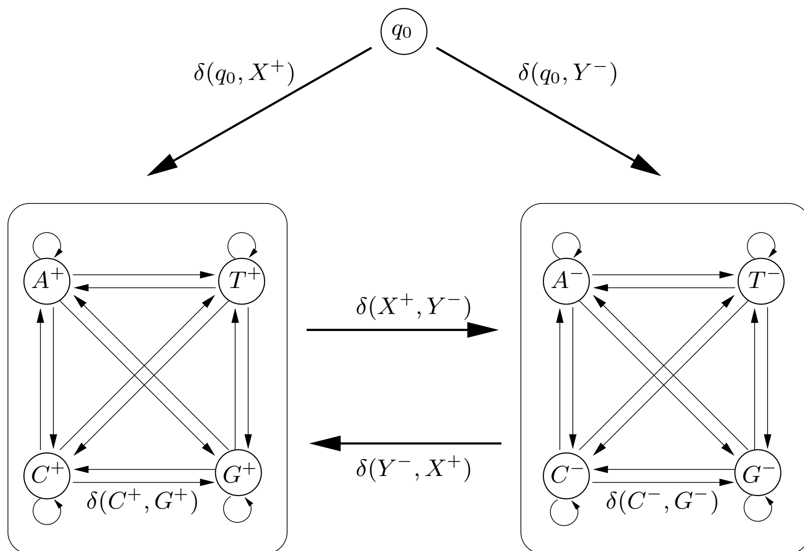


Fig. 9.3. The HMM $\mathcal{M}_{\texttt{CG}}$ for determining the CG-islands in a DNA sequence. The states of $\mathcal{M}_{\texttt{CG}}$ (excluding the initial state) are divided into two classes: The states marked with + represent the CG-islands, the states marked with − represent the remainder of the DNA sequence. The labeling of the transitions with the corresponding transition probabilities are only shown exemplarily in the figure for the transitions from C^+ to G^+ and from C^- to G^-. The transitions between the two classes and from the initial state are only shown schematically; X and Y stand for all symbols in $\{A, C, G, T\}$. Moreover, the emission probabilities are omitted since the emitted symbol is unambiguously determined in every state of the HMM

$$\begin{aligned}
\eta(A^+, \texttt{A}) = \eta(A^-, \texttt{A}) = 1, \\
\eta(C^+, \texttt{C}) = \eta(C^-, \texttt{C}) = 1, \\
\eta(G^+, \texttt{G}) = \eta(G^-, \texttt{G}) = 1, \\
\eta(T^+, \texttt{T}) = \eta(T^-, \texttt{T}) = 1.
\end{aligned}$$

This means that in the states X^+ and X^- only the symbol X can be emitted for all $X \in \{A, C, G, T\}$.

The HMM $\mathcal{M}_{\texttt{CG}}$ is shown schematically in Figure 9.3. Using the Viterbi algorithm we can now determine a most probable path π in $\mathcal{M}_{\texttt{CG}}$ for any given DNA sequence. The path π then gives us a hypothesis where the CG-islands are located within the sequence, namely, at the positions where the symbols of the sequence were emitted from a state marked +.

9.5 Summary

After having sequenced a DNA, the next important problem is to determine biologically interesting regions within the DNA sequence, called signals. There are many approaches for addressing and, at least partially, solving this problem. Some of these approaches use combinatorial techniques; many are based on statistical methods.

One of the combinatorial approaches is the computation of identical or similar substrings in several DNA sequences. This leads to the consensus string problem which consists of finding, for a given set of strings and a given length, a median string of this length that has a minimum overall Hamming distance to some substring from each of the given strings. This problem is NP-hard but well approximable.

Also, computing tandem repeats, i.e., consecutive identical substrings, in a DNA sequence can be used for finding signals. This problem is efficiently solvable using suffix trees.

One of the statistical approaches for signal finding is to determine all substrings in the DNA sequence that occur significantly more frequently or significantly less frequently than one would expect in a random sequence of the same length. The difficulty here lies in judging the significance of the frequency, since the variance for the number of expected occurrences of a substring depends on the structure of its self-overlaps. This difficulty can be overcome by determining the autocorrelation polynomial of the substring.

Another statistical tool that has many applications in molecular biology are Hidden Markov Models, which one can think of as probabilistic finite automata emitting one symbol at each computational step. Using Hidden Markov Models, one can for example determine the `CG`-islands within a DNA sequence. These are regions in the DNA where the normally rare dinucleotide `CG` occurs very frequently; they point to coding regions in many genomes.

9.6 Bibliographic Notes

An overview of the different approaches for signal finding in DNA sequences is given in the books by Pevzner [159], Ewens and Grant [68], Durbin et al. [62], and Clote and Backofen [49]. An introduction to the needed background in statistics and probability theory can also be found in the book by Ross [169].

The book by Pevzner [159] gives a good overview over the different methods for determining identical or similar substrings. The proof of NP-completeness of the consensus string problem goes back to Li et al. [135]. The approximation algorithm presented for the consensus string problem was also designed and analyzed by Li et al. [134, 135].

The biological importance of tandem repeats was described by Benson [26]. Our presentation of the computation of all tandem repeats leans on the book by Gusfield [91]. The efficient computation of the lowest common ancestor in

a suffix tree was first described by Harel and Tarjan [100]; a simplified version is due to Schieber und Vishkin [175]; this method is also described in detail in the book by Gusfield [91]. Main and Lorentz [142] have introduced the first algorithm for determining all tandem repeats; an extension to tandem repeats with a constant number of mismatches is due to Landau und Schmidt [126]. Crochemore [53] designed an algorithm for efficiently computing all tandem repeats with more than two copies.

The papers by van Helden et al. [194], Leung et al. [131], and Karlin et al. [114] point out examples for the biological importance of significantly more frequent or significantly less frequent substrings. An overview of the different approaches and statistical models for determining frequent or infrequent substrings is given in the books by Ewens and Grant [68] and Pevzner [159] and in the survey paper by Reinert et al. [165]. The model we presented here was investigated by Pevzner et al. [160] and by Gentleman and Mullin [82], the definition of the autocorrelation polynomial goes back to Guibas und Odlyzko [88]. A more recent, rather combinatorial approach is due to Apostolico et al. [14].

The Hidden Markov Models were introduced by Baum and Petrie [23]; an overview can be found in the paper by Rabiner [163] or in the book by Durbin et al. [62]. The application of Hidden Markov Models to the `CG`-island problem was initially proposed by Churchill [44]. The dice example is taken from the book by Durbin et al. [62]. The Viterbi algorithm was designed by Viterbi [198]. There are many more applications of Hidden Markov Model in molecular biology, for an overview we refer the reader to the books by Ewens and Grant [68], Durbin et al. [62], and Baldi and Brunak [21].

10

Genome Rearrangements

In Chapter 5 we saw how to compare DNA sequences to each other. As a measure for this comparison we used the number of local mutations needed to transform one sequence into the other. Besides the local mutations, i.e., deleting, inserting, and substituting a single nucleotide, another kind of mutation also occurs in nature, changing the DNA sequence at a higher level. These mutations cut out larger parts of a chromosome and put it back into the sequence at another position or in reverse orientation.

We may assume that mutations of this type normally cut the DNA sequences between the genes only, since cutting a gene usually leads to its destruction, and this will normally decrease the fitness of the organism or may even have a lethal effect. This means that the genes themselves are not altered by these mutations, but only their order changes. We call this kind of mutations *genome rearrangements* in the following.

We can now use the number of such genome rearrangements as a measure of the distance between entire genomes. We discuss some of the resulting models here. The chapter is organized as follows. In Section 10.1 we introduce different models of genome rearrangements. We then look at two different models in greater detail in Sections 10.2 and 10.3. These models differ in whether or not we also distinguish, in addition to the order of the genes, their reading direction. Section 10.4 is devoted to a question related to genome rearrangements. We compare organisms whose genomes are distributed over several chromosomes, and where we do not know the exact order of the genes but know only their assignment to the chromosomes. The chapter closes with a summary in Section 10.5 and bibliographic remarks in Section 10.6.

10.1 Modeling

In closely related species, the DNA sequences coding for single genes often are nearly identical. If one wants to build a phylogenetic tree of such closely

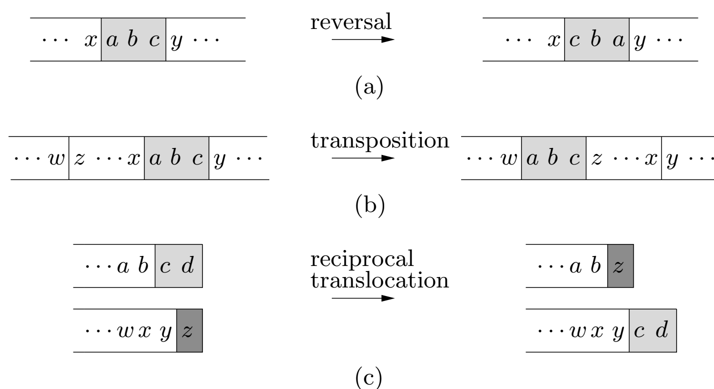


Fig. 10.1. Different types of genome rearrangements

related species,[1] a comparison of the sequences of single genes does not help much, since the differences are too small. On the other hand, often the *order* of the genes varies a lot. Thus, it looks promising to compare the entire genomes with each other and to use the number of genome rearrangements needed to transform one genome into another as a measure of evolutionary distance.

We distinguish between different types of transformations that can alter the order of the genes. First, we distinguish between *intrachromosomal* transformations, changing the order of the genes within one chromosome, and *interchromosomal* transformations, exchanging parts of gene sequences between different chromosomes. Figure 10.1 schematically shows some possible transformations.

Figures 10.1 (a) and (b) show two types of intrachromosomal transformations. A *reversal* (also called *inversion*), as shown in Figure 10.1 (a), cuts out a part of the gene sequence and puts it back in reverse orientation. A *transposition*, as shown in Figure 10.1 (b), also cuts out a part of the gene sequence, but it puts it back in the same orientation at another position.

In Figure 10.1 (c), one possible type of interchromosomal transformation is shown, the *reciprocal translocation* which interchanges the ends of two chromosomes.

In a special case, occurring, for example, in mitochondrial genomes or in the genomes of chloroplasts,[2] reversals occur as essentially the only form of genome rearrangements. We study this case in greater detail in the following. To determine the distance between two genomes, we try to find the minimum

[1] We present some methods for building phylogenetic trees in detail in Chapter 11.

[2] Mitochondria and chloroplasts are cell organelles carrying their own genomes. These genomes are especially well studied since they are on the one hand rather small and on the other hand easier to extract from a cell for further investigation, due to the fact that there are usually several hundreds of copies of these genomes within a single cell.

number of reversals transforming one genome into another. We first investigate the special case where each genome consists of exactly one chromosome. We further assume that there are no multiple occurrences of the same gene within a genome. We can hence describe the order of the genes by a permutation, identifying one element of the permutation with each particular gene such that the same element is assigned to homologous genes[3] in both organisms. A reversal of a part of the gene sequence corresponds to a reversal of the permutation. Thus, we can now formalize our task as the problem of transforming a given permutation into another given permutation with the minimum number of reversals. This is equivalent to the problem of sorting a permutation by reversals, since we can assume without loss of generality that the target permutation is the identity permutation.

In this case there are still two possible models depending on whether or not we also know the reading direction of the genes or.[4] If the reading direction is unknown, the resulting problem is to sort a normal permutation by reversals. We will deal with this problem in Section 10.2. But if the reading direction is known, this leads to the problem of sorting a *directed permutation* by reversals.[5] A directed permutation is a permutation where each element gets assigned one of two directions. We formally define this problem in Section 10.3. As an illustrative example, Figure 10.2 shows a series of reversals transforming the order of genes in the mitochondrial genome of cabbage (*Brassica oleracea*) into the order of the respective homologous genes in the mitochondrial genome of turnip (*Brassica campestris*). Here, the minus signs indicate a reading direction from right to left.

10.2 Sorting Undirected Permutations

In this section we deal with the problem of sorting a normal undirected permutation with a minimum number of reversals. We start with the formal definition of a reversal.

Definition 10.1. *Let $\pi = (\pi_1, \ldots, \pi_n)$ be a permutation of order n. For $1 \leqslant i < j \leqslant n$, an (i,j)-*reversal *is a permutation $\rho(i,j)$, such that*

$$\pi \cdot \rho(i,j) = (\pi_1, \ldots, \pi_{i-1}, \pi_j, \pi_{j-1}, \ldots, \pi_{i+1}, \pi_i, \pi_{j+1}, \ldots, \pi_n).$$

Moreover, we define an (i,i)-reversal to be the identity permutation, and, for $1 \leqslant j < i \leqslant n$, let the (i,j)-reversal be the same as the (j,i)-reversal.

---

[3] Two genes in different but related organisms are called *homologous* if they have the same ancestor gene in evolutionary history.

[4] A different reading direction of two genes can occur if a subset of the genes is transcribed from one strand of the DNA and the remainder of the genes are transcribed from the other strand.

[5] In the literature, directed permutation are often also called *signed permutations*, the problem is accordingly described as sorting a signed permutation by reversals.

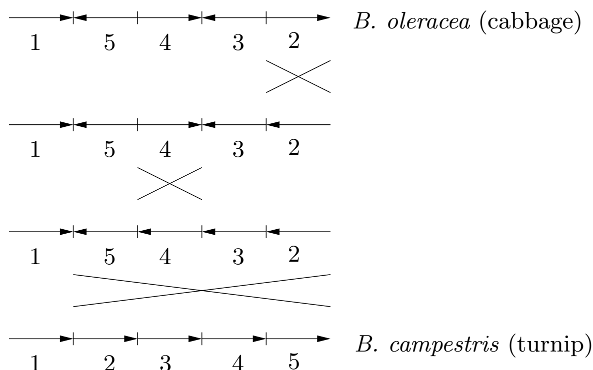


Fig. 10.2. An example for sorting a directed permutation by reversals: transforming the gene order in the mitochondrial genome of cabbage into the gene order in the mitochondrial genome of turnip

Using this notation, we can now define the problem of sorting a permutation by reversals as follows.

Definition 10.2. *The problem of sorting an (undirected) permutation by reversals, the* MinSR *problem, is the following optimization problem:*

Input: A natural number n and a permutation $\pi = (\pi_1, \ldots, \pi_n)$ of order n.
Feasible solutions: Every sequence $\rho_1, \ldots, \rho_t$ of reversals of order n, such that the identity permutation can be described as the consecutive application of the permutations $\rho_1, \ldots, \rho_t$ to π, i.e., $\pi\rho_1 \cdots \rho_t = (1, \ldots, n)$.
Costs: For each feasible solution $\rho_1, \ldots, \rho_t$, the cost is the number t of reversals.
Optimization goal: Minimization.

In the definition of the MinSR problem, we have assumed that the second permutation, i.e., the one we want to transform the first permutation into, is the identity permutation. Note that this is no real restriction, we can always attain this by renumbering the genes appropriately. We illustrate the above definitions with an example.

Example 10.1. Consider the permutations $\pi = (2, 1, 3, 7, 5, 4, 8, 6)$ and $\sigma = (1, 2, 3, 4, 5, 6, 7, 8)$. Then, π can be transformed into σ by the reversals shown in Figure 10.3. $\Diamond$

The question now is whether the MinSR problem is efficiently solvable. Unfortunately, the answer is negative, as shown by the following theorem.

Theorem 10.1. *The* MinSR *problem is NP-hard.* $\square$

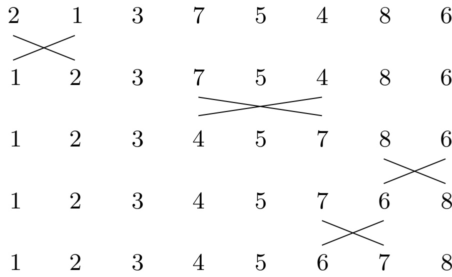


Fig. 10.3. Reversal sequence sorting permutation $\pi = (2, 1, 3, 7, 5, 4, 8, 6)$ into $\sigma = (1, 2, 3, 4, 5, 6, 7, 8)$

Theorem 10.1 was proved by Caprara [40]. But the reduction used in the proof is technically quite involved; we thus skip the proof here.

Since there is no hope for an exact polynomial-time solution to the problem, we present an approximation algorithm achieving an approximation ratio of 2. For this algorithm, we need the following definitions.

Definition 10.3. *Let* $\pi = (\pi_1, \ldots, \pi_n)$ *be a permutation of order* n*. We define* $\pi_0 = 0$ *and* $\pi_{n+1} = n + 1$*. Then, we call* $ext(\pi) = (\pi_0, \pi_1, \ldots, \pi_n, \pi_{n+1})$ *the* extended representation *of* π*.*

In the remainder of this chapter, we no longer make a sharp distinction between a permutation π and its extended representation $ext(\pi)$; we use the notion $ext(\pi)$ only occasionally for clarification.

Definition 10.4. *Let* $\pi = (\pi_1, \ldots, \pi_n)$ *be a permutation of order* n*. A* breakpoint *of* π *is a pair* $(i, i + 1) \in \{0, \ldots, n\} \times \{1, \ldots, n + 1\}$ *of positions such that* $|\pi_i - \pi_{i+1}| \neq 1$ *holds. By* $brp(\pi)$ *we denote the number of breakpoints of* π*.*

The following example illustrates this definition.

Example 10.2. Consider the permutation $\pi = (4, 3, 2, 7, 1, 5, 6, 8)$ of order 8. Here, $ext(\pi) = (0, 4, 3, 2, 7, 1, 5, 6, 8, 9)$. In the following representation, a breakpoint $(i, i + 1)$ of π is depicted by a vertical line between π_i and π_{i+1}:

$$0 | 4\ 3\ 2 | 7 | 1 | 5\ 6 | 8\ 9$$

$\Diamond$

Note that the identity permutation is the only permutation without breakpoints, since the elements 0 and $n + 1$ of $ext(\pi)$ cannot be moved away from their positions by any reversal.

The notion of breakpoints now enables us to determine a lower bound for the number of reversals necessary to transform a given permutation π into the identity permutation.

Lemma 10.1. *Let π be a permutation of order n. Then, at least $\lceil brp(\pi)/2 \rceil$ reversals are necessary to transform π into the identity permutation of order n.*

Proof. Since the identity permutation is the only permutation without breakpoints, transforming a given permutation into the identity permutation is equivalent to eliminating all breakpoints. An (i, j)-reversal can only affect the breakpoints $(i-1, i)$ and $(j, j+1)$; thus, each reversal can decrease the number of breakpoints by at most 2. □

We show that it is possible to find a sequence of reversals sorting an arbitrary given permutation and eliminating *on average* one breakpoint with each reversal. For the proof we need another definition.

Definition 10.5. *Let π be a permutation of order n. Let $k = brp(\pi)$, and let $(i_1, i_1 + 1), \ldots, (i_k, i_k + 1)$, for $i_1 < \cdots < i_k$, be the breakpoints of π. Then we call the $k + 1$ sequences $s_0 = (\pi_0, \ldots, \pi_{i_1}), s_1 = (\pi_{i_1+1}, \ldots, \pi_{i_2}), \ldots, s_k = (\pi_{i_k+1}, \ldots, \pi_{n+1})$ the* strips *of π.*

The strip s_j is called ascending *if $\pi_{i_j+1} < \ldots < \pi_{i_{j+1}}$, and* descending *if $\pi_{i_j+1} > \ldots > \pi_{i_{j+1}}$, for all $0 \leqslant j \leqslant k$. A strip s_j of length 1, i.e., a strip consisting of only one element of the permutation, is also called descending if $1 \leqslant j \leqslant k - 1$. But if the strips s_0 or s_k consist of one element only, they are called ascending.*

Informally speaking, the strips of a given permutation are the maximal ascending or descending sequences of elements, separated by breakpoints. The seemingly arbitrary convention about which strips of length 1 are called ascending or descending will be motivated later by our algorithm. First, we illustrate Definition 10.5 with the permutation from our above example.

Example 10.3. The permutation $\pi = (4, 3, 2, 7, 1, 5, 6, 8)$ has five breakpoints, as we saw in Example 10.2. The breakpoints separate $ext(\pi)$ into the strips $s_0 = (0)$, $s_1 = (4, 3, 2)$, $s_2 = (7)$, $s_3 = (1)$, $s_4 = (5, 6)$, and $s_5 = (8, 9)$. The strips s_0, s_4, and s_5 are ascending; the strips s_1, s_2, and s_3 are descending. ◊

Note that the identity permutation can also be characterized as the only permutation consisting only of a single ascending strip. As we will see, the descending strips play an important role for eliminating the breakpoints of a given permutation.

Lemma 10.2. *Let π be a permutation of order n. Let $k \in \{0, \ldots, n + 1\}$ be an element of $ext(\pi)$.*

(a) If k lies in a descending strip of $ext(\pi)$, and $k - 1$ lies in an ascending strip of $ext(\pi)$, then there exists a reversal ρ such that $brp(\pi\rho) < brp(\pi)$.
(b) If l lies in a descending strip of $ext(\pi)$, and $l + 1$ lies in an ascending strip of $ext(\pi)$, then there exists a reversal σ such that $brp(\pi\sigma) < brp(\pi)$.

Proof. (a) Let k lie in a decreasing strip s and let $k-1$ lie in an ascending strip s'. Then k as well as $k-1$ have to be the last element of their respective strips, and thus are part of a breakpoint. We distinguish two cases with respect to the order of the two strips s und s'. A reversal ρ eliminating one breakpoint is shown for the case where s precedes s' in $ext(\pi)$ in Figure 10.4 (a), and for the opposite case in Figure 10.4 (b).

(b) The proof is analogous to the proof of (a). Let l lie in a descending strip s, and let $l+1$ lie in an ascending strip s'. We again distinguish two cases according to the order of s and s'. A reversal σ removing one breakpoint is shown in Figure 10.4 (c) for the case where s precedes s' in $ext(\pi)$, and in Figure 10.4 (d) for the opposite case. □

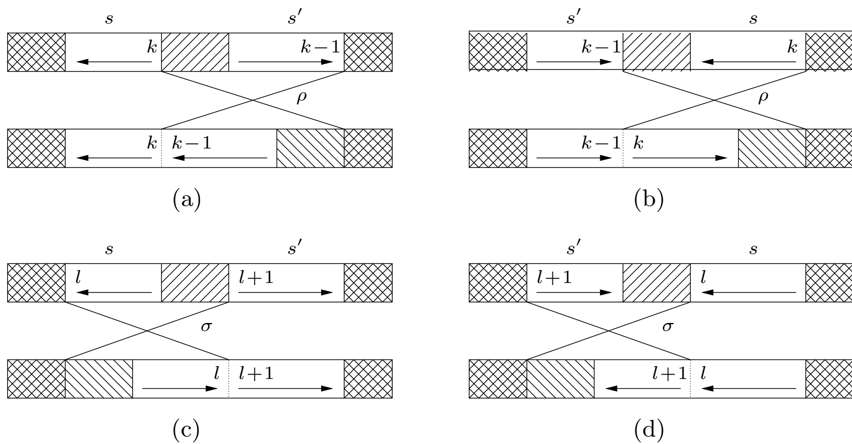


Fig. 10.4. Eliminating one breakpoint in the proof of Lemma 10.2. In (a) and (b) the two cases for the proof of Lemma 10.2 (a) are shown; (c) and (d) illustrate the two cases for the proof of Lemma 10.2 (b)

From Lemma 10.2 we can conclude that the existence of a descending strip always ensures that we can find a reversal eliminating at least one breakpoint.

Lemma 10.3. *Let π be a permutation with a descending strip. Then, there exists a reversal ρ such that $brp(\pi\rho) < brp(\pi)$.*

Proof. We choose k as the smallest element of π occurring in a descending strip. Then the element $k-1$ has to lie in an ascending strip; this also holds for the case $k=1$, since we have defined the strip $(\pi_0) = (0)$ to be ascending. We can then simply apply Lemma 10.2 (a). □

To reach our goal of finding a series of reversals eliminating one breakpoint per reversal on average, we now have to consider permutations consisting of ascending strips only. We show that this situation cannot occur too often and that it is always preceded by a reversal eliminating two breakpoints.

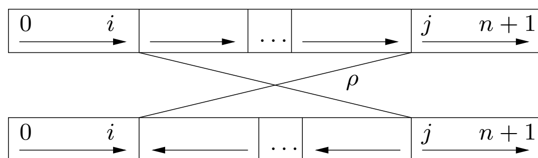


Fig. 10.5. The situation in the proof of Lemma 10.4

Lemma 10.4. *Let π be a permutation without descending strips. Then π is either the identity permutation or there exists a reversal ρ such that $\pi\rho$ contains a descending strip and $brp(\pi\rho) \leqslant brp(\pi)$ holds.*

Proof. Let π be different from the identity permutation. Then π contains at least two breakpoints, as the following consideration shows: We know that $\pi_0 = 0$ as well as $\pi_{n+1} = n + 1$ lie in ascending strips. If 0 and $n + 1$ would lie within the same strip, then π would be the identity permutation. Thus, there exist at least two ascending strips $s_0 = (\pi_0, \ldots, \pi_i) = (0, \ldots, i)$ and $s_1 = (\pi_j, \ldots, \pi_{n+1}) = (j, \ldots, n + 1)$ in π with $j > i + 1$. Hence, at least the breakpoints $(i, i + 1)$ and $(j - 1, j)$ exist in π.

The reversal ρ, reversing the part of π between the two breakpoints, turns the strips in between into descending strips and does not create a new breakpoint. This reversal is shown in Figure 10.5. □

We next show that there always exists a strategy such that a permutation without descending strips is only reached by a reversal eliminating two breakpoints (with the exception of a starting permutation without ascending strips).

Lemma 10.5. *Let π be a permutation with a descending strip. Let k be the smallest element in a descending strip of π and let l be the largest element in a descending strip of π. Let ρ be the reversal placing $k - 1$ next to k, and let σ be the reversal placing $l + 1$ next to l. If $\pi\rho$ as well as $\pi\sigma$ do not contain any descending strip, then $\rho = \sigma$ and $brp(\pi\rho) = brp(\pi) - 2$.*

Proof. Since k is the smallest element within a descending strip, $k - 1$ has to lie within an ascending strip. According to Lemma 10.2 (a), this results in one of the two situations shown in Figures 10.4 (a) and (b). If $k - 1$ lies to the right of k, then the reversal preserves the descending strip containing k (see Figure 10.4 (a)). We thus can assume that $k - 1$ lies to the left of k in π (see Figure 10.4 (b)).

Since l is the largest element in a descending strip, $l + 1$ has to lie within an ascending strip. According to Lemma 10.2 (b), this results in one of the two situations shown in Figures 10.4 (c) or (d). If $l + 1$ lies to the left of l, then this reversal preserves the descending strip containing l (see Figure 10.4 (d)). We thus can assume that $l + 1$ occurs to the right of l in π (see Figure 10.4 (c)).

Algorithm 10.1 Approximation algorithm for the MinSR problem

Input: A permutation π of order n.

1. Set *list* $:= \emptyset$.
2. **while** π is not the identity permutation **do**
   - **if** π has a descending strip **then**
     - Compute the smallest element k inside a descending strip of π.
     - Compute the position i of k in π and the position i' of $k-1$ in π.
     - **if** $i < i'$ **then**
       - $\rho := (i+1, i')$-reversal of $ext(\pi)$
     - **else** $\{i > i'\}$
       - $\rho := (i'+1, i)$-reversal of $ext(\pi)$
     - **if** $\pi\rho$ has no descending strip **then**
       - Compute the largest element l in a descending strip of π.
       - Compute the position j of l in π and the position j' of $l+1$ in π.
       - **if** $j < j'$ **then**
         - $\rho := (j, j'-1)$-reversal of $ext(\pi)$
       - **else** $\{j > j'\}$
         - $\rho := (j', j-1)$-reversal of $ext(\pi)$
   - **else** $\{\pi$ has no descending strip$\}$
     - $\rho :=$ reversal cutting at the first two breakpoints of $ext(\pi)$
   - $\pi := \pi\rho$
   - *list* $:=$ *list* $\cup \rho$

Output: The list *list* of reversals.

These considerations imply that k has to lie within the part of the permutation reversed by σ, since otherwise the descending strip containing k in π would be preserved in $\pi\sigma$. Analogously, l has to lie inside the interval reversed by ρ. We now show that this implies $\rho = \sigma$.

If we assume $\rho \neq \sigma$, then there exists a strip belonging to the interval of only one of the two reversals, and, without loss of generality, only to the interval of ρ. If this strip is ascending in π, it is descending in $\pi\rho$; if it is descending in π, it is preserved or even prolonged in $\pi\sigma$. Since neither of these cases is possible, no such strip exists, and thus $\rho = \sigma$. The reversal $\rho = \sigma$ hence eliminates two breakpoints since it brings k and $k-1$ as well as l and $l+1$ next to each other in $\pi\rho$. □

Lemmas 10.3, 10.4, and 10.5 now imply a 2-approximation algorithm for the MinSR problem (see Algorithm 10.1).

Before we start proving the approximation ratio of Algorithm 10.1, we illustrate its work with an example.

Example 10.4. We consider the permutation $\pi = (3, 4, 1, 2, 7, 8, 5, 6)$. An optimal sequence of reversals for sorting π is shown in Figure 10.6 (a), Figure 10.6 (b) shows the sequence of reversals computed by Algorithm 10.1. In Figure 10.6 (b), the ascending strips are marked with arrows from left to right,

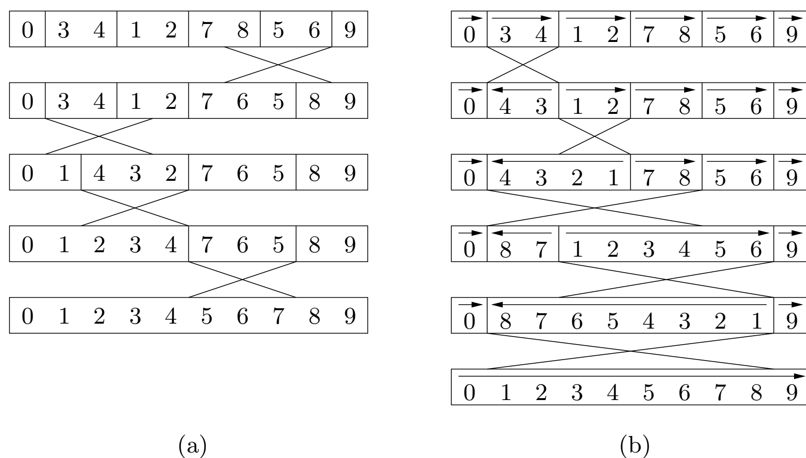


Fig. 10.6. Sorting the permutation $\pi = (3, 4, 1, 2, 7, 8, 5, 6)$. (a) An optimal reversal sequence; (b) the sequence computed by Algorithm 10.1

and the descending strips are marked with arrows from right to left. In the given permutation $ext(\pi)$, there are no descending strips; hence, the algorithm reverses the strip between the first two breakpoints. This reversal does not eliminate any breakpoint. In the next step, 3 is the smallest element in a descending strip, and the algorithm tries to place element 3 next to element 2 by a reversal. The algorithm succeeds by reversing the strip $(1, 2)$; this leads to a permutation that still has a descending strip, $(4, 3, 2, 1)$. Trying to place the smallest element 1 in this descending strip next to 0 fails, since this would generate a permutation without descending strips. The algorithm thus places the largest element 4 in the descending strip next to 5. This results in a permutation containing the descending strip $(8, 7)$. Using two further reversals, placing the smallest element in a descending strip next to its predecessor, the algorithm eventually reaches the identity permutation after five reversals.

But the optimal solution only needs four reversals, as shown in Figure 10.6 (a). Note that two of the reversals cut at positions inside a strip, while the algorithm generally cuts at breakpoints only. This shows that in general it may be advantageous to split already sorted regions. $\Diamond$

Theorem 10.2. *Algorithm 10.1 is a polynomial-time* 2*-approximation algorithm for the* MinSR *problem.*

Proof. According to Lemma 10.1, it suffices to prove that every reversal computed by the algorithm eliminates one breakpoint on average.

If the current permutation contains a descending strip, the reversal carried out by the algorithm eliminates one breakpoint, according to Lemma 10.3, if the resulting permutation also contains a descending strip. If the resulting

permutation consists of ascending strips only, the executed reversal eliminates two breakpoints, according to Lemma 10.5.

If the current permutation consists of ascending strips only, the executed reversal does not increase the number of breakpoints, according to Lemma 10.4. This situation can only occur at the very beginning of the computation or immediately after a reversal eliminating two breakpoints. Thus, every such reversal, except the one at the very beginning, can be compensated by the preceding reversal to achieve two eliminated breakpoints by two reversals. The reversal at the beginning of the computation can be compensated by the last reversal, which has to eliminate two reversals since there does not exist any permutation with exactly one breakpoint.

Overall, each reversal carried out by the algorithm eliminates at least one breakpoint on average. □

We will now analyze the time complexity of Algorithm 10.1. A permutation of order n can contain at most $n+1$ breakpoints. Since the number of breakpoints is on average reduced by at least one with each executed reversal, the algorithm needs at most $n+1$ executions of the outer while loop. The operations in each execution of the while loop, i.e., the computation of the descending strips in at most two different permutations, can obviously be done in linear time. Overall, this results in time complexity of $O(n^2)$ for sorting a permutation of order n.

There are several other approximation algorithms known for the MinSR problem. Currently, the best known algorithm was designed by Berman et al. [31] and reaches an approximation ratio of 1.375. This algorithm is based on technically very involved constructions; we will thus not present it here.

10.3 Sorting Directed Permutations

In this section we deal with the problem of sorting a directed permutation by reversals. As already described in Section 10.1, a directed permutation is one where each element additionally gets assigned one of two directions. We start with a more formal definition.

Definition 10.6. *A* directed permutation *of order n is defined as an n-tuple $\pi = (\pi_1, \ldots, \pi_n)$, where $\pi_i \in \{\overrightarrow{1}, \ldots, \overrightarrow{n}, \overleftarrow{1} \ldots, \overleftarrow{n}\}$ for $1 \leqslant i \leqslant n$, such that from each set $\{\overrightarrow{j}, \overleftarrow{j}\}$, $1 \leqslant j \leqslant n$, exactly one element occurs in π.*

Moreover, let $\overrightarrow{Perm_n}$ denote the set of all directed permutations of order n.

For example, $\overrightarrow{1}\,\overleftarrow{3}\,\overleftarrow{5}\,\overrightarrow{2}\,\overleftarrow{4}$ is a directed permutation of order 5.

We will now formally define the notion of a reversal for a directed permutation.

Definition 10.7. *Let $\pi = (\pi_1, \ldots, \pi_n)$ be a directed permutation of order n. For $1 \leqslant i \leqslant j \leqslant n$, an (i,j)*-reversal of order *n is a function $\rho(i,j) : \overrightarrow{Perm_n} \to \overrightarrow{Perm_n}$ such that*

$$\pi \cdot \rho(i,j) = (\pi_1, \ldots, \pi_{i-1}, \overline{\pi_j}, \overline{\pi_{j-1}}, \ldots, \overline{\pi_{i+1}}, \overline{\pi_i}, \pi_{j+1}, \ldots, \pi_n),$$

where $\overline{\pi_k} = \overrightarrow{x}$ if $\pi_k = \overleftarrow{x}$, and $\overline{\pi_k} = \overleftarrow{x}$ if $\pi_k = \overrightarrow{x}$, for all $1 \leqslant k \leqslant n$.

An example for the application of reversals to a directed permutation is shown in Figure 10.2 in Section 10.1.

We can now formally define the problem of sorting a directed permutation with a minimum number of reversals as follows.

Definition 10.8. *The problem of sorting a directed permutation with a minimum number of reversals, the* MinOSR *problem, is the following optimization problem:*

Input: A natural number n and two directed permutations π and σ of order n.
Feasible solutions: Every sequence $\rho_1, \ldots, \rho_t$ of reversals of order n, such that $\pi\rho_1 \ldots \rho_t = \sigma$.
Costs: For each feasible solution $\rho_1, \ldots, \rho_t$, the costs are given by the number t of reversals.
Optimization goal: Minimization.

Note that in the case of directed permutations, the choice of the permutation σ, into which π has to be transformed, really does matter. In contrast to the undirected case, transforming π into the directed identity permutation $(\overrightarrow{1}, \overrightarrow{2}, \ldots, \overrightarrow{n})$ is a proper subproblem of the MinOSR problem.

Also, we can ask whether the problem it is efficiently solvable. But in contrast to the case of the undirected permutations, the answer is surprisingly positive, as shown by the following theorem.

Theorem 10.3. *The* MinOSR *problem is optimally solvable in polynomial time.* □

Theorem 10.3 was proved in 1995 by Hannenhalli and Pevzner [96]. But the algorithm by Hannenhalli and Pevzner is technically quite involved; thus, we skip it here.

Up to now, we have always modeled a genome by a linear sequence of genes. But in many organisms the genes are distributed over several chromosomes. In this case, a genome can be modeled as an (unordered) set of linear gene sequences. The operations on such genomes are not limited to intrachromosomal ones such as reversals, but they also include *translocations* cutting off part of a chromosome and appending it to another chromosome. Computing a minimal sequence of operations from this extended operation set that transforms a given genome into another one is also possible in polynomial time as shown by Hannenhalli and Pevzner [97]. This construction uses similar techniques as the proof of Theorem 10.3 in [96], but it is technically even more involved.

10.4 Computing the Syntenic Distance

In the preceding sections we presented different methods for comparing two genomes whose sequences of genes were known to us. But in many cases we do not have such exact data. For a given genome with several chromosomes, it is easier to find out the assignment of the genes to the chromosomes than to determine the exact order of the genes. Also, this data, though less precise, can be used to estimate the distance between genomes. In this model, we consider three types of interchromosomal mutations: the division of a chromosome into two parts, called *fission*, the merging of two chromosomes into one, called *fusion*, and the so-called *translocation*, where a part is cut off from each of two chromosomes each and appended to the other chromosome. The minimum number of such operations necessary to transform a given genome into another one is called the *syntenic distance* of the genomes.[6]

For the formal definition of this model, we assume that the set of genes is the same in both considered genomes, or put another way, we only consider the genes for which we also know their homologue in another genome.

Consider two genomes $\mathcal{G}_1$ and $\mathcal{G}_2$, let $G = \{g_1, \ldots, g_m\}$ be the set of genes occurring in both genomes. Then, in our context, we can describe $\mathcal{G}_1$ and $\mathcal{G}_2$ as two partitions $\mathcal{G}_1 = \{S_1, \ldots, S_k\}$ and $\mathcal{G}_2 = \{T_1, \ldots, T_n\}$ of G, where the single sets of the partitions correspond to the chromosomes. Our task now is to transform the partition $\mathcal{G}_1$ into the partition $\mathcal{G}_2$ using the smallest possible number of operations modeling the interchromosomal mutations as described above.

Before we present a formal definition of this model, we briefly discuss its drawbacks. There are several reasons why the syntenic distance calculated within this model can differ significantly from the true evolutionary distance of the genomes. The main problem is that, in nature, genes are linearly ordered on a chromosome, and translocations occurring in nature only exchange parts of the chromosomes, while retaining the order of the genes, such as with the reciprocal translocations shown in Figure 10.1 (c). This can lead to the situation where the computed syntenic distance is smaller than the number of real transformations. Another problem with the model is that, in nature, fusions and fissions are much scarcer than reciprocal translocations, in contrast to our model where all operations are counted with the same weight. Moreover, the synteny data used in practice is often incomplete, since not all genes in the given genomes are known and not all homologies between the known genes have been determined.

In spite of all these problems, computing the syntenic distance can be a sensible model for computing the distance between genomes where the true order of the genes is unknown.

For formally describing a genome and the operations on it, we use a slightly generalized model. As a ground set (set of genes) we allow an arbitrary finite

[6] This notion goes back to the fact that two genes lying on the same chromosome are called *syntenic*.

set, and as a genome over this ground set we allow an arbitrary family of nonempty subsets of the ground set whose union equals the ground set, i.e., we allow a gene to occur on different chromosomes. This generalization of the definition will later allow for a reformulation of the problem which makes the proofs easier.

Definition 10.9. *Let* $G = \{g_1, \ldots, g_m\}$ *be a set and let* $\mathcal{G}_1 = \{S_1, \ldots, S_k\}$ *and* $\mathcal{G}_2 = \{T_1, \ldots, T_n\}$, *where* $S_i \subseteq G$, $S_i \neq \emptyset$, *for all* $1 \leqslant i \leqslant k$, *and* $\bigcup_{i=1}^{k} S_i = G$; *and* $T_j \subseteq G$, $T_j \neq \emptyset$, *for all* $1 \leqslant j \leqslant n$, *and* $\bigcup_{j=1}^{n} T_j = G$.

We call $\mathcal{G}_1$ *and* $\mathcal{G}_2$ genomes *over* G. *We denote the set of all genomes over* G *by* Γ_G. *The following three operations are called* syntenic operations*:*

- *A* fusion *is a function* $\phi : \Gamma_G \to \Gamma_G$ *such that the following holds for* $\mathcal{G}_2 = \phi(\mathcal{G}_1)$*: There exist* $i_1, i_2 \in \{1, \ldots, k\}$ *and* $j_1 \in \{1, \ldots, n\}$ *such that* $S_{i_1} \cup S_{i_2} = T_{j_1}$ *and* $\{S_i \mid i \notin \{i_1, i_2\}\} = \{T_j \mid j \neq j_1\}$ *hold.*
- *A* fission *is a function* $\psi : \Gamma_G \to \Gamma_G$ *such that the following holds for* $\mathcal{G}_2 = \psi(\mathcal{G}_1)$*: There exist* $i_1 \in \{1, \ldots, k\}$ *and* $j_1, j_2 \in \{1, \ldots, n\}$ *such that* $S_{i_1} = T_{j_1} \cup T_{j_2}$ *and* $\{S_i \mid i \neq i_1\} = \{T_j \mid j \notin \{j_1, j_2\}\}$ *hold.*
- *A* translocation *is a function* $\rho : \Gamma_G \to \Gamma_G$ *such that the following holds for* $\mathcal{G}_2 = \rho(\mathcal{G}_1)$*: There exist* $i_1, i_2 \in \{1, \ldots, k\}$ *and* $j_1, j_2 \in \{1, \ldots, n\}$ *such that* $S_{i_1} \cup S_{i_2} = T_{j_1} \cup T_{j_2}$ *and* $\{S_i \mid i \notin \{i_1, i_2\}\} = \{T_j \mid j \notin \{j_1, j_2\}\}$ *hold.*

Informally speaking, a fusion corresponds to merging two chromosomes, a fission corresponds to breaking one chromosome into two, and a translocation corresponds to the exchange of genes between two chromosomes.

Now we are ready to define the syntenic distance of two genomes formally.

Definition 10.10. *Let* G *be a finite set and let* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *be two genomes over* G. *Then we define the* syntenic distance $syn(\mathcal{G}_1, \mathcal{G}_2)$ *of* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *as the minimum number of syntenic operations necessary to transform* $\mathcal{G}_1$ *into* $\mathcal{G}_2$.

We illustrate the definitions with an example.

Example 10.5. Let $G = \{g_1, \ldots, g_8\}$ be a set and let $\mathcal{G}_1 = \{\{g_1, g_3, g_4\}, \{g_2, g_6\}, \{g_5, g_7, g_8\}\}$ and $\mathcal{G}_2 = \{\{g_1, g_2\}, \{g_3, g_4, g_5\}, \{g_6, g_7, g_8\}\}$ be two genomes. Then, $\mathcal{G}_1$ can be transformed into $\mathcal{G}_2$ by the following operations:

$$\begin{aligned}
\mathcal{G}_1 &= \{\{g_1, g_3, g_4\}, \{g_2, g_6\}, \{g_5, g_7, g_8\}\} \\
&\xrightarrow{\text{fusion}} \quad \{\{g_1, g_3, g_4\}, \{g_2, g_5, g_6, g_7, g_8\}\} \\
&\xrightarrow{\text{translocation}} \{\{g_1, g_2\}, \{g_3, g_4, g_5, g_6, g_7, g_8\}\} \\
&\xrightarrow{\text{fission}} \quad \{\{g_1, g_2\}, \{g_3, g_4, g_5\}, \{g_6, g_7, g_8\}\} = \mathcal{G}_2
\end{aligned}$$

Here, the first executed operation is a fusion of the second and the third subsets. The second operation is a translocation exchanging the elements g_3 and g_4 from the first subset for the element g_2 from the second subset. In the last step, a fission of the second subset is executed.

But this sequence of operations is not optimal; the transformation of $\mathcal{G}_1$ into $\mathcal{G}_2$ is also possible using only two translocations:

$$\begin{aligned}\mathcal{G}_1 &= \{\{g_1, g_3, g_4\}, \{g_2, g_6\}, \{g_5, g_7, g_8\}\} \\ &\longrightarrow \{\{g_1, g_2\}, \{g_3, g_4, g_6\}, \{g_5, g_7, g_8\}\} \\ &\longrightarrow \{\{g_1, g_2\}, \{g_3, g_4, g_5\}, \{g_6, g_7, g_8\}\} = \mathcal{G}_2\end{aligned}$$

◊

Lemma 10.6. *Let G be a finite set. Then the syntenic distance syn is a metric*[7] *on Γ_G.*

Proof. Obviously, $syn(\mathcal{G}, \mathcal{G}) = 0$ holds for any $\mathcal{G} \in \Gamma_G$, since no syntenic operation is needed to transform $\mathcal{G}$ into itself. It is also clear that the triangle inequality $syn(\mathcal{G}_1, \mathcal{G}_2) \leqslant syn(\mathcal{G}_1, \mathcal{G}_3) + syn(\mathcal{G}_3, \mathcal{G}_2)$ is satisfied for all $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \Gamma_G$. It remains for us to show that the syntenic distance is symmetric, i.e., that $syn(\mathcal{G}_1, \mathcal{G}_2) = syn(\mathcal{G}_2, \mathcal{G}_1)$ holds for all $\mathcal{G}_1, \mathcal{G}_2 \in \Gamma_G$. This immediately follows from the fact that there exists an inverse operation for each syntenic operation. For a fission, this is the fusion of the two constructed subsets; for a fusion, it is an appropriate fission; and for any translocation, there obviously exists another translocation undoing it. □

According to the symmetry shown in Lemma 10.6, we can assume without loss of generality that the second genome does not contain fewer subsets (chromosomes) than the first one. We will now formally define the problem of determining the syntenic distance.

Definition 10.11. *The problem of computing the syntenic distance, the* MinSynteny problem*, is the following optimization problem:*

Input: A finite set G and two genomes $\mathcal{G}_1, \mathcal{G}_2 \in \Gamma_G$.
Feasible solutions: Every sequence $\theta_1, \ldots, \theta_l$ of syntenic operations satisfying $\mathcal{G}_1\theta_1 \cdots \theta_l = \mathcal{G}_2$.
Costs: For a feasible solution $\theta_1, \ldots, \theta_l$, the costs are $cost(\theta_1, \ldots, \theta_l) = l$.
Optimization goal: Minimization.

As the following theorem shows, the MinSynteny problem is a hard optimization problem, and thus also the computing problem is hard.

Theorem 10.4. *The* MinSynteny *problem is NP-hard.* □

Theorem 10.4 was proved by DasGupta et al. [55], we skip the proof here.

In the following we show that the MinSynteny problem can be approximated with an approximation ratio of 2.

As the first step, we present a method for transforming any input instance of the MinSynteny problem to some type of normal form that will facilitate our proofs.

[7] Recall that a metric on a set S is a function $d : S \times S \rightarrow \mathbb{R}^{\geqslant 0}$ satisfying the conditions $d(x, x) = 0$, $d(x, y) = d(y, x)$ (symmetry), and $d(x, y) \leqslant d(x, z) + d(z, y)$ (triangle inequality) for all $x, y, z \in S$.

Definition 10.12. *Let G be a finite set and let $\mathcal{G}_1 = \{S_1, \dots, S_k\} \in \Gamma_G$ and $\mathcal{G}_2 = \{T_1, \dots, T_n\} \in \Gamma_G$ be two genomes over G, where $n \geqslant k$.*

We define $G' = \{1, \dots, n\}$, $\mathcal{G}'_2 = \{\{1\}, \dots, \{n\}\}$, and $\mathcal{G}'_1 = \{S'_1, \dots, S'_k\}$, where

$$S'_i = \bigcup_{x \in S_i} \{j \mid x \in T_j\}$$

for all $1 \leqslant i \leqslant k$. The pair $(\mathcal{G}'_1, \mathcal{G}'_2)$ is then called the compact representation *of $(G, \mathcal{G}_1, \mathcal{G}_2)$.*

The following example illustrates the computation of the compact representation.

Example 10.6. Consider $G = \{g_1, \dots, g_8\}$ and the genomes

$$\mathcal{G}_1 = \{\{g_1, g_3, g_4\}, \{g_2, g_6\}, \{g_5, g_7, g_8\}\}$$

and

$$\mathcal{G}_2 = \{\{g_1, g_2\}, \{g_3, g_4, g_5\}, \{g_6, g_7, g_8\}\}$$

from Example 10.5.

Then, $G' = \{1, 2, 3\}$ holds, and thus $\mathcal{G}'_2 = \{\{1\}, \{2\}, \{3\}\}$. The first subset of $\mathcal{G}_1$ contains elements from the first and the second subset of $\mathcal{G}_2$, the second subset of $\mathcal{G}_1$ contains elements from the first and the third subset of $\mathcal{G}_2$, and the third subset of $\mathcal{G}_1$ contains elements from the second and the third subset of $\mathcal{G}_2$. Hence, $\mathcal{G}'_1 = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$. ♢

The compact representation of an input instance for the MinSynteny problem is obviously computable in polynomial time. The following lemma shows that $(\mathcal{G}'_1, \mathcal{G}'_2)$ can really be used as a representation of $(G, \mathcal{G}_1, \mathcal{G}_2)$.

Lemma 10.7. *Let G be a finite set and let $\mathcal{G}_1 = \{S_1, \dots, S_k\} \in \Gamma_G$ and $\mathcal{G}_2 = \{T_1, \dots, T_n\} \in \Gamma_G$ be two genomes over G, where $n \geqslant k$. Moreover, let $(\mathcal{G}'_1, \mathcal{G}'_2)$ be the compact representation of $(G, \mathcal{G}_1, \mathcal{G}_2)$.*

Then, $syn(\mathcal{G}_1, \mathcal{G}_2) = syn(\mathcal{G}'_1, \mathcal{G}'_2)$.

Proof idea. The proof of the claim is based on the following idea: To transform $\mathcal{G}_1$ into $\mathcal{G}_2$ by a sequence of syntenic operations, in the set S_i, $1 \leqslant i \leqslant k$, all the elements that are to be transferred into the same set T_j can be grouped together. Then it is possible to show that there always exists an optimal sequence of syntenic operations where no such group of elements is divided by fission or translocation.

This means that every set $\{j\}$ in the definition of $\mathcal{G}'_2$ stands for the subset T_j of $\mathcal{G}_2$, and every $j \in S'_i$ in the definition of $\mathcal{G}'_1$ stands for the set of all elements that have to be moved to T_j. □

Note that the definition of $\mathcal{G}'_1$ may lead to the situation where an element j might occur in several sets S'_i, as we have seen in Example 10.6. This is the

reason we defined the syntenic operations not just for partitions of a ground set in Definition 10.9.

For the remainder of this section we assume that all input instances for the MinSynteny problem are given by their compact representation. An input instance for the MinSynteny problem can be described by a graph as follows.

Definition 10.13. *Let $I = (\mathcal{G}_1, \mathcal{G}_2)$ be the compact representation of an input instance for the* MinSynteny *problem, where $\mathcal{G}_2 = \{\{1\}, \ldots, \{n\}\}$ and $\mathcal{G}_1 = \{S_1, \ldots, S_k\}$. The* synteny graph *for I is an undirected graph $Syngraph(I) = (V, E)$ with $V = \mathcal{G}_1$ and $\{S_i, S_j\} \in E$ if and only if $i \neq j$ and $S_i \cap S_j \neq \emptyset$, for all $1 \leqslant i, j \leqslant k$.*

If p is the number of connected components of $Syngraph(I)$, we say that p is the number of components *of I.*

Informally, the synteny graph can be described as the intersection graph of the sets from $\mathcal{G}_1$.

We now show how to derive a lower bound on the syntenic distance based on the number of components of the input instance.

Lemma 10.8. *Let $I = (\mathcal{G}_1, \mathcal{G}_2)$ be the compact representation of an input instance for the* MinSynteny *problem with p components. Moreover, let $\mathcal{G}_2 = \{\{1\}, \ldots, \{n\}\}$ and $\mathcal{G}_1 = \{S_1, \ldots, S_k\}$.*

Then, $syn(\mathcal{G}_1, \mathcal{G}_2) \geqslant n - p$.

Proof. Let $\theta_1, \ldots, \theta_m$ be an optimal sequence of syntenic operations for I. Let $\mathcal{H}_0 = \mathcal{G}_1$ and let $\mathcal{H}_l = \mathcal{G}_1\theta_1 \ldots \theta_l$ for all $1 \leqslant l \leqslant m$. Moreover, let $I_l = (\mathcal{H}_l, \mathcal{G}_2)$ for all $0 \leqslant l \leqslant m$. According to our assumption, p is the number of components of I_0, and n is the number of components of I_m. We show that I_l can have at most one more component than I_{l-1}. This immediately implies the claim.

Let $\mathcal{H}_{l-1} = \{T_1, \ldots, T_{k_{l-1}}\}$ and $\mathcal{H}_l = \{T'_1, \ldots, T'_{k_l}\}$. We distinguish three cases according to the type of the syntenic operation θ_l.

- If θ_l is a fusion merging T_i and T_j to yield T'_q, then the two components containing T_i and T_j in $Syngraph(I_{l-1})$ are also merged in $Syngraph(I_l)$, and all other components remain the same. In this case $Syngraph(I_l)$ hence has at most as many components as $Syngraph(I_{l-1})$.
- If θ_l is a fission dividing T_i into T'_q and T'_r, then there are two possible situations. If the component of $Syngraph(I_{l-1})$ containing T_i remains connected without T_i, then T'_q and T'_r lie inside the same component in $Syngraph(I_l)$, and the number of components does not change. If the component C of $Syngraph(I_{l-1})$ containing T_i gets disconnected without T_i, then $Syngraph(I_l)$ has exactly one component more than $Syngraph(I_{l-1})$, since all vertices from C lie either in the component of T'_q or in the component of T'_r in $Syngraph(I_l)$.
- If θ_l is a translocation that transforms T_i and T_j into the new sets T'_q and T'_r, we again distinguish two situations. If the vertices T'_q and T'_r are connected by an edge in $Syngraph(I_l)$, then $Syngraph(I_l)$ has at most as

Algorithm 10.2 Approximation algorithm for the MinSynteny problem

Input: The compact representation $I = (\mathcal{G}_1, \mathcal{G}_2)$ of an input instance for the MinSynteny problem, where $\mathcal{G}_2 = \{\{1\}, \ldots, \{n\}\}$ and $\mathcal{G}_1 = \{S_1, \ldots, S_k\}$.

1. Compute the synteny graph *Syngraph*(I) and determine its connected components $C_1, \ldots, C_p$.
2. For each connected component, execute as many fusions as necessary to shrink it to a single vertex.
3. Separate the single elements within each vertex by a sequence of fissions.

Output: The sequence of executed fusions and fissions.

many components as *Syngraph*(I_{l-1}), since every path through T_i or T_j in *Syngraph*(I_{l-1}) can be substituted by a path through T'_q or T'_r, or both, in *Syngraph*(I_l). If the edge between the vertices T'_q and T'_r is missing, then *Syngraph*(I_l) has at most one more component than *Syngraph*(I_{l-1}), since deleting an edge can increase the number of components by at most one. □

Now, Algorithm 10.2 computes a 2-approximation for the MinSynteny problem; it uses only fusions and fissions.

Theorem 10.5. *Algorithm 10.2 is a polynomial-time* 2*-approximation algorithm for the* MinSynteny *problem.*

Proof. Let k_i be the number of vertices within the component C_i and let n_i be the number of all distinct elements in all sets in C_i, for all $1 \leqslant i \leqslant p$. According to the definition of *Syngraph*(I), $\sum_{i=1}^{p} n_i = n$ and $\sum_{i=1}^{p} k_i = k$ hold. The algorithm executes exactly $k_i - 1$ fusions and $n_i - 1$ subsequent fissions for each component C_i; this adds up to $n + k - 2p$ syntenic operations. From $n \geqslant k$ we know that $n + k - 2p \leqslant 2 \cdot (n - p)$. Since according to Lemma 10.8 at least $n - p$ syntenic operations are necessary, the algorithm yields a 2-approximation.

All steps of the algorithm are obviously executable in polynomial time; an efficient algorithm for computing the connected components of a graph can be found in Chapter 21 of the book by Cormen et al. [51]. □

The following example shows that there exists an input where Algorithm 10.2 needs exactly twice as many syntenic operations as an optimal solution.

Example 10.7. Let $I = (\mathcal{G}_1, \mathcal{G}_2)$ be the compact representation of an input instance for the MinSynteny problem, where $\mathcal{G}_2 = \{\{1\}, \ldots, \{n\}\}$ and $\mathcal{G}_1 = \{\{1\}, \{1,2\}, \{1,2,3\}, \ldots, \{1,2,\ldots,n\}\}$.

Then the synteny graph for i consists of exactly one component, and Algorithm 10.2 executes $n - 1$ fusions and $n - 1$ fissions.

But, in this example, $n - 1$ translocations are sufficient: First, execute a translocation transforming the sets $\{1, 2, \ldots, n-1\}$ and $\{1, 2, \ldots, n\}$ into the

sets $\{1, 2, \ldots, n-1\}$ and $\{n\}$, and continue this iteratively to transform the sets $\{1, 2, \ldots, n-i\}$ and $\{1, 2, \ldots, n-i+1\}$ into the sets $\{1, 2, \ldots, n-i\}$ and $\{n-i+1\}$. $\Diamond$

10.5 Summary

One approach for estimating the degree of relationship between organisms or species is the comparison of whole genomes, based on the order of the genes. This approach is especially useful for comparing closely related genomes where the sequences of homologous genes do not differ too much, but the order of the genes within the genome is rather different.

In genome rearrangements, one assumes that a linear order of the genes is known for two given genomes, and that one order has been reached from the other by a sequence of operations such as reversals. One tries to find a sequence of as few reversals as possible that produces this transformation. The problem can be modeled by undirected permutations if the reading direction of the genes is unknown. The resulting optimization problem is NP-hard, but there exist several approximation algorithms. If, additionally, the reading direction of the genes is known, the problem can be modeled by directed permutations, which leads to an optimization problem that is solvable in polynomial time.

Another model for comparing whole genomes is the computation of the syntenic distance. Here, one assumes that the genes of a genome are spread over several chromosomes, and that only the corresponding chromosomes are known for the genes, not the exact order of the genes. Computing the syntenic distance is also NP-hard, and for this problem there are also approximation algorithms known with a constant approximation ratio.

10.6 Bibliographic Notes

A detailed presentation of the theory of genome rearrangements can be found in the books by Setubal and Meidanis [180] and Pevzner [159].

The modeling of genome rearrangements by undirected permutations goes back to Watterson et al. [206]. Caprara [40] has shown the NP-hardness of this problem; the 2-approximation algorithm presented is due to Kececioglu and Sankoff [121], and our presentation of the algorithm is based on that in the book by Setubal and Meidanis [180]. There exist several improved approximation algorithms for the problem; the best currently known algorithm achieves an approximation ratio of 1.375 and was designed by Berman et al. [31].

A polynomial-time algorithm for sorting a directed permutation by reversals was invented by Hannenhalli and Pevzner [96]; simplifications to their proof can be found in the papers by Bergeron [28] and Bergeron et al. [29].

The paper by Hannenhalli and Pevzner [96] also contains the biological example shown in Figure 10.2. More efficient algorithms for sorting directed permutations were designed by Berman and Hannenhalli [30] and by Kaplan et al. [113]. A generalization of this approach to genomes with more than one chromosome is also due to Hannenhalli and Pevzner [97].

The model of the syntenic distance was proposed by Ferretti et al. [73], the approximation algorithm presented is due to DasGupta et al. [55]. The model was further investigated by Liben-Nowell and Kleinberg [136, 138, 122, 137] and by Pisanti and Sagot [162]. The suitability of syntenic data for estimating distances between genomes was investigated by Ehrlich et al. [174, 65] using statistical methods.

A detailed overview of the different models of genome rearrangements containing several approaches not presented in this book is given in the paper by Sankoff and El-Mabrouk [173]. Another approach, investigated by Bafna and Pevzner [20], sorts a permutation by transpositions (see Figure 10.1 (b)).

Instead of merely counting the number of operations required to transform one genome into another, Pinter and Skiena [161] considered weighted reversal operations, assigning a certain weight to each reversal with respect to the length it operates on, and tried to find a reversal sequence of minimum weight. Further studies in this context were performed by Bender et al. [24] and Swidan et al. [188].

11

Phylogenetic Trees

An important topic in biology is the reconstruction of kindred relations within a given set of biological objects. These objects can, for example, be different biological species or even different single genes. In this context, these objects are often called *taxa*.[1] The most commonly used approach for reconstructing relations between taxa is the construction of a *phylogenetic tree*, also called *phylogeny* for short. This denotes a tree where each leaf is labeled with exactly one of the taxa and where the inner vertices represent their hypothetical ancestors, such that the distance between two taxa in the tree serves as a measure for their degree of relationship.

There are many different models and approaches for constructing phylogenetic trees, some of which we will present in this chapter. The different models can, on the one hand, be classified by the goal they are trying to reach, i.e., by the conditions the constructed tree has to satisfy, and, on the other hand, by the type of given information about the relation of the taxa.

In most cases, we assume that a phylogenetic tree is a binary tree, i.e., that each inner vertex has exactly three neighbors (except for a root vertex, if it exists, which may have two neighbors). If there is a root known in the tree, it represents the common ancestor of all taxa. In this case, the tree determines an unambiguous direction of evolution. Restricting ourselves to binary trees means that we assume that an elementary evolutionary step always consists of dividing one taxon into two different taxa. Some methods for determining phylogenies do not allow us to deduce the direction of evolution. In this case, we can only construct an undirected tree without a distinct root vertex. Another difference between the approaches is that we are in many cases only able to determine the topology of the tree, i.e., its branching structure, but in other cases we are also able to assign lengths to the edges of the tree that correspond to the time that has passed between the two branching events at

[1] The notion *taxa* (in singular, *taxon*) is derived from *taxonomy* (from the Greek *taxis* = order, classification), denoting the classification of organisms in a biological system.

the ends of the edge, or the time from the last branching event up to the present if the edge is incident to a leaf.

According to the data given for constructing the phylogenetic tree, we distinguish mainly two types of information. The first type of information is a distance measure assigning a distance to each pair of taxa. This can for example be an alignment score if the taxa are given as homologous genes from different individuals, or it can be any of the measures for the comparison of whole genomes, as described in Chapter 10, if the taxa are whole genomes (or their respective biological species). The other type of information that can serve as an input is a set of attributes (often called *characters*) that can take one of finitely many values (often called *states*) for each of the taxa, together with a matrix describing which state each character takes for each taxon. If the taxa are biological species, the characters might be phenotypical data like the number of legs, the ability to fly, or the color of hair, but may also be genotypical data like the number of chromosomes, the presence of distinct genes, or something similar. If we want to construct a phylogenetic tree for a set of homologous genes, we can also use a multiple alignment of the gene sequences for determining the characters; every position of the alignment defines a character; the state at a position corresponds to the nucleotide at the position.

As we can see from the above discussion, constructing phylogenetic trees is a very extensive area; it would require a book of its own to cover the topic in depth. We thus limit ourselves here to presenting some examples for algorithmically interesting approaches. The chapter is organized as follows: In the first two sections we present two approaches for constructing a phylogenetic tree from the pairwise distances of the given taxa. Section 11.1 deals with the special case of ultrametric distances; in this case, the constructed phylogenetic tree can be uniquely determined. In Section 11.2 we characterize the set of distance measures for which it is possible to construct a phylogenetic tree that reflects the given distances. The next two sections are dedicated to determining a phylogenetic tree from character data. In Section 11.3 we investigate characters that can have only two distinct states, and in Section 11.4 we present some methods using the DNA sequences of homologous genes as character data. The chapter closes with a summary in Section 11.5 and bibliographic notes in Section 11.6.

11.1 Ultrametric Distances

In this section we deal with computing a phylogenetic tree for a given set of taxa for which a distance measure is known for the pairwise distances between the taxa. We do not investigate arbitrary distance measures, but only those having some useful properties. To describe these properties in greater detail, we first need the formal definition of a metric.

Definition 11.1. *Let A be a set of taxa. Let $d : A \times A \to \mathbb{Q}^{\geqslant 0}$ be a function. Then, d is a* metric *on A if it satisfies the following properties:*

(i) For all $a, b \in A$, $d(a, b) = 0$ holds if and only if $a = b$.
(ii) For all $a, b \in A$, $d(a, b) = d(b, a)$ (symmetry).
(iii) For all $a, b, c \in A$, $d(a, b) \leqslant d(a, c) + d(c, b)$ (triangle inequality).

In the following, we consider a further restriction of metric distance measures, as described by the following definition.

Definition 11.2. *Let A be a set of taxa. Let $d : A \times A \to \mathbb{Q}^{\geqslant 0}$ be a metric on A. Then d is a* ultrametric *if it additionally satisfies the following* three point condition*:*

For all $a, b, c \in A$, two of the distances $d(a, b), d(a, c), d(b, c)$ are equal and not smaller than the third one.

The three point condition says that, for three arbitrarily chosen taxa $a, b, c \in A$, one of the conditions $d(a, b) \leqslant d(a, c) = d(b, c)$, $d(a, c) \leqslant d(a, b) = d(b, c)$, and $d(b, c) \leqslant d(a, b) = d(a, c)$ holds. This means that the three point condition is a stronger restriction than the triangle inequality.

Our goal is to find a phylogenetic tree with a root where the edge lengths are also known, and where the path lengths from the root to a leaf are equal for all leaves. Here, the length of a path is defined, as usual, as the sum of edge lengths on the path. Such a phylogenetic tree is called an *ultrametric tree*. An ultrametric tree corresponds to an ideal model of evolution where the evolution speed is the same within each branch of the tree.

We start with a formal definition of an ultrametric tree.

Definition 11.3. *Let $A = \{a_1, \ldots, a_n\}$ be a set of taxa. A directed edge-weighted tree $T = (V, E, d)$ with root r and edge weight function $d : E \to \mathbb{Q}^{\geqslant 0}$ is an* ultrametric tree *for A if it satifies the following conditions:*

(i) T is a binary tree, i.e., every inner vertex of T has exactly two successors.
(ii) T has exactly n leaves, labeled with the taxa $\{a_1, \ldots, a_n\}$.
(iii) The sum of edge weights on every path from the root to any leaf is the same.

The distance *between two arbitrary vertices x and y of T is the sum of edge weights on the path from x to y in T; we denote it by $dist_T(x, y)$.*

We first illustrate Definition 11.3 with an example.

Example 11.1. In Figure 11.1, an ultrametric tree for the taxa $\{a, b, c, d, e\}$ is shown. Since the direction of the edges is uniquely determined by fixing the root, we have not shown it explicitly in the figure. In this example, $dist_T(r, x) = 6$ holds for all leaves $x \in \{a, b, c, d, e\}$. ◊

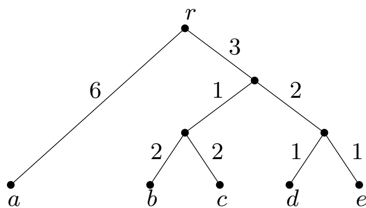


Fig. 11.1. An example of an ultrametric tree

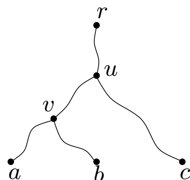


Fig. 11.2. The situation in the proof of Lemma 11.1

We show that an ultrametric tree for a given set of taxa, together with a given distance measure where the distances between the leaves correspond to the given distances between the taxa, exists if and only if the given distance measure is an ultrametric. We start by proving one direction of this claim.

Lemma 11.1. *Let A be a set of taxa and let $T = (V, E, d)$ be an ultrametric phylogenetic tree for A. Then, the distances $dist_T$ between the taxa in T define an ultrametric on A.*

Proof. Let $a, b, c \in A$ be three leaves in T. It suffices to show that $dist_T(a, b)$, $dist_T(a, c)$, and $dist_T(b, c)$ satisfy the three point condition, since conditions (i) and (ii) of Definiton 11.1 are trivially satisfied and the three point condition is a stronger condition than the triangle inequality. We first assume that $a = b$ holds. Then,

$$0 = dist_T(a, b) \leqslant dist_T(a, c) = dist_T(b, c);$$

hence, the three point condition is satisfied. In the following, we assume that a, b, and c are pairwise distinct. Without loss of generality, we further assume that there exists a vertex v that is a common ancestor of a and b, but not of c. Moreover, let u be the least common ancestor of a, b, and c, and let r denote the root of T. This leads to the situation shown in Figure 11.2. Since T is an ultrametric tree, we know that

$$dist_T(r, a) = dist_T(r, b) = dist_T(r, c). \tag{11.1}$$

Since the paths from r to a and from r to b differ only below v, this also implies that $dist_T(v, a) = dist_T(v, b)$. Thus,

$$dist_T(a, c) = dist_T(a, v) + dist_T(v, c) = dist_T(b, v) + dist_T(v, c) = dist_T(b, c).$$

To satisfy the three point condition, it remains for us to show that $dist_T(a, b) \leqslant dist_T(a, c)$ holds. Equation (11.1) implies $dist_T(u, a) = dist_T(u, c)$. Thus,

$$\begin{aligned} dist_T(a, c) &= 2 \cdot dist_T(a, u) \\ &= 2 \cdot (dist_T(a, v) + dist_T(v, u)) \\ &\geqslant 2 \cdot (dist_T(a, v)) \\ &= dist_T(a, b). \end{aligned}$$

The distances between the taxa a, b, and c hence satisfy the three point condition, and $dist_T$ is thus an ultrametric. □

We now constructively prove the other direction, showing that there always exists an ultrametric tree for a set A of n taxa for any given ultrametric distance function on A. We design an algorithm constructing an ultrametric tree for this type of input. The algorithm is based on the following idea.

The vertices of the tree will be subsets of A. The algorithm starts with the set of all singleton subsets of A as vertices and computes their pairwise distances. The vertices will later become the leaves of the tree, but they are not yet connected. Next, the algorithm chooses two subsets at the minimum distance in the forest constructed so far that do not yet have an ancestor in the forest, and their union is added as a vertex to the forest, connected via edges to the two chosen subsets. Then the algorithm calculates the distance of the new vertex to all other vertices in the graph. After $n - 1$ steps, this yields a tree whose root is labeled with the whole set A. As we later show, the tree constructed in this way is ultrametric. This idea is described in greater detail in Algorithm 11.1, also called the *UPGMA algorithm* [2].

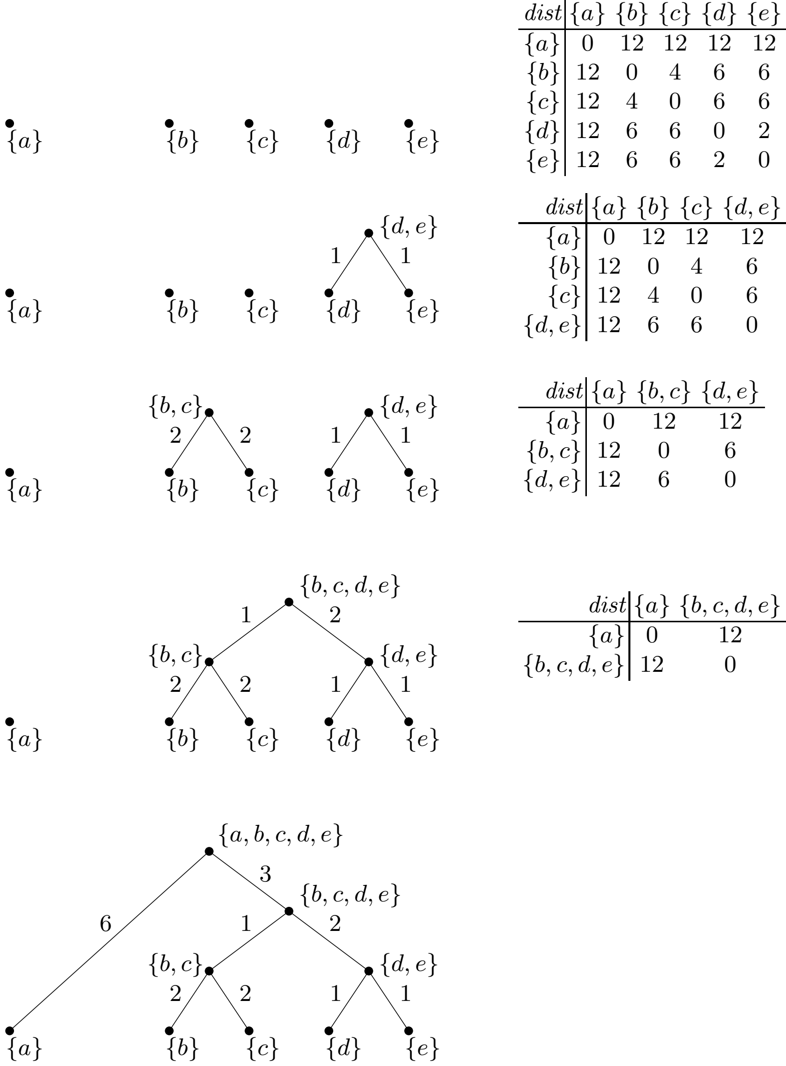
Before we prove the correctness of the UPGMA algorithm, we first illustrate its work with an example.

Example 11.2. Consider the set $A = \{a, b, c, d, e\}$ of taxa with the following distance function $\delta : A \times A \to \mathbb{N}$:

δ	a	b	c	d	e
a	0	12	12	12	12
b	12	0	4	6	6
c	12	4	0	6	6
d	12	6	6	0	2
e	12	6	6	2	0

It is easy to check that δ is an ultrametric distance function. The work of Algorithm 11.1 on the input A and δ is shown in Figure 11.3. There, for each step of the algorithm the already constructed part of the ultrametric tree as well as the distance function *dist* on the current set Γ are shown. As can be seen from the figure, the algorithm outputs the ultrametric tree from Example 11.1. ◊

[2] The acronym UPGMA stands for *Unweighted Pair Group Method with Arithmetic Mean.*



$dist$	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$
$\{a\}$	0	12	12	12	12
$\{b\}$	12	0	4	6	6
$\{c\}$	12	4	0	6	6
$\{d\}$	12	6	6	0	2
$\{e\}$	12	6	6	2	0

$dist$	$\{a\}$	$\{b\}$	$\{c\}$	$\{d,e\}$
$\{a\}$	0	12	12	12
$\{b\}$	12	0	4	6
$\{c\}$	12	4	0	6
$\{d,e\}$	12	6	6	0

$dist$	$\{a\}$	$\{b,c\}$	$\{d,e\}$
$\{a\}$	0	12	12
$\{b,c\}$	12	0	6
$\{d,e\}$	12	6	0

$dist$	$\{a\}$	$\{b,c,d,e\}$
$\{a\}$	0	12
$\{b,c,d,e\}$	12	0

Fig. 11.3. The work of Algorithm 11.1 on the input instance from Example 11.2

Algorithm 11.1 UPGMA algorithm for constructing an ultrametric tree

Input: A set $A = \{a_1, \ldots, a_n\}$ of taxa and an ultrametric distance function $\delta : A \times A \to \mathbb{Q}^{\geqslant 0}$.

1. Initialization:
   a) Define $\Gamma := \{\{a_1\}, \ldots, \{a_n\}\}$.
   b) Define $dist(\{a_i\}, \{a_j\}) := \delta(a_i, a_j)$ for all $i, j \in \{1, \ldots, n\}$.
   c) Define $height(\{a_i\}) := 0$ for all $i \in \{1, \ldots, n\}$.
   d) Let $V := \Gamma$ and $E := \emptyset$.
2. **while** $|\Gamma| \geqslant 2$ **do**
   a) Find $C_1, C_2 \in \Gamma$, $C_1 \neq C_2$, such that $dist(C_1, C_2)$ is minimal, and define $D := C_1 \cup C_2$.
   b) Let $\Gamma := (\Gamma - \{C_1, C_2\}) \cup \{D\}$.
   c) Let $dist(D, X) = dist(X, D) := \dfrac{dist(C_1, X) + dist(C_2, X)}{2}$ for all $X \in \Gamma$.
   d) Define $V := V \cup \{D\}$ and $E := E \cup \{(D, C_1), (D, C_2)\}$.
   e) Define $height(D) := \dfrac{dist(C_1, C_2)}{2}$.
   f) Define $d(D, C_1) := height(D) - height(C_1)$ and
   $d(D, C_2) := height(D) - height(C_2)$.

Output: The ultrametric tree $T = (V, E, d)$ for A.

We now prove the correctness of the UPGMA algorithm and estimate its running time.

Theorem 11.1. *For a given set $A = \{a_1, \ldots, a_n\}$ of taxa and an ultrametric distance function δ on A, Algorithm 11.1 computes an ultrametric tree for A in $O(n^3)$ time.*

Proof. We start with proving the correctness of Algorithm 11.1. The algorithm obviously constructs a tree. It starts with the set of leaves $\{a_1\}, \ldots, \{a_n\}$ and adds a new vertex in each execution of the while loop in step 2 (d) that merges two already existing subtrees. By definition of the edge weights in step 2 (f) it is clear that in every subtree the distance from its root to all leaves is the same. We call this distance from a vertex X to all leaves in the subtree rooted at X the height of X, and we denote it by $height(X)$. The height is initialized with 0 in step 1 (c) for the leaves and calculated for a new vertex in step 2 (e).

It remains for us to show that the edge weights computed in step 2 (f) are well defined, i.e., that they will never get negative. To prove this, it suffices to show, for a newly constructed vertex D with its children C_1 and C_2, that $height(D) \geqslant height(C_1)$ and $height(D) \geqslant height(C_2)$ holds. This follows from the fact that the algorithm chooses two subsets with minimum distance in step 2 (a). For a formal proof, we show the following generalization: Let D_i be the vertex added in the i-th iteration of step 2, let $C_{1,i}$ and $C_{2,i}$ be its children, let V_i be the vertex set after iteration i, and let Γ_i be the set Γ after

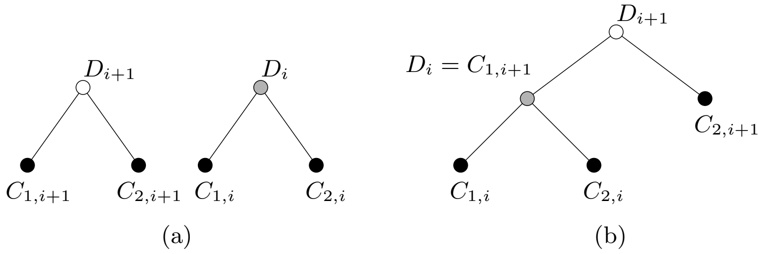


Fig. 11.4. The two cases in the proof of Inequality (11.2). The vertices in Γ_{i-1} are drawn black, the vertices in $\Gamma_i - \Gamma_{i-1}$ are drawn grey, and the vertices in $\mathcal{C}_{i+1} - \Gamma_i$ are drawn white

iteration i. Then, for all $1 \leqslant i \leqslant n-1$ and for all $X \in V_{i-1}$ the following holds:

$$height(D_i) \geqslant height(X).$$

We prove this claim by induction on the number i of iterations. For the first iteration, the claim follows immediately, since all leaves have height 0 according to step 1 (c). For the induction step, we consider the vertex D_{i+1}. Due to the induction hypothesis, it is sufficient to show that $height(D_{i+1}) \geqslant height(D_i)$, since only the vertex D_{i+1} was added in iteration $i+1$. According to the computation of *height* in step 2 (e) of the algorithm it remains to show that

$$dist(C_{1,i+1}, C_{2,i+1}) \geqslant dist(C_{1,i}, C_{2,i}). \tag{11.2}$$

For the proof of Inequality (11.2), we distinguish two cases according to whether D_i is a child of D_{i+1} or not. The two cases are shown schematically in Figure 11.4. We first consider the case where D_i is not a child of D_{i+1}. The resulting situation is shown in Figure 11.4 (a). In particular, this implies that $C_{1,i+1}, C_{2,i+1}, C_{1,i}, C_{2,i} \in \Gamma_{i-1}$. Since in step 2 (a) of the i-th iteration the vertices $C_{1,i}$ and $C_{2,i}$ were chosen as those with the minimum distance $dist(C_{1,j}, C_{2,j})$, Inequality (11.2) follows immediately.

We now investigate the case where D_i is a child of D_{i+1}. Without loss of generality we assume that $D_i = C_{1,i+1}$ holds. The resulting situation is shown in Figure 11.4 (b). In particular, $C_{2,i+1}, C_{1,i}, C_{2,i} \in \Gamma_{i-1}$. Due to the criterion of choice in step 2 (a), this implies $dist(C_{1,i}, C_{2,i}) \leqslant dist(C_{1,i}, C_{2,i+1})$ and $dist(C_{1,i}, C_{2,i}) \leqslant dist(C_{2,i}, C_{2,i+1})$, since the vertices $C_{1,i}$ and $C_{2,i}$ were chosen in the i-th iteration. Following the computation of *dist* in step 2 (c), this leads to

$$\begin{aligned} dist(C_{1,i+1}, C_{2,i+1}) &= dist(D_i, C_{2,i+1}) \\ &= \frac{dist(C_{1,i}, C_{2,i+1}) + dist(C_{2,i}, C_{2,i+1})}{2} \\ &\geqslant \frac{dist(C_{1,i}, C_{2,i}) + dist(C_{1,i}, C_{2,i})}{2} \\ &= dist(C_{1,i}, C_{2,i}). \end{aligned}$$

This proves the correctness of the UPGMA algorithm. We now analyze its running time.

The initialization of Γ, *height*, V, and E in steps 1 (a), (c), and (d) can be done in $O(n)$ time, the initialization of *dist* in step 1 (b) needs time in $O(n^2)$. Overall, step 1 has a running time in $O(n^2)$. The while loop in step 2 is executed exactly $n - 1$ times, since exactly one vertex is added in each iteration, and a binary tree with n leaves has exactly $n - 1$ inner vertices. Step 2 (a) needs time in $O(n^2)$, since the minimum of $O(n^2)$ values has to be calculated. All other substeps during one iteration of the while loop are obviously possible in $O(n)$ time. Overall, step 2, and thus the entire algorithm, runs in $O(n^3)$ time. □

It is even possible to show that the ultrametric tree computed by the UPGMA algorithm is unique up to isomorphism for every input with an ultrametric distance function. For details we refer the reader to the bibliographic remarks in Section 11.6.

Finally, we note that the UPGMA algorithm can also be used as a heuristic for inputs where the distance function is not ultrametric. But in this case the calculated edge lengths cannot be used since they might become negative. In spite of this, the topology of the computed tree is in many cases a useful approximation of the real phylogenetic tree.

11.2 Additive Trees

In the previous section we saw how to reconstruct a phylogenetic tree for a set of taxa with ultrametric distances. Unfortunately, the distance data occurring in practice often is not ultrametric. In this section we present a method for constructing a phylogenetic tree under slightly weaker assumptions about the given distance measure. But for this purpose we consider a slightly different model, where we allow the taxa to be assigned not only to the leaves of the tree but also to its inner vertices. This model is especially suitable for reconstructing a phologenetic tree for a set of taxa containing not only extant species but also extinct species that are supposed to belong among the ancestors of the extant ones considered. We start by formally defining this model.

Definition 11.4. *Let A be a set of n taxa and let $\delta : A \times A \to \mathbb{Q}^{\geqslant 0}$ be a metric distance measure on A. Let $T = (V, E, d)$ be an edge-weighted tree with $A \subseteq V$. For all $a, b \in A$, let dist(a, b) denote the sum of edge weights on the path from a to b in T. The tree T is called an* additive tree *for A and δ if dist$(a, b) = \delta(a, b)$ holds for all $a, b \in A$.*

The problem we face now is to decide, for a given set A of taxa with a given distance measure δ, whether there exists an additive tree for A and δ. We will see that it is possible to answer this question in polynomial time.

In the following, we restrict our attention to the special case of the problem where one of the given taxa is assigned to each vertex in the tree.

Definition 11.5. *Let A be a set of taxa and let δ be a metric distance measure on A. An additive tree $T = (V, E, d)$ for A and δ is called a* compact additive tree *for A and δ if $V = A$ holds.*

We now formally define the problem we want to solve in the remainder of this section.

Definition 11.6. *The problem of determining a compact additive tree, the* CompactAddTree problem*, is the following computing problem:*

Input: A set A of taxa and a metric distance measure $\delta : A \times A \to \mathbb{Q}^{\geqslant 0}$.
Output: A compact additive tree for A and δ, if it exists, and an error message otherwise.

To solve the CompactAddTree problem we use the following representation of the input data.

Definition 11.7. *Let A be a set of taxa and let δ be a metric distance measure on A. The* distance graph *for A and δ is the complete edge-weighted graph $G(A, \delta) = (V, E, d)$ with $V = A$ and $d(a, b) = \delta(a, b)$ for all $a, b \in A$.*

The following theorem shows how to compute a compact additive tree from the distance graph.

Theorem 11.2. *Let A be a set of taxa and let δ be a metric distance measure on A. If there exists a compact additive tree T for A, then T is the uniquely determined minimum spanning tree of $G(A, \delta)$.*

Proof. Let T be a compact additive tree for A and δ. We show that no edge outside of T can be contained in a minimum spanning tree. Let $e = \{x, y\}$ be an edge not contained in T. Since the path from x to y in T has to have a total weight of $\delta(x, y)$, and since all edge weights in T are strictly greater than 0, $\delta(x, y)$ has to be strictly greater than the weight of every single edge on the path from x to y in T. We show that e cannot be contained in any minimum spanning tree of $G(A, \delta)$. This implies that T is the unique minimum spanning tree of $G(A, \delta)$.

For the proof, we assume that $e = \{x, y\}$ is not contained in T, but in a minimum spanning tree $T' = (A, E')$ of $G(A, \delta)$. Let $G' = (A, E' - \{e\})$ be the graph originating from deleting the edge e from T', and let S and S' be the connected components of G'. Without loss of generality we assume that x lies inside S and y lies inside S'. Then, there exists an edge e' on the path P from x to y in T connecting a vertex from S with a vertex from S'. Since e' is an edge in T, in particular, since $e' \neq e$ holds, and since e' connects S and S', it is not an edge in T'. This implies that also $T'' = (A, (E' - \{e\}) \cup \{e'\})$ is a spanning tree of $G(A, \delta)$. As shown above, the weight $\delta(x, y)$ of e is strictly greater

than the weight of any edge on the path P, and thus strictly greater than the weight of e'. Hence, T'' has a strictly lower weight than T', contradicting the assumption that T' is a minimum spanning tree. □

Algorithm 11.2 solves the CompactAddTree problem by computing a minimum spanning tree of $G(A, \delta)$ and simultaneously checking its uniqueness as well as the additivity constraint. The computation of the minimum spanning tree is based on Prim's algorithm, i.e., the algorithm starts with an arbitrary vertex and successively adds the cheapest edge from the so far constructed subtree to a vertex not yet reached. If there is more than one cheapest edge in some step, the algorithm rejects the input, since the minimum spanning tree cannot be unique in this case.

Algorithm 11.2 Computation of a compact additive tree

Input: A set $A = \{a_1, \ldots, a_n\}$ of taxa and a metric distance measure δ on A.

1. Compute the distance graph $G(A, \delta) = (V, E')$.
2. Initialize the tree $T = (U, E, \delta)$ with $U := \{a_1\}$ and $E := \emptyset$.
3. **while** $U \neq V$ **do**
   a) Compute the edge $\{x, y\}$ in $G(A, \delta)$ with the smallest weight such that $x \in U$ and $y \in V - U$.
   b) Check whether the chosen edge is uniquely determined. If not, stop the computation with the output "There is no compact additive tree."
   c) Check whether if the additivity constraint for y and for all vertices in U is satisfied. If not, stop the computation with the output "There is no compact additive tree."
   d) Define $U := U \cup \{y\}$ and $E := E \cup \{\{x, y\}\}$.

Output: The compact additive tree $T = (U, E, \delta)$.

We now prove the correctness of Algorithm 11.2 and analyze its running time. For the proof of correctness, we need the following well-known result.

Lemma 11.2. *Let $G = (V, E, d)$ be a complete edge-weighted graph, let U be a proper subset of V, and let $\{u, v\}$ be an edge of minimal weight such that $u \in U$ and $v \in V - U$. Then there exists a minimum spanning tree of G containing the edge $\{u, v\}$.*

Proof. For the proof we assume that there is no minimum spanning tree of G containing $\{u, v\}$. Let T be an arbitrary minimum spanning tree of G. If we add $\{u, v\}$ to T, this generates a cycle containing the edge $\{u, v\}$. Since u and v are also connected in T and $u \in U, v \in V - U$ holds, there is another edge $\{u', v'\}$ inside this cycle with $u' \in U$ and $v' \in V - U$. The graph $T' = (T \cup \{u, v\}) - \{u', v'\}$ is another spanning tree of G that is no more expensive than T due to the minimal costs of $\{u, v\}$. This contradicts the assumption that there is no minimum spanning tree containing $\{u, v\}$. □

Theorem 11.3. *Algorithm 11.2 solves the* CompactAddTree *problem for a set A of n taxa and a metric distance function δ in $O(n^2 \log n)$ time.*

Proof. We first prove the correctness of the algorithm. Since, for every iteration of the while loop in step 3, the claim of Lemma 11.2 holds for the current set U and the newly chosen edge $\{x, y\}$, the algorithm computes a minimum spanning tree, if it does not terminate before because of a violated uniqueness or additivity constraint. Due to Theorem 11.2, the computed spanning tree is a compact additive tree for A and δ since the algorithm stops its computation as soon as the uniqueness constraint is violated.

Computing the distance graph in step 1 can be done in $O(n^2)$; the initialization in step 2 is obviously possible in linear time. The algorithm needs at most $n-1$ iterations of the while loop in step 3. The computations during one iteration can be done in $O(n \log n)$ time; this results in an overall running time in $O(n^2 \log n)$. □

11.3 Characters with Binary States

In the previous sections we have seen two examples for constructing a phylogenetic tree from pairwise distances of the given taxa. In the remainder of this chapter we present some approaches for constructing phylogenetic trees based on the knowledge of discrete characters of the taxa. We start in this section with a simple special case, where each character has exactly two different states. We denote these states by 0 and 1, and we additionally make the following assumptions.

- All characters are inherited independently from each other.
- The evolution of each character can only lead from state 0 to state 1; there is no development back from 1 to 0.
- The states of the characters for the ancestors of the given taxa in the phylogenetic tree are purely hypothetical and do not infer anything about the characters of the real ancestors. In other words, the method just serves for clustering the given taxa, and not for predicting the character states of the ancestors.

We start with a discussion of the biological relevance of the model. In every approach for phylogeny construction based on character data, one assumes that the same character states in different taxa imply a common ancestor. But this is not always the case in nature. Particularly, identical phenotypical character states[3] could also be due to convergence phenomena, i.e., to the independent development of similar characters, like the ability to fly in birds and

[3] *Phenotypical characters* are characters describing the outward appearance of an organism, whereas *genotypical characters* are characters that can be determined from the genes or the DNA sequences.

	character					
taxon	d_1	d_2	d_3	d_4	d_5	d_6
a_1	0	0	0	0	1	0
a_2	0	1	0	0	1	0
a_3	0	0	1	1	0	0
a_4	1	0	1	1	0	0
a_5	0	0	1	0	0	1

Table 11.1. An example of a binary character matrix

bats. Such misleading convergence effects can mostly be avoided by considering genotypical characters only. One frequently used method for extracting characters is the comparison of non-coding regions within the DNA sequence. Another type of genotypical character, giving very reliable results and even having binary states, is a statement about the regulation of protein expression of the following form: "The presence of protein A increases/decreases the expression of protein B."

The assumption that the states of a character can only evolve in one direction is justified by the fact that backward developments occur very rarely in nature.

In the following, based on the above assumptions, we present an efficient method for constructing a phylogenetic tree. To proceed, we first need the following formal definition of characters.

Definition 11.8. *Let $A = \{a_1, \ldots, a_n\}$ be a set of n taxa and let $D = \{d_1, \ldots, d_m\}$ be a set of m characters. An $(n \times m)$-matrix M, where $M(i, j) = \alpha$ holds if and only if the taxon a_i takes the state $\alpha \in \{0, 1\}$ for the character d_j for all $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$, is called* binary character matrix *for A and D.*

If a binary character takes the state 1 in a taxon, we also say that this taxon has this character. The following example illustrates this definition.

Example 11.3. Let $A = \{a_1, \ldots, a_5\}$ be a set of taxa and let $D = \{d_1, \ldots, d_6\}$ be a set of binary characters. A binary character matrix for A and D is shown in Table 11.1. $\diamondsuit$

We now construct a tree whose leaves correspond to the given taxa. The root of this tree is to represent a common ancestor of all given taxa that do not have any of the considered characters. The edges of the tree will be labeled with the characters in such a way that every taxon has exactly its characters (those in state 1) occurring as labels on the path from the root to the respective leaf. Moreover, every character will occur as a label of exactly one edge in the tree. We formalize this goal with the following definition.

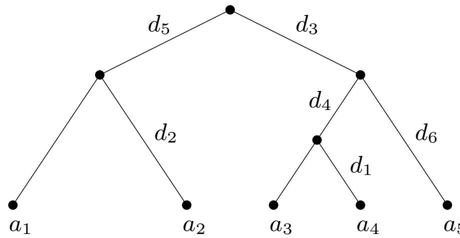


Fig. 11.5. A perfect phylogenetic tree for the character matrix from Table 11.1

Definition 11.9. *Let $A = \{a_1, \ldots, a_n\}$ be a set of taxa, let $D = \{d_1, \ldots, d_m\}$ be a set of binary characters, and let M be a binary character matrix for A and D. A* perfect phylogenetic tree *for M is a (partially) edge-labeled rooted tree $T = (V, E, d)$ with exactly n leaves satisfying the following conditions:*

(i) The leaves of T correspond to the taxa in A.
(ii) The edge labeling function $d : E \to D \cup \{\lambda\}$ assigns either the empty label λ or an element from D to each edge; moreover, each of the characters from D gets assigned to exactly one edge of T.
(iii) For each taxon a_i, the set of edge labels on the path from the root to the leaf a_i is exactly the set of characters of a_i.

We illustrate also this definition with an example.

Example 11.4. Figure 11.5 shows a perfect phylogenetic tree for the character matrix from Table 11.1. As we can see from this example, it is not necessary that each edge be labeled with a character. ◊

We now formally define the problem of determining a perfect phylogenetic tree.

Definition 11.10. *The problem of deciding the existence of a perfect phylogenetic tree, the* ExPerfPhyl *problem, is the following decision problem:*

Input: A set $A = \{a_1, \ldots, a_n\}$ of taxa, a set $D = \{d_1, \ldots, d_m\}$ of binary characters, and a binary character matrix M for A and D.
Output: Yes *if there exists a perfect phylogenetic tree for M,* No *otherwise.*

The problem of computing a perfect phylogenetic tree, the PerfPhyl *problem, is the following computing problem:*

Input: A set $A = \{a_1, \ldots, a_n\}$ of taxa, a set $D = \{d_1, \ldots, d_m\}$ of binary characters, and a binary character matrix M for A and D for which a perfect phylogenetic tree exists.
Output: A perfect phylogenetic tree T for M.

M'	character					
taxon	d_3	d_4	d_5	d_1	d_2	d_6
	c_1	c_2	c_3	c_4	c_5	c_6
a_1	0	0	1	0	0	0
a_2	0	0	1	0	1	0
a_3	1	1	0	0	0	0
a_4	1	1	0	1	0	0
a_5	1	0	0	0	0	1

Table 11.2. The column-wise sorted matrix M' from Example 11.5

We now proceed as follows. We first present an algorithm solving the PERFPHYL problem under an additional precondition regarding the form of the character matrix. After that we show that this additional condition exactly characterizes the set of character matrices that allow for a perfect phylogenetic tree. We then close this section with an algorithm testing a character metrix for the condition and thus solving the EXPERFPHYL problem.

To be able to formulate the condition, we first introduce the following notion for each character, referring to the set of taxa having the character.

Definition 11.11. *Let $A = \{a_1, \ldots, a_n\}$ be a set of taxa, let $D = \{d_1, \ldots, d_m\}$ be a set of binary characters, and let M be a binary character matrix for A and D. For all $1 \leqslant j \leqslant m$, let A_j denote the set of taxa having the character d_j.*

Two set A_i and A_j are called tree compatible*, if either they are disjoint or one is a subset of the other.*

We now present an algorithm constructing, for a set A of n taxa, a set D of m characters, and a binary character matrix M for A and D, a perfect phylogenetic tree under the condition that, for all $i, j \in \{1, \ldots, m\}$, the sets A_i and A_j are tree compatible. This algorithm is based on the following idea. It first sorts the columns of M by decreasing number of 1s. Then it iteratively constructs a tree whose vertices are labeled with subsets of A. It starts with the root of the tree and labels it with the entire set A. For each column j of M, the algorithm searches for the vertex labeled with the smallest superset of A_j and adds a new vertex labeled A_j as its child. In the last step, the algorithm adds all missing singleton sets from A as leaves at appropriate positions in the tree. The method is shown in Algorithm 11.3.

The following example illustrates the work of Algorithm 11.3.

Example 11.5. If the character matrix from Table 11.1 is sorted column-wise by decreasing number of 1s, this yields the matrix M' as shown in Table 11.2. From this, Algorithm 11.3 constructs a perfect phylogenetic tree as shown in Figure 11.6. The constructed tree corresponds to the perfect phylogenetic tree from Figure 11.5. ◊

Algorithm 11.3 Construction of a perfect phylogenetic tree

Input: A set $A = \{a_1, \ldots, a_n\}$ of taxa, a set $D = \{d_1, \ldots, d_m\}$ of binary characters, and a binary character matrix M for A and D, such that all sets A_i and A_j are pairwise different and tree compatible.

1. Sort the columns of M by decreasing number of ones.
2. Initialization:
   $V := \{A\}$
   $E := \emptyset$
3. **for** $j := 1$ **to** m **do**
   Search for the vertex $X \in V$ representing the smallest superset of A_j.
   $V := V \cup \{A_j\}$
   $E := E \cup \{(X, A_j)\}$
   $d(X, A_j) := d_j$
4. **for** $i := 1$ **to** n **do**
   if $\{a_i\} \notin V$ **then**
   Search for the vertex $X \in V$ representing the smallest set containing a_i.
   $V := V \cup \{\{a_i\}\}$
   $E := E \cup \{(X, \{a_i\})\}$
   $d(X, \{a_i\}) := \lambda$

Output: The perfect phylogenetic tree $T = (V, E, d)$ for M.

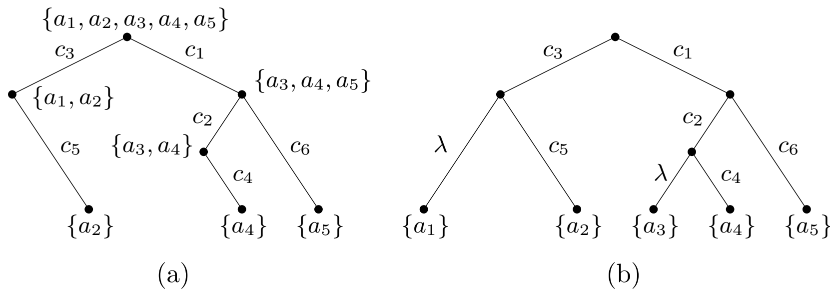


Fig. 11.6. The perfect phylogenetic tree constructed by Algorithm 11.3 in Example 11.5. (a) The tree after step 3; (b) the complete tree after step 4

We now prove the correctness of Algorithm 11.3 and analyze its running time.

Theorem 11.4. *Algorithm 11.3 solves the* PERFPHYL *problem for a set A of n taxa, a set D of m characters, and a character matrix M for A and D in time $O(n \cdot m^2)$ if the sets A_i and A_j are tree compatible for all $i, j \in \{1, \ldots, m\}$.*

Proof. We first show that Algorithm 11.3 constructs a perfect phylogenetic tree for M if the conditions of the theorem are satisfied. We assume without loss of generality that all columns of M are pairwise different. (We can delete identical columns without losing any information.)

According to the sorting of the columns of M and our additional condition for M, we can guarantee that for two sets A_i and A_j with $i < j$ either A_i and A_j are disjoint or $A_j \subsetneq A_i$ holds. This implies that the algorithm always considers the larger set first. Because of the precondition of the theorem, the smallest superset of the currently considered set A_j in step 3 is always uniquely determined, and because of sorting the columns in step 1, all occurring supersets of A_j are already present in the tree. We now have to show that the three properties of a perfect phylogenetic tree from Definition 11.9 are satisfied for the constructed tree T. Step 4 of the algorithm ensures that the leaves of T are exactly the singleton sets $\{a_i\}$ for $1 \leqslant i \leqslant n$. The edge label d_j is placed on exactly one edge by the algorithm for all $j \in \{1, \ldots, m\}$, i.e., on the newly constructed edge in the j-th iteration of step 3. Thus, T satisfies the first and the second property of a perfect phylogenetic tree. It remains to show that exactly those edge labels occur on the path from the root to the leaf $\{a_i\}$ that correspond to the characters of a_i. But this follows from the fact that the algorithm chooses the smallest superset of the current set A_j, and the sets A_j are sorted by decreasing size. Thus, the vertices on the path from the root to the leaf $\{a_i\}$ are exactly the sets A_j where $a_i \in A_j$.

We now analyze the time complexity of the algorithm. Counting the number of ones in the columns of M is obviously possible in $O(n \cdot m)$ time; the subsequent sorting can be done in $O(m \log m)$. Since there are at most $m = 2^n$ different characters, step 1 can be executed in $O(n \cdot m)$ time. The initialization in Step 2 is obviously possible in constant time. Searching for the vertex representing the smallest superset of the current set A_j in step 3 of the algorithm can be done in $O(n \cdot m)$ time, since A_j has to be compared to at most m other sets of size at most n. Step 3 hence needs overall time in $O(n \cdot m^2)$. Searching for the vertex representing the smallest set containing a_i in step 4 can be executed in time $O(m)$, since the test, whether or not a_i is contained in A_j, can be implemented in constant time by a table lookup. Thus, step 4 can be implemented in $O(n \cdot m)$ time, and Algorithm 11.3 needs time in $O(n \cdot m^2)$ overall. □

The following theorem shows that the additional precondition constitutes no essential restriction, since it exactly characterizes the character matrices for which a perfect phylogenetic tree exists.

Theorem 11.5. *Let $A = \{a_1, \ldots, a_n\}$ be a set of taxa, let $D = \{d_1, \ldots, d_m\}$ be a set of binary characters, and let M be a binary character matrix for A and D. Then there exists a perfect phylogenetic tree for M if and only if A_i and A_j are tree compatible for all $i, j \in \{1, \ldots, m\}$.*

Proof. It is a direct consequence of Theorem 11.4 that tree compatibility implies the existence of a perfect phylogenetic tree,

If on the other hand there is a perfect phylogenetic tree given for the character matrix M, it is easy to see that a set A_i corresponds to the set of leaves inside the subtree rooted at the edge labeled i. Since two such subtrees are either disjoint or included in each other, the claim follows. □

Theorem 11.5 implies that Algorithm 11.3 solves the PerfPhyl problem. There exists an even more efficient algorithm solving the PerfPhyl problem in $O(n \cdot m)$ time. We refer the reader to the bibliographic notes in Section 11.6.

After we have seen how to solve the PerfPhyl problem under the assumption that there exists a perfect phylogenetic tree for any input, the question remains open on how to efficiently check for the existence of such a tree.

The ExPerfPhyl problem could, for example, be solved by directly checking the tree compatibility condition from Theorem 11.5. Since there are m sets A_i, each possibly of size $O(n)$, and all pairs of these sets have to be considered, a naive implementation results in a running time in $O(n \cdot m^2)$ for the test. A more efficient algorithm for the ExPerfPhyl problem is based on the following idea: The algorithm sorts the columns of the binary character matrix M by decreasing number of 1s and calculates, for each 1 in M, the column number of the next 1 to the left of it. If there is a column in M containing two 1s with different such values, then there is no perfect phylogenetic tree for M, as we show in the following. This method is shown in Algorithm 11.4.

Theorem 11.6. *Algorithm 11.4 solves the* ExPerfPhyl *problem for a set A of n taxa, a set D of m characters, and a character matrix for A and D in $O(n \cdot m)$ time.*

Proof. We first prove the correctness of Algorithm 11.4. It suffices to show that the existence of a column in L containing two different values not equal to 0 implies the existence of two indices $j, j' \in \{1, \ldots, m\}$ such that the sets A_j and $A_{j'}$ are not tree compatible. Let j be a column index such that $L(i_1, j) = l_1 \neq l_2 = L(i_2, j)$ for some $i_1, i_2 \in \{1, \ldots, n\}$ and $l_1 < l_2$ and both $l_1, l_2 \neq 0$. According to the definition of L, this implies that $a_{i_1}, a_{i_2} \in A_j$, $a_{i_2} \in A_{l_2}$, and, because $l_1 < l_2$, also $a_{i_1} \notin A_{l_2}$ holds. Since the columns are sorted,

$$A_j \not\subseteq A_{l_2}. \tag{11.3}$$

From $a_{i_1} \in A_j$ and $a_{i_1} \notin A_{l_2}$ we know

$$A_{l_2} \not\subseteq A_j. \tag{11.4}$$

Since $a_{i_2} \in A_j$ and $a_{i_2} \in A_{l_2}$, the following holds:

$$A_j \cap A_{l_2} \neq \emptyset. \tag{11.5}$$

From (11.3), (11.4), and (11.5) we can infer that the sets A_j and A_{l_2} are not tree compatible.

Sorting the columns by decreasing number of 1s can be done in $O(n \cdot m)$, as shown in the proof of Theorem 11.4. The initialization of the auxiliary $(n \times m)$-matrix in step 2 is also possible in $O(n \cdot m)$ time. Furthermore, computing L in step 3 and checking L in step 4 are obviously also implementable in $O(n \cdot m)$ time, and hence the overall running time of Algorithm 11.4 is also in $O(n \cdot m)$. □

Algorithm 11.4 Existence of a perfect phylogenetic tree

Input: A set $A = \{a_1, \ldots, a_n\}$ of taxa, a set $D = \{d_1, \ldots, d_m\}$ of binary characters, and a binary character matrix M for A and D.

1. Sort the columns of M by decreasing number of 1s.
2. Initialize an auxiliary matrix L:

   for $i = 1$ **to** n **do**
       for $j = 1$ **to** m **do**
           $L(i, j) := 0$
3. Compute in L for each 1 in M the position of the rightmost one to its left (-1, if such a 1 does not exist).

   for $i = 1$ **to** n **do**
       $k := -1$
       for $j = 1$ **to** m **do**
           if $M(i, j) = 1$ **then**
               $L(i, j) := k$
               $k := j$
4. Check whether two different values $\neq 0$ occur in a column of L (if so, no perfect phylogenetic tree exists):

   $p :=$ true
   for $j = 1$ **to** m **do**
       $l := 0$
       for $i = 1$ **to** n **do**
           if $L(i, j) \neq 0$ **then**
               if $l = 0$ **then**
                   $l := L(i, j)$
               else if $l \neq L(i, j)$ **then**
                   $p :=$ false

Output: YES if $p =$ true, NO otherwise.

We illustrate the work of Algorithm 11.4 with an example.

Example 11.6. Sorting the character matrix from Table 11.1 column-wise by decreasing number of 1s yields the matrix M' shown in Table 11.2.

From this, Algorithm 11.4 constructs the matrix L as shown in Table 11.3. Since every column of L contains only one value not equal to 0, there exists a perfect phylogenetic tree for the given character matrix M. ◊

11.4 The Parsimony Principle and the Quartet Method

In this section, we discuss in greater detail the possibility of using DNA sequences of homologous genes as characters.

We assume that the taxa are given by a set of strings of identical length. The strings are to model a set of homologous genes; they can be determined,

L	character					
taxon	d_3	d_4	d_5	d_1	d_2	d_6
	c_1	c_2	c_3	c_4	c_5	c_6
a_1	0	0	-1	0	0	0
a_2	0	0	-1	0	3	0
a_3	-1	1	0	0	0	0
a_4	-1	1	0	2	0	0
a_5	-1	0	0	0	0	1

Table 11.3. The auxiliary matrix L from Example 11.6

for example, by a multiple alignment of the DNA sequences of the genes. We can then interpret the single columns of the alignment as the characters. A gap symbol occurring in the multiple alignment can be interpreted as an additional state of the characters, or we can restrict our attention to the columns of the alignment without gap symbols.

Our goal is to construct the topology of an unrooted binary phylogenetic tree for a given set of such taxa and characters, such that the leaves of the tree correspond to the given taxa. This means that we do not attempt to estimate the length of the tree edges, and hence a phylogenetic tree constructed in this way can only give us information about the relative degree of the kindred inside the given set of taxa.

The construction of such a phylogenetic tree is done in two steps. First, we present a method for measuring the costs of a given topology for a given set of taxa and characters. After that we discuss some ways for computing (at least approximately) the topology with minimal costs.

Measuring the costs of a given phylogenetic tree is based on the following idea. We are given a binary tree whose leaves are labeled with DNA sequences of length k. Now we search for a labeling of the inner vertices with DNA sequences of length k such that the following cost measure is minimized: For each edge $\{x, y\}$ in the tree, we count the number of substitutions that are necessary to transform the DNA sequence in x into the DNA sequence in y. Then we sum up the numbers over all edges to yield the overall cost of the tree. This method of calculating the costs of a phylogenetic tree is often called *parsimony principle* in the literature. We formally define this method as follows.

Definition 11.12. *Let $S = \{s_1, \ldots, s_n\}$ be a set of taxa. An* unrooted phylogenetic tree *for S is an unrooted binary tree*[4] *with exactly n leaves that are (one-to-one) labeled with the taxa from S.*

[4] Recall that an unrooted binary tree was defined in Definition 3.13 as a tree where each inner vertex has degree 3.

Definition 11.13. *The problem of measuring the cost of an unrooted phylogenetic tree according to the parsimony principle, the* parsimony problem, *is the following optimization problem:*

Input: A set $S = \{s_1, \ldots, s_n\}$ of strings of length k over an alphabet Σ and an unrooted phylogenetic tree $T = (V, E)$ for S.
Feasible solutions: For an input instance, every function $\beta : V \to \Sigma^k$ assigning strings from Σ^k to the vertices of T is a feasible solution if the leaves of the tree are mapped to the strings given in the input.
Costs: For a feasible solution β, the costs are defined as

$$cost(\beta) = \sum_{\{x,y\} \in E} dist_H(\beta(x), \beta(y)),$$

where $dist_H$ denotes the Hamming distance[5] *of two strings.*
Optimization goal: Minimization.

For an optimal solution β, we refer to $cost(\beta)$ also as to the *parsimony score* of T.

Next, we present a method for solving the parsimony problem based on the following idea. First, we insert an additional root at an arbitrary position in the tree by replacing an arbitrary edge with a root vertex with two incident edges. Then we traverse the tree bottom-up, starting at the leaves, and store a set of feasible labels for every vertex. Here, a string is considered to be a feasible label if it minimizes the Hamming distance between a vertex and its children. In an additional top-down traversal of the tree, we choose one of the feasible strings for each vertex. This method is shown in greater detail in Algorithm 11.5; it is called *Fitch algorithm*, after its inventor.

Before we prove the correctness of the Fitch algorithm and analyze its running time, we first illustrate its work with an example. Since the algorithm handles each position of the given strings separately, it is sufficient to consider strings of length 1 in the example.

Example 11.7. In Figure 11.7 (a), an unrooted phylogenetic tree is shown as an input for the parsimony problem, where the leaves are labeled with strings of length 1. From this, Algorithm 11.5 constructs a binary tree with a root r by expanding the edge drawn in bold in Figure 11.7 (a). The resulting tree is shown in Figure 11.7 (b). The R-sets, which are computed recursively in step 4 of the algorithm, are also shown. Moreover, all of those vertices are marked with $+1$, where the computation of the respective R-set leads to incrementing the counter *cost*. Figure 11.7 (c) shows a possible resulting tree after the traceback in step 5, where the edges with Hamming distance 1 are labeled 1. Finally, in Figure 11.7 (d) the result for the original unrooted tree is shown. ♢

[5] Recall that the Hamming distance of two strings was introduced in Definition 9.1 as the number of positions where the strings differ from each other.

Algorithm 11.5 Fitch algorithm for the parsimony problem

Input: A set S of n strings of length k over an alphabet Σ and an unrooted phylogenetic tree $T = (V, E)$ for S.

1. Construct a tree $T' = (V', E')$ with root $r \notin V$ by defining $V' := V \cup \{r\}$ and $E' := (E - \{\{x, y\}\}) \cup \{\{x, r\}, \{y, r\}\}$ for some arbitrary edge $\{x, y\} \in E$.
2. Initialize the set $R(x, l)$ of feasible symbols at position l of the label of x with $\emptyset$ for all $x \in V'$ and all positions $l \in \{1, \ldots, k\}$.
3. Initialize a counter c by 0, counting the number of necessary mismatches.
4. **for** $l := 1$ **to** k **do**
   Call the procedure *Fitch*(r, l).
5. Traceback:
   - Choose for the root r one of its children y and define, for all positions $l \in \{1, \ldots, k\}$,
   $$\beta(r, l) := a \text{ for some arbitrary } a \in R(y, l) \cap R(r, l).$$
   - Traverse T' top-down from the root to the leaves and define, for each vertex y with parent x and for each position $l \in \{1, \ldots, k\}$,
   $$\beta(y, l) := \begin{cases} \beta(x, l) & \text{if } \beta(x, l) \in R(y, l), \\ a \in R(y, l) \text{ arbitrary} & \text{otherwise.} \end{cases} \tag{11.6}$$
   - Define $\beta(x) := \beta(x, 1) \ldots \beta(x, k)$ for all $x \in V$ and $cost(\beta) := c$.

Output: The strings $\beta(x)$ for all vertices $x \in V$ and the costs $cost(\beta)$ of the label.

Procedure *Fitch*(x, l):

1. If x is a leaf, then determine the string $t = t_1 \ldots t_k$ with which x is labeled and define $R(x, l) := \{t_l\}$.
2. If x is an inner vertex, do the following:
   - Determine the two children y and z of x and call the procedures *Fitch*(y, l) and *Fitch*(z, l) to determine the sets $R(y, l)$ and $R(z, l)$.
   - If $R(y, l) \cap R(z, l) \neq \emptyset$ holds, then there is no mismatch necessary on the edges from x to y and to z at position l; thus, define $R(x, l) := R(y, l) \cap R(z, l)$.
   - If $R(y, l) \cap R(z, l) = \emptyset$ holds, then a mismatch is necessary; increment the counter *cost* by 1 and define $R(x, l) := R(y, l) \cup R(z, l)$; this ensures that a mismatch at position l will occur on only one of the two edges $\{x, y\}$ and $\{x, z\}$.

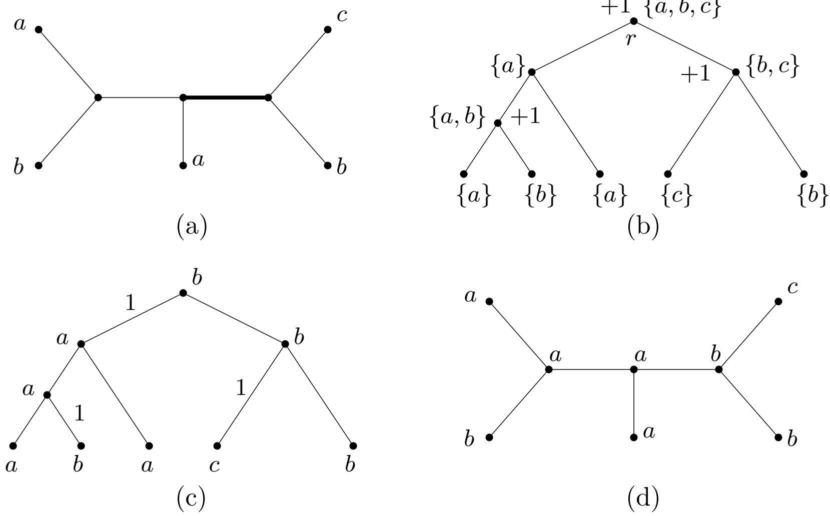


Fig. 11.7. An example for the work of the Fitch algorithm

As the following theorem shows, the Fitch algorithm gives us an efficient method for solving the parsimony problem.

Theorem 11.7. *Algorithm 11.5 solves the parsimony problem in $O(n \cdot k)$ time.*

Proof. We first prove the correctness of Algorithm 11.5. We note that the algorithm treats the positions of the given strings separately, i.e., it is sufficient to analyze the behavior of the algorithm for a fixed position l only.

To all leaves of the tree, the procedure *Fitch* assigns only the symbol that is already given in the input as a feasible label. Since the traceback in step 5 chooses only symbols from the R-set, as computed by the *Fitch* procedure, as final labels, a feasible function β is calculated for all leaves of the tree; it remains for us to show that β leads to a minimum number of mismatches. We now consider an inner vertex x with children y and z. Whenever it is possible that the labels of y and z agree at position l, the procedure *Fitch* chooses exactly these feasible agreeing symbols as feasible labels for x, by defining $R(x, l) = R(y, l) \cap R(z, l)$. If this is not possible, the comparison of the labels of x and its children enforces a mismatch at position l. The procedure then assures that such a mismatch only occurs in comparison with one of the children by defining $R(x, l) = R(y, l) \cup R(z, l)$. At the same time, it increments the counter *cost* for the number of mismatches by 1. This guarantees that the second case of Equation (11.6), where the symbol chosen does not agree with the one already chosen for the parent vertex, can occur at most once in the traceback for computing the β values for y and z. Hence, the algorithm constructs a function β with a minimum number of mismatches for the extended tree T'.

It remains for us to show that this result can be carried over to the original tree t without the root r. This follows from the fact that, due to the first substep of step 5, the root always gets the same label as one of its children. The root can hence be deleted without changing the number of mismatches. Thus, Algorithm 11.5 solves the parsimony problem.

For analyzing the running time of the algorithm, we treat $|\Sigma|$ as a constant, and thus the R-sets are of constant size as well. Since every vertex is considered exactly once for each position in the recursive calls of the procedure *Fitch* as well as in the traceback step, and since the running time is constant in each of these cases, the algorithm has an overall running time in $O(n \cdot k)$. □

With the Fitch algorithm we have seen a method for determining the cost of a given topology. But our goal is to find the topology with minimum cost. We first define this problem formally.

Definition 11.14. *The problem of finding an unrooted phylogenetic tree minimizing the parsimony score for a given set of strings, the* MinParTop problem*, is the following optimization problem:*

Input: A set S of n strings of length k.
Feasible solutions: Every unrooted phylogenetic tree for S.
Costs: The cost of a feasible solution is the optimal parsimony score of the tree.
Optimization goal: Minimization.

A naive approach to solving the MinParTop problem could be to examine all possible unrooted phylogenetic trees with n leaves, and to choose the one with the minimum parsimony score. But this is not possible efficiently, since there are exponentially many such binary trees, as shown by the following theorem.

Theorem 11.8. *For all $n \geqslant 3$, the number of non-isomorphic unrooted phylogenetic trees for n taxa is*

$$\prod_{i=3}^{n}(2i-5) = \frac{(2n-4)!}{2^{n-2} \cdot (n-2)!}.$$

Proof idea. We first note that an unrooted binary tree with n leaves has exactly $2n-3$ edges. This can be easily shown using induction on n. For $n = 3$ there is exactly one such binary tree. Now, every binary tree with n leaves can be constructed from a binary tree with $n-1$ leaves by dividing an arbitrary edge by an additional vertex and appending the new leaf to this new inner vertex. There are $2(n-1)-3 = 2n-5$ ways of doing this. Using a simple inductive argument, the claim follows. The construction used is shown in Figure 11.8 for $n = 4$. □

Unfortunately, there is no efficient algorithm for the MinParTop problem known. Hence, one mostly uses heuristic methods in practice. Besides general

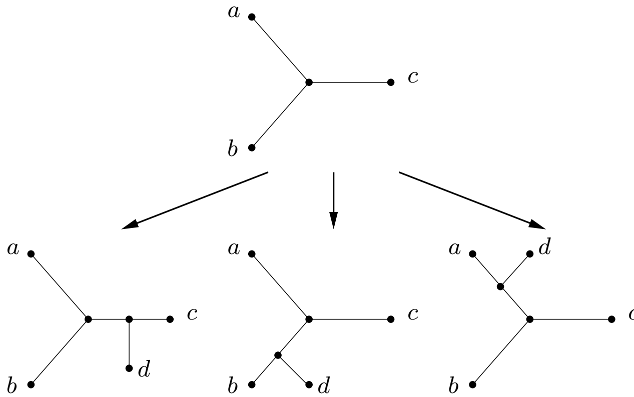


Fig. 11.8. The construction of all possible unrooted phylogenetic trees for four taxa from the unique unrooted phylogenetic tree for three taxa

heuristics like local search or simulated annealing, which can be used for many different problems,[6] some more specific heuristics have been developed. We examine one of them, successfully used in practice.

This heuristic is based on the idea of assembling a phylogenetic tree for a set S of n taxa from phylogenetic trees for subsets of S. Usually, one uses subsets of size 4; therefore, this approach is called *quartet method.* To describe the method in greater detail, we first need the formal definition of a quartet.

Definition 11.15. *A* quartet *is an unrooted phylogenetic tree for four taxa. For a set $S = \{a, b, c, d\}$ there are exactly three different quartets; they correspond to the three trees with four leaves each as shown in Figure 11.8. We denote these quartets by $(a, b; c, d)$, $(a, c; b, d)$, and $(a, d; b, c)$, where the first two and the last two entries denote the pairs of leaves having a common parent vertex.*

An optimal quartet *for S is a quartet with the minimum parsimony score.*

We now give a definition allowing us to compare a quartet to a larger unrooted phylogenetic tree.

Definition 11.16. *Let S be a set of n taxa, and let T be an unrooted phylogenetic tree for S. Let $S' = \{a, b, c, d\}$ be a subset of S, and let $Q = (a, b; c, d)$ be a quartet for S'. Let P_1 be the path from a to b in T, and let P_2 be the path from c to d in T. Then Q is said to be* consistent *with T if the two paths P_1 and P_2 are disjoint.*

Looking at quartets for finding a reasonably good solution to the MinParTop problem is motivated by the following result.

[6] For an overview of these general heuristics, see, for example, the book by Hromkovič [105].

Theorem 11.9. *Let S be a set of taxa, and let T be an unrooted phylogenetic tree for S. Let $\mathcal{Q}_T$ be the set of all quartets consistent with T. Then T can be uniquely reconstructed from $\mathcal{Q}_T$ in polynomial time.* □

Theorem 11.9 was proved by Buneman [38]; we skip the proof here.

This theorem implies that, if we succeed in finding all consistent quartets of the phylogenetic tree wanted, we are able to uniquely reconstruct the tree. But since we do not know any method for this, we try to approximate the set of all consistent quartets by computing the quartet with the optimal parsimony score for all subsets of four taxa. This computation can obviously be done in polynomial time, since for a set S of n taxa there are $\binom{n}{4} \in O(n^4)$ subsets of four elements, and for each of these subsets we just have to compute the parsimony score of three binary trees. This is possible in time proportional to the length of the taxa strings using Algorithm 11.5. Starting with the quartets, we then try to find an unrooted phylogenetic tree T such that a maximum number of quartets is consistent with T. This problem can be formalized as follows.

Definition 11.17. *The problem of finding an unrooted phylogenetic tree with which a maximum number of quartets is consistent, the* MaxQuartettConsist problem*, is the following optimization problem:*

Input: A set S of n taxa.
Feasible solutions: Every unrooted phylogenetic tree T for S.
Costs: For such a binary tree T, the costs are given by the number of all optimal quartets for subsets of S that are consistent with T.
Optimization goal: Maximization.

Unfortunately, even the MaxQuartettConsist problem is a hard optimization problem, as shown by the following theorem.

Theorem 11.10. *The* MaxQuartettConsist *problem is NP-hard.* □

Theorem 11.10 was proved by Berry et al. [32], we do not present the proof here. On the other hand, Jiang et al. [109] have shown that the optimal solution to the MaxQuartettConsist problem can be approximated arbitrarily well, but with a running time that is exponential in the reciprocal value of the desired approximation ratio.[7] Since the algorithm is technically very involved, we do not present it here. Instead, we investigate a heuristic for the MaxQuartettConsist problem, for which no proof of an approximation ratio is known, but which is frequently used in practice.

The heuristic is called *quartet puzzling.* Its idea is based on choosing a random order of the taxa, starting with the optimal quartet for the first four

[7] This means that the algorithm is, like Algorithm 9.1 from Section 9.1 for the consensus string approximation, an example of a polynomial-time approximation scheme.

Algorithm 11.6 Quartet puzzling

Input: A set S of n taxa.

1. Compute the optimal quartet $Q(S')$ for each four element subset $S' \subseteq S$ (according to the parsimony score).
2. Choose an order $a_1, \ldots, a_n$ of the elements in S uniformly at random.
3. Define $T := Q(\{a_1, a_2, a_3, a_4\})$.
4. **for** $i := 5$ **to** n **do**
   - Initialize the costs of all edges in T with 0.
   - For all $S' = \{b_1, b_2, b_3\} \subseteq \{a_1, \ldots, a_{i-1}\}$ such that $Q(\{b_1, b_2, b_3, a_i\})$ is of the form $(b_1, b_2; b_3, a_i)$, increase the edge costs on the path from b_1 to b_2 in T by 1 each.
   - Choose an edge $\{x, y\}$ in T with minimum cost, delete it, and insert a new vertex that is adjacent to x and y and the new leaf a_i.

Output: The unrooted phylogenetic tree T for S.

---

taxa according to this order, and then inserting one taxon after another into the tree, where the optimal quartets are used to find a good position for insertion. This method is presented in greater detail in Algorithm 11.6.

Often, steps 2 to 4 of Algorithm 11.6 are executed several times, and some kind of consensus of the computed solutions is constructed. This method is based on the idea that edges occurring in many of the computed trees describe a biologically meaningful connection with high probability. Computing the edge costs in step 4 is based on the following idea: The edges, where inserting the current taxon would lead many quartets to be inconsistent with the constructed tree, get a bad value. We now illustrate the work of Algorithm 11.6 with an example.

Example 11.8. Let $S = \{a, b, c, d, e\}$ be a set of taxa and let $(a, b; c, d)$, $(a, b; c, e)$, $(a, d; b, e)$, $(a, c; d, e)$, and $(b, d; c, e)$ be the optimal quartets for all four element subsets of S. Let a, b, c, d, e be the order chosen in step 2 of Algorithm 11.6. Then, the algorithm starts with the quartet $(a, b; c, d)$ and computes the edge costs from the other quartets of the tree and, eventually, an unrooted phylogenetic tree for the entire set S, as shown in Figure 11.9. $\diamondsuit$

Finally, we note that the quartet method can also be naturally used for other scoring functions than the parsimony score, since the parsimony score is only used in step 1 of Algorithm 11.6, and the rest of the algorithm is independent of the actual computation of the optimal quartets.

11.5 Summary

A tree describing kindred relations between different biological objects like species and homologous genes, the taxa, is called a phylogenetic tree. There

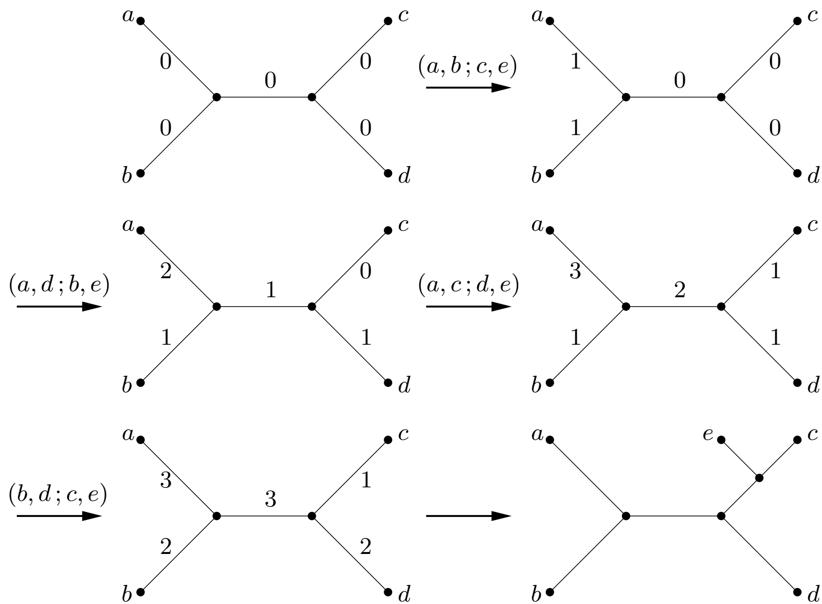


Fig. 11.9. An example for the work of Algorithm 11.6

are many different models of phylogenetic trees. In some models, the edges of the tree are labeled with distances between the taxa; other models simply try to describe the branching structure of the phylogenetic tree. Moreover, a phylogenetic tree can have a distinguished root vertex indicating the direction of evolution, or it can be unrooted, describing the relative degree of the kindred between the species. Computing a phylogenetic tree can be based on different kinds of information; one mainly distinguishes between distance information and discrete character state information.

If the distances in a given set of taxa are ultrametric, i.e., if for any three taxa two of the distances are identical and larger than the third, the (binary) phylogenetic tree is unique, and it is possible to compute it efficiently. A less stringent restriction on the given distance measure is additivity. If a distance measure is additive, there exists a phylogenetic tree whose edge labels exactly represent the distances. In contrast to ultrametric trees, the taxa are mapped not only to the leaves of the tree, but also to the inner vertices. The additivity constraint is efficiently verifiable, and computing a phylogenetic tree from additive distances can be done efficiently.

If a set of discrete characters is given as input instead of a distance measure, one usually tries to determine the topology of the phylogenetic tree without edge labeling. If the characters can only take one of two distinct states, and, additionally, the state of any character is assumed to change only in one direction, it is possible to efficiently determine whether a phylogenetic tree

consistent with the characters exists. If such a tree exists, it is also possible to compute it efficiently.

One can also consider DNA sequences directly as taxa; the characters are then the columns, without gaps, of a multiple alignment. Within this model, one can score a given phylogenetic tree for a given set of taxa according to the parsimony principle. For this approach, one also determines a DNA sequence for each inner vertex of the tree such that the number of mismatches along the edges of the tree is minimized. Computing the parsimony score can be done efficiently for a given tree, but there is no polynomial-time algorithm known for determining a tree with the minimum parsimony score. Hence, heuristic methods are often used to approximate a solution. One heuristic approach is to compute the phylogenetic tree for a given set of taxa from the optimal phylogenetic trees for all four element subsets of taxa. Finding an optimal solution for this quartet method is still NP-hard, but there are good approximation algorithms as well as practically successful heuristics known for this problem.

11.6 Bibliographic Notes

An introduction to the field of phylogenetic trees is given in the books by Gusfield [91], Setubal and Meidanis [180], Clote and Backofen [49], Ewens and Grant [68], and Durbin et al. [62]. In the last three of these books, also an overview of the numerous statistical approaches for determining phylogenies, which we omitted in our presentation, is given. As introductory literature we also want to mention the survey papers by Swofford and Olsen [189] and Felsenstein [70, 71]. The mathematical background of phylogenetic trees is presented in the book by Semple and Steel [179] from a non-algorithmic perspective.

The computation of phylogenetic trees from ultrametric distances is described in the books by Clote and Backofen [49], Ewens and Grant [68], and Gusfield [91]. The latter book also contains the proof of uniqueness. The UPGMA algorithm is due to Sokal and Michener [185]; an improved algorithm was presented by Fitch and Margoliash [75]. In the book by Gusfield [91], another algorithm for determining ultrametric trees is described that runs in time $O(n^2)$ for n taxa. Since the distances occurring in practice are not ultrametric in most cases, Farach et al. [69] have developed a method for finding an ultrametric tree under the assumption that, instead of exact distances, an interval is given for each pair of taxa, within which the distance of the taxa has to lie. This so-called sandwich approach is also described in the book by Setubal and Meidanis [180].

Constructing phylogenies from additive distances is described in the books by Gusfield [91] and Setubal and Meidanis [180], as well as in a survey paper by Barthelemy und Guenoche [22]. The first solution to the problem is due to Buneman [38].

Gusfield [89] was the first to show how to construct a perfect phylogenetic tree from binary characters; his method was later simplified by Waterman [201]. It is also described in the books by Gusfield [91] and Setubal and Meidanis [180].

The parsimony principle is presented in the book by Durbin et al. [62]. The Fitch algorithm goes back to Fitch [74]. As early as in 1870, Schröder [178] counted the number of unrooted phylogenetic trees. The quartet method is presented in the book by Clote and Backofen [49], and in a comprehensive survey paper by Kearney [118]. The reconstruction of a phylogenetic tree from the set of its consistent quartets is due to Buneman [38]. The NP-hardness of determining an unrooted phylogenetic tree that is consistent with a maximum number of quartets was proved by Berry et al. [32]; the polynomial approximation scheme for this problem goes back to Jiang et al. [109], and the method of quartet puzzling was proposed by Strimmer and von Haeseler [186].

12

Haplotyping

Many organisms, including all vertebrates, have a diploid genome, i.e., their genome contains two copies of each chromosome. The two copies are called *haplotypes*; they are not identical, but differ slightly from each other. Normally, one of the copies is inherited from the father and the other one is inherited from the mother. Therefore, it is very helpful, for example, for investigating complex genetic diseases, to know both haplotypes; but unfortunately it is difficult to determine the haplotypes of a given organism experimentally. Sequencing methods as described in Chapters 6 to 8 only yield information about the *genotype*, which consists of some kind of consensus of the two haplotypes. This results from the fact that it is hard to extract exactly one copy of one chromosome from a cell for further investigation instead of extracting the entire DNA from the nucleus.

In this chapter, we present some computational methods for inferring the haplotypes from genotype data. There are two main approaches for reaching this goal, depending on the input data. In the first scenario, we are given the genotype information for a population of several related individuals, and we look for the set of haplotypes occurring within the population. We describe this approach in Section 12.1. The second approach, described in Section 12.2, tries to infer the haplotypes for a single individual from genotype data of sequenced DNA fragments.

The first of the above approaches especially relies on the assumption that one of the haplotypes as a whole is inherited from the father, and the other is inherited as a whole from the mother. But, in nature, recombination events might also occur, leading to the situation where a haplotype of the child is composed from parts of two parental haplotypes. Nevertheless, such recombination does not occur very frequently; thus, in a small population there most probably exist long blocks inside the chromosomes that were not divided by recombination events in any of the corresponding haplotypes.

As usual, we conclude this chapter with a summary, in Section 12.3, and some bibliographic notes, in Section 12.4.

12.1 Inferring Haplotypes from a Population

The goal of this section is to present some methods for inferring the haplotypes in a population of related individuals, given their genotypes. To develop a formal model, we have to make some assumptions on the given data.

The most important differences between the DNA sequences of different individuals of a species are the *single nucleotide polymorphisms*, or *SNPs* (pronounced "snips") for short. A SNP is a difference occurring at a single position of the DNA sequences, where each of two (or even more) nucleotides is present in a significant fraction of the population; usually, this fraction is assumed to be at least 10%. We call the positions in the sequences where SNPs occur *SNP sites*. SNPs with more than two different nucleotides are very rare, and we assume for the rest of this chapter that only two different nucleotides occur at each SNP site. In this context, we abstract from the concrete nucleotides and use the symbols 0 and 1 instead.

In the following, we will slightly abstract from biological reality by considering SNPs to be the only differences between the considered DNA sequences. We further assume that we are given a map of the genome containing all SNP positions. Such a map of SNP sites has already been compiled for the human genome.

Since we know the SNP positions in the DNA sequence and do not care about possible further differences between the sequences, we can restrict our attention to the sequence of SNP sites, without considering the rest of the genome. When considering a genotype, we can distinguish between two types of SNP sites: If the two corresponding haplotypes agree at the position, the site is called *homozygous*; if the haplotypes disagree, it is called *heterozygous*. We can now define our formal model of haplotypes and genotypes.

Definition 12.1. *For a given genome with m SNP sites, a* haplotype *is denoted by a vector $(h_1, \ldots, h_m) \in \{0,1\}^m$, where the entry h_i of this vector describes which of the two possible nucleotides the haplotype carries at SNP site i.*

A genotype *is denoted by a vector $(g_1, \ldots, g_m) \in \{0,1,2\}^m$, where g_i coincides with the value of the respective haplotypes if i is a homozygous site, and $g_i = 2$ if i is a heterozygous site.*

We say that a genotype $g \in \{0,1,2\}^m$ can be resolved *by a pair of haplotypes $h, h' \in \{0,1\}^m$ if $h_i = h'_i = g_i$ for all $1 \leqslant i \leqslant m$ such that $g_i \in \{0,1\}$, and $h_i \neq h'_i$ for all $1 \leqslant i \leqslant m$ such that $g_i = 2$. The unordered haplotype pair $\{h, h'\}$ is called a* resolving pair *for the genotype g.*

We illustrate this formal definition with a short example.

Example 12.1. Consider an individual X, whose genome contains five SNP sites, where the two copies of its chromosome, restricted to the SNP sites, are $x_1 = \texttt{CATAG}$ and $x_2 = \texttt{CAATC}$. These haplotypes can be formally represented by the vectors $h_1 = (1,0,0,0,0)$ and $h_2 = (1,0,1,1,1)$. The corresponding

genotype of X is $g = (1, 0, 2, 2, 2)$, since the first two SNP sites are homozygous while the last three sites are heterozygous.

Conversely, given the genotype g as above, $\{h_1, h_2\}$ is a resolving pair for g; but $\{h'_1, h'_2\}$ is also a resolving pair for g, where $h'_1 = (1, 0, 1, 0, 1)$ and $h'_2 = (1, 0, 0, 1, 0)$. $\diamondsuit$

Note that the mapping of the nucleotides to the symbols 0 and 1 can be made arbitrarily, as long as, for each SNP site, the two possible nucleotides are mapped onto different symbols. For example, it is possible to map an `A` to 1 at one site and to 0 at another site.

In the next lemma, we count the number of resolving pairs for a given genotype.

Lemma 12.1. *Let $g = (g_1, \ldots, g_m) \in \{0, 1, 2\}^m$ be a genotype with exactly l entries of value 2. Then there exist 2^l different resolving pairs for g.*

Proof. Each of the l heterozygous positions i, where $g_i = 2$, can be resolved independently in one of two possible ways. $\square$

We describe a set of genotypes or haplotypes by a matrix defined as follows.

Definition 12.2. *Let $S = \{s_1, \ldots, s_m\}$ be a set of SNPs, let $G = \{g_1, \ldots, g_n\}$ be a set of genotypes for S, and let $H = \{h_1, \ldots, h_k\}$ be a set of haplotypes for S. A* genotype matrix *for S and G is an $(n \times m)$-matrix M over $\{0, 1, 2\}$ such that $M(i, j) = g_i(j)$ for all $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$. A* haplotype matrix *for S and H is a $(k \times m)$-matrix M over $\{0, 1\}$ such that $M(i, j) = h_i(j)$ for all $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, m\}$.*

As mentioned above, our goal in this section is, given the genotypes of a population, to deduce the set of haplotypes that are present within it. Of course, we can easily find a resolving pair of haplotypes for any genotype independently. But this would lead to exponentially many feasible solutions, as shown by Lemma 12.1, in contrast to the one true, biological solution we want to find. In order to get biologically meaningful results, we have to add some requirements for a solution to be feasible. Several different approaches have been discussed in the literature; see Section 12.4 for an overview. The approach we follow in the remainder of this section is based on perfect phylogenetic trees.

Recall our definition of a perfect phylogenetic tree in Section 11.3. For a binary character matrix for k taxa and m characters, Definition 11.9 defines a perfect phylogenetic tree as a rooted tree with exactly k leaves labeled with the taxa, where the root is labeled with the all-zero vector, and each character appears as a label of exactly one edge. The label describes the unique point in the tree where the character changes from 0 to 1.

In our current scenario, the taxa correspond to the haplotypes and the characters correspond to the SNPs. But since the values 0 and 1 for the two occurring nucleotides at any SNP site were chosen arbitrarily, we cannot

guarantee that the common ancestor of all present haplotypes corresponds to the all-zero vector. Thus, we have to relax the definition of a perfect phylogeny to allow for an arbitrary root.

Definition 12.3. *Let $H = \{h_1, \ldots, h_k\}$ be a set of haplotypes, let $S = \{s_1, \ldots, s_m\}$ be a set of SNPs, and let M be a haplotype matrix for H and S. An* undirected perfect phylogenetic tree *for M is a (partially) edge-labeled tree $T = (V, E, d)$ with a root r and exactly k leaves satisfying the following conditions:*

(i) The leaves in T correspond to the haplotypes in H and are labeled with the respective vectors.
(ii) The root r is labeled with some binary vector $(r_1, \ldots, r_m)$ of length m.
(iii) The edge-labeling function $d\colon E \to S \cup \{\lambda\}$ assigns either the empty label λ or one of the elements from S to each edge such that each SNP from S gets assigned to exactly one edge in T.
(iv) For any haplotype h_i, the set of edge labels on the path from r to the leaf h_i equals the set of SNPs in which h_i and the label of r differ.

Our goal is now to resolve a given set of k genotypes into a set of at most $2k$ haplotypes for which a perfect phylogenetic tree exists. We can formally state the problem as follows.

Definition 12.4. *The* perfect phylogeny haplotyping problem*, or the PPH problem for short, is the following computing problem:*

Input: A set S of m SNPs, a set G of k genotypes, and a genotype matrix M_G for S and G.
Output: A set H of at most $2k$ haplotypes and a haplotype matrix M_H for S and H, such that H contains a resolving pair for each genotype $g \in G$ and there exists an undirected perfect phylogenetic tree T for H; or an error message if no such H exists.

In the remainder of this section, we show how to efficiently solve the PPH problem. We will start by characterizing the sets of haplotypes that allow for a perfect phylogenetic tree. We first need the following definition.

Definition 12.5. *A* complete pair matrix *is a (4×2)-matrix over $\{0,1\}$, containing all four possible rows $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$ (in arbitrary order).*

Any complete pair matrix satisfies the property described by the following lemma.

Lemma 12.2. *Let M be a complete pair matrix, and let M' be the (4×2)-matrix obtained by flipping the entries in one column of M, i.e., $M'(i,j) = 1 - M(i,j)$ for $i \in \{1,2,3,4\}$ an M' is also a complete pair matrix.*

Proof. We leave the very easy proof to the reader. □

In the following, we frequently analyze submatrices[1] formed by two columns of the original matrix. We will use the following notation.

Definition 12.6. *Let M be a (genotype or haplotype) $(n \times m)$-matrix. For any two columns $j_1, j_2 \in \{1, \ldots, m\}$, let $M^{[j_1,j_2]}$ denote the restriction of M to the two columns j_1 and j_2.*

Example 12.2. Consider the genotype matrix

$$M_G = \begin{pmatrix} 2 & 2 & 2 \\ 1 & 1 & 2 \\ 0 & 0 & 1 \\ 2 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

Then,

$$M_G^{[1,3]} = \begin{pmatrix} 2 & 2 \\ 1 & 2 \\ 0 & 1 \\ 2 & 1 \\ 1 & 0 \end{pmatrix}.$$

$\diamondsuit$

The next lemma characterizes the haplotype matrices for which an undirected perfect phylogenetic tree exists.

Lemma 12.3. *Let M be a haplotype matrix for k haplotypes and m SNPs. Then there exists an undirected perfect phylogenetic tree for M if and only if M does not contain a complete pair matrix as a submatrix.*

Proof. We show the two implications of the claim separately. First, we assume that there exists a complete pair matrix M' as a submatrix of M. It is easy to see that M' does not admit an undirected perfect phylogenetic tree. Intuitively, for any root labeling, it is not possible to place the four leaves inside a tree such that each of the two edge labels describing the change in the value in its column appears exactly once. More formally, one could perform a complete case analysis over all possible edge-labeled trees with four leaves and any root labeling, but we omit the details here. Thus, M' does not admit an undirected perfect phylogeny, and this obviously implies that M also does not.

For the other direction of the proof, let $T = (V, E, d)$ be an undirected perfect phylogenetic tree for M with root r labeled $(r_1, \ldots, r_m)$. We can construct a new matrix M' from M by exchanging 0s and 1s in all columns j where $r_j = 1$. Then, T with root label $(0, \ldots, 0)$ is an undirected phylogenetic tree for M'. Since the root label now is the all-zero vector, T can be seen as a

[1] Here, by submatrix we mean a matrix induced by any subset of rows and columns. Thus, a submatrix does not have to be contiguous.

perfect phylogeny in the sense of Definition 11.9, and we can apply Theorem 11.5 to it.

As in Section 11.3, let, for any column i of M', A_i denote the set of row indices where column i contains the value 1. From Theorem 11.5 we know that the sets A_i and A_j are either disjoint or one of the sets is contained in the other, for any two columns i and j. If the sets A_i and A_j are disjoint, then $M'^{[i,j]}$ does not contain the row $(1, 1)$. If $A_i \subseteq A_j$, then $M'^{[i,j]}$ does not contain the row $(1, 0)$; and analogously, if $A_i \supseteq A_j$, then $M'^{[i,j]}$ does not contain the row $(0, 1)$. Thus, M' does not contain a complete pair matrix. According to Lemma 12.2, this implies that M also does not contain a complete pair matrix. □

We use this characterization in the sequel to describe an efficient algorithm for the PPH problem. Let $S = \{s_1, \ldots, s_m\}$ be a set of m SNPs, let $G = \{g_1, \ldots, g_n\}$ be a set of genotypes, and let M_G be a genotype matrix for S and G. We describe an algorithm that constructs a $(2k \times m)$-haplotype matrix M_H resolving M_G if such a matrix exists, or outputs an error message otherwise. For any row i of M_G, M_H will contain two rows i and i' such that the corresponding haplotypes h_i and $h_{i'}$ resolve the genotype g_i. In the following, we assume that row i in M_G is resolved by the rows i and $i' = n+i$ in M_H.

Note that, for any i and j such that $M_G(i, j) \in \{0, 1\}$, the corresponding entries of M_H are unambiguously given as $M_H(i, j) = M_H(i', j) = M_G(i, j)$, i.e., the homozygous entries from the genotype matrix are copied into the respective positions of the haplotype matrix. Also, all rows i of M_G containing exactly one 2-entry can be resolved unambiguously (up to the order of the rows i and i' of M_H). But already for two 2-entries, there are two essentially different ways of resolving a row, as shown in the following example.

Example 12.3. Consider the row $(2, 2)$ in a genotype matrix containing only two columns. This row can be resolved to

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}.$$

◊

For any pair (j_1, j_2) of columns of M_G, it is easy to see whether the columns can be resolved such that $M_H^{[j_1,j_2]}$ does not contain a complete pair matrix. This is due to the fact that every row in $M_G^{[j_1,j_2]}$ containing exactly one 2-entry unambiguously determines two rows of $M_H^{[j_1,j_2]}$. Thus, the rows of $M_G^{[j_1,j_2]}$ different from $(2, 2)$ unambiguously define a set of rows of $M_H^{[j_1,j_2]}$. If this set does not already contain all four rows of a complete pair matrix, then it is also possible to resolve any $(2, 2)$-rows in $M_G^{[j_1,j_2]}$ without generating a complete pair matrix (see Example 12.3).

This observation motivates the following definition.

Definition 12.7. *Let M_G be a $(n \times m)$-genotype matrix. We call M_G* regular *if any column pair (j_1, j_2) can be resolved without generating a complete pair matrix.*

In the following, we restrict our attention to regular genotype matrices, and we distinguish two types of column pairs, depending on whether the restricted matrix contains the row $(2, 2)$ or not.

Definition 12.8. *Let j_1 and j_2 be two columns of M_G. We call j_1 and j_2* companion columns*, or* companions *for short, if there exists a row i such that $M_G(i, j_1) = M_G(i, j_2) = 2$. Such a row is called a* companion row *for j_1 and j_2.*

As we have seen in Example 12.3, there are two ways to resolve the entries $M_G(i, j_1)$ and $M_G(i, j_2)$ for any companion pair (j_1, j_2) with companion row i: Either both columns get the same value in the i-th and i'-th rows of M_H, or they get different values in both rows. This motivates the following definition.

Definition 12.9. *Let j_1 and j_2 be two companion columns of a genotype matrix M_G with companion row i, and let M_H be a haplotype matrix for M_G. If $M_H(i, j_1) = M_H(i, j_2)$ and $M_H(i', j_1) = M_H(i', j_2)$, we call j_1 and j_2* equated *with respect to i; if $M_H(i, j_1) \neq M_H(i, j_2)$ and $M_H(i', j_1) \neq M_H(i', j_2)$, we call them* negated *with respect to i.*

The following lemma shows that two columns cannot be equated with respect to one companion row and negated with respect to another.

Lemma 12.4. *Let j_1 and j_2 be two companion columns of M_G, and let i_1 and i_2 be two companion rows for j_1 and j_2. Then, in any PPH solution, j_1 and j_2 cannot be equated with respect to i_1 and negated with respect to i_2.*

Proof. Assume to the contrary that j_1 and j_2 are equated with respect to i_1 and negated with respect to i_2. Then the rows i_1 and i_1', restricted to j_1 and j_2, are $(0, 0)$ and $(1, 1)$, and the rows i_2 and i_2', restricted to j_1 and j_2, are $(0, 1)$ and $(1, 0)$. This implies that M_H contains a complete pair matrix, contradicting Lemma 12.3. □

Lemma 12.4 gives the idea for the following definition.

Definition 12.10. *Let j_1, j_2 be two companion columns of M_G. The haplotype matrix M_H* equates *j_1 and j_2 if, for any companion row i for j_1 and j_2, j_1 and j_2 are equated with respect to i. The haplotype matrix M_H* negates *j_1 and j_2 if, for any companion row i for j_1 and j_2, j_1 and j_2 are negated with respect to i.*

In some cases, the entries of M_G directly force any algorithm to equate (or negate) a given companion pair, as we show in the following example.

Example 12.4. Consider the genotype matrix

$$M_G = \begin{pmatrix} 2 & 2 \\ 2 & 1 \\ 0 & 2 \end{pmatrix}.$$

If the two columns of this matrix are negated, this leads to the haplotype matrix

$$M_H = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$$

containing a complete pair matrix. But equating the two columns leads to the haplotype matrix

$$M_H = \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{pmatrix},$$

which is a feasible solution to the PPH problem. ♢

We now formally characterize the patterns in M_G forcing two columns to be equated or negated.

Definition 12.11. *Let j_1 and j_2 be two companion columns of M_G. A pair (i_1, i_2) of non-companion rows for j_1 and j_2 is called a* forcing pair *for j_1 and j_2 if any possible way of resolving of i_1 and i_2 forces the companion pair (j_1, j_2) either to be equated or to be negated. The restriction of a forcing pair to the columns j_1 and j_2 is called a* forcing pattern*.*

Example 12.5. Consider again the genotype matrix

$$M_G = \begin{pmatrix} 2 & 2 & 2 \\ 1 & 1 & 2 \\ 0 & 0 & 1 \\ 2 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

from Example 12.2. Then, $\{2, 3\}$ is a simple forcing pair for the columns 1 and 2 of M_G, that forces them to be equated; the corresponding forcing pattern is $\{(0, 0), (1, 1)\}$. Similarly, $(3, 5)$ is a forcing pair for the columns 2 and 3 of M_G, forcing them to be negated; the corresponding forcing pattern is $\{(0, 1), (1, 0)\}$. ♢

Lemma 12.5. *Let j_1 and j_2 be two companion columns of M_G, and let (i_1, i_2) be a pair of non-companion rows for j_1, j_2 in $M_G^{[j_1,j_2]}$ such that $M_G(i_1, j_1) = M_G(i_2, j_2) = 2$. Then, (i_1, i_2) is a forcing pattern for j_1, j_2.*

Proof. Let $x = M_G(i_1, j_2)$ and $y = M_G(i_2, j_1)$. Since (i_1, i_2) are non-companion rows, we know that $x, y \in \{0, 1\}$, i.e., $M^{[j_1,j_2]}$ restricted to the rows i_1 and i_2 has the form

$$\begin{pmatrix} 2 & x \\ y & 2 \end{pmatrix}$$

for some $x, y \in \{0, 1, 2\}$. Thus, $M_H^{[j_1,j_2]}$ has to contain the rows $(x, 0)$, $(x, 1)$, $(0, y)$, and $(1, y)$. An easy case distinction based on the values of x and y shows that the columns j_1 and j_2 have to be equated if $x \neq y$, and they have to be negated if $x = y$. □

We can now combine the results from Example 12.5 and Lemma 12.5 to obtain the following characterization lemma.

Lemma 12.6. *Let j_1 and j_2 be two companion columns of M_G. Then*

(a) $\{(0,0),(1,1)\}$, $\{(0,2),(1,1)\}$, $\{(1,2),(0,0)\}$, $\{(2,0),(1,1)\}$, $\{(2,1),(0,0)\}$, $\{(0,2),(2,1)\}$, *and* $\{(1,2),(2,0)\}$ *are forcing patterns forcing the companion columns to be equated,*
(b) $\{(0,1),(1,0)\}$, $\{(0,2),(1,0)\}$, $\{(1,2),(0,1)\}$, $\{(2,0),(0,1)\}$, $\{(2,1),(1,0)\}$, $\{(0,2),(2,0)\}$, *and* $\{(1,2),(2,1)\}$ *are forcing patterns forcing the companion columns to be negated, and*
(c) there are no other forcing patterns.

Proof. We observe that, in order to be a forcing pattern, a pair of rows in $M_G^{[j_1,j_2]}$ has to be resolved into a set of rows in $M_H^{[j_1,j_2]}$, which includes either the set $E = \{(0,0),(1,1)\}$ or the set $N = \{(0,1),(1,0)\}$, but not both (otherwise, $M_H^{[j_1,j_2]}$ would contain a complete pair matrix, and M_G would not be regular). If $M_H^{[j_1,j_2]}$ includes the set E, this companion pair is forced to be equated; if $M_H^{[j_1,j_2]}$ includes the set N, the companion pair is forced to be negated. This, together with Lemma 12.5, immediately implies claims (a) and (b).

It is easy to see that a companion pair is not forced to be equated or negated if neither E nor N are included in $M_H^{[j_1,j_2]}$. An easy case analysis based on this observation proves claim (c); we leave the details to the reader. □

But not just the matrix entries inside a companion pair can force it to be equated or negated. As we will see, this information can be propagated from one companion pair to another. For convenience, we first introduce a function indicating which companion pairs are equated and which ones are negated.

Definition 12.12. *Let M_G be a regular genotype matrix, and let M_H be a haplotype matrix that is a feasible solution to the PPH problem with input M_G. Let $P \subseteq S \times S$ be the set of pairs of companion columns of M_G. Then the* indicator function *for M_H is a function $\mathcal{I}_{M_H}: P \to \{0,1\}$ such that $\mathcal{I}_{M_H}(j_1, j_2) = 0$ if j_1 and j_2 are equated in M_H, and $\mathcal{I}_{M_H}(j_1, j_2) = 1$ if j_1 and j_2 are negated in M_H.*

Whenever M_H is clear from the context, we write $\mathcal{I}$ instead of $\mathcal{I}_{M_H}$.

Lemma 12.7. *Let M_G be a regular genotype matrix, and let M_H be a feasible solution to the PPH problem. Let j, k, and l be three pairwise different columns of M_G such that there exists a row i where $M_G(i,j) = M_G(i,k) = M_G(i,l) = 2$. Then, $\mathcal{I}(j,k) = \mathcal{I}(j,l) \oplus \mathcal{I}(l,k)$, where $\oplus$ denotes the exclusive-or operator.*[2]

Proof. It is easy to observe that the indicator function $\mathcal{I}$ is defined in such a way that $\mathcal{I}(j_1, j_2) = M_H(i, j_1) \oplus M_H(i, j_2)$ holds for any two companion columns j_1 and j_2 with companion row i. Thus, we have

$$\begin{aligned}\mathcal{I}(j,l) \oplus \mathcal{I}(l,k) &= M_H(i,j) \oplus M_H(i,l) \oplus M_H(i,l) \oplus M_H(i,k)\\ &= M_H(i,j) \oplus M_H(i,k)\\ &= \mathcal{I}(j,k)\end{aligned}$$

$\square$

We now show that finding a feasible indicator function, i.e., equating or negating the companion pairs respecting the forcing patterns and the propagation rule from Lemma 12.7, is sufficient to construct a feasible haplotype matrix.

We start with the formal definition of a feasible indicator function.

Definition 12.13. *Let M_G be a regular genotype matrix, and let P be the set of companion pairs for M_G. An indicator function $\mathcal{I}: P \to \{0,1\}$ is called* feasible *for M_G if it satisfies the following conditions for all columns j, k, and l:*

(a) If (j,k) is a companion pair and there exists a forcing pattern forcing (j,k) to be equated, then $\mathcal{I}(j,k) = 0$.
(b) If (j,k) is a companion pair and there exists a forcing pattern forcing (j,k) to be negated, then $\mathcal{I}(j,k) = 1$.
(c) If there exists a row i such that $M_G(i,j) = M_G(i,k) = M_G(i,l) = 2$, then $\mathcal{I}(j,k) = \mathcal{I}(j,l) \oplus \mathcal{I}(l,k)$.

Lemma 12.8. *Let M_G be a regular genotype matrix, and let P be the set of companion pairs for M_G. Then there exists a feasible solution M_H to the PPH problem if and only if there exists a feasible indicator function $\mathcal{I}: P \to \{0,1\}$.*

---

[2] Recall that, for Boolean variables x and y, the exclusive-or operator is defined by $x \oplus y = 1$ if $x \neq y$, and $x \oplus y = 0$ if $x = y$.

Proof. For the first direction of the proof, let M_H be a feasible haplotype matrix for M_G. This matrix M_H defines an indicator function $\mathcal{I}_{M_H}$ as described in Definition 12.12. Since M_H is a feasible solution to the PPH problem, $\mathcal{I}_{M_H}$ has to obey all restrictions posed by conditions (a) and (b) of Definition 12.13 according to the definition of forcing patterns (otherwise, M_H would contain a complete pair matrix); furthermore, $\mathcal{I}_{M_H}$ also satisfies condition (c) of Definition 12.13 due to Lemma 12.7.

For the opposite direction, consider Algorithm 12.1. This algorithm constructs a haplotype matrix M_H from a given indicator function $\mathcal{I}$. We now show that the haplotype matrix constructed by Algorithm 12.1 is indeed a feasible solution to the PPH problem if the given indicator function satisfies conditions (a) to (c) of Definition 12.13. This means we have to show that the constructed haplotype matrix does not contain a complete pair submatrix.

We first show that all values $M_H(i,j)$ are independent of the choice of the companion column in the "else if" part of the algorithm, i.e., we want to show that, for all i and for all $j'_1 < j'_2 < j$ such that $M_G(i,j'_1) = M_G(i,j'_2) = M_G(i,j) = 2$, the following holds:

$$M_H(i,j'_1) \oplus \mathcal{I}(j,j'_1) = M_H(i,j'_2) \oplus \mathcal{I}(j,j'_2) \tag{12.1}$$

For the proof, we use induction on the number of possible choices for such a companion column. We look at one iteration of the algorithm considering the position (i,j) of the matrix. For the induction basis, if there is at most one index $j' < j$ such that i is a companion row for (j,j'), the claim is trivial for j. For the induction step, let $j'_1 \neq j'_2$ be two companion columns for j such that $j'_1 < j'_2 < j$ and $M_G(i,j'_1) = M_G(i,j'_2) = M_G(i,j) = 2$. According to our induction hypothesis, we know that $M_H(i,j'_2)$ is uniquely defined as

$$M_H(i,j'_2) = M_H(i,j'_1) \oplus \mathcal{I}(j'_1,j'_2).$$

Plugging in condition (c) from Definition 12.13, which says

$$\mathcal{I}(j'_1,j'_2) = \mathcal{I}(j'_1,j) \oplus \mathcal{I}(j'_2,j),$$

we get

$$M_H(i,j'_2) = M_H(i,j'_1) \oplus \mathcal{I}(j'_1,j) \oplus \mathcal{I}(j'_2,j),$$

which is obviously equivalent to Equation (12.1). We have now seen that the haplotype matrix M_H computed by Algorithm 12.1 is uniquely determined for any input genotype matrix M_G. Any restriction of M_H to a pair of non-companion columns does not contain a complete pair matrix since M_G is regular, and any restriction of M_H to a companion pair does not contain a complete pair matrix either, due to conditions (a) and (b) of Definition 12.13. Thus, M_H is a feasible solution to the PPH problem. □

According to Lemma 12.8, finding a feasible indicator function is sufficient for solving the PPH problem. In the following, we describe how to efficiently perform this task. Our approach is based on constructing a graph whose vertices are the columns of M_G and whose edges describe the companion pairs.

Algorithm 12.1 Constructing a haplotype matrix from an indicator function

Input: A regular $(n \times m)$-genotype matrix M_G with set P of companion pairs and a feasible indicator function $\mathcal{I}\colon P \to \{0,1\}$ for M_G.

```
for j := 1 to m do
  for i := 1 to n do
    if M_G(i,j) ∈ {0,1} then
      M_H(i,j) := M_G(i,j)
      M_H(i',j) := M_G(i,j)
    else if there exists some j' < j such that i is a companion row for (j,j')
    then
      M_H(i,j) := M_H(i,j') ⊕ I(j,j')
      M_H(i',j) := M_H(i',j') ⊕ I(j,j')
    else
      M_H(i,j) := 0
      M_H(i',j) := 1
```

Output: The $(2n \times m)$-haplotype matrix M_H.

Definition 12.14. *Let M_G be a regular genotype $(n \times m)$-matrix, and let P be the set of companion pairs for M_G. The* companion graph *for M_G is an edge-labeled (undirected) graph $C(M_G) = (V, E_P, l)$ where the vertex set $V = \{1, \ldots, m\}$ is the set of columns of M_G, the edge set E_P is defined by $\{x,y\} \in E_P$ if and only if $(x,y) \in P$ (and thus also $(y,x) \in P$), and $l\colon E_P \to \{\mathsf{F}, \mathsf{N}\}$ is an edge-labeling function such that $l(e) = \mathsf{F}$ if the corresponding companion pair has a forcing pattern, and $l(e) = \mathsf{N}$ otherwise. We call the edges of $C(M_G)$ that are labeled F* forced edges*, and the edges labeled N* non-forced edges*.*

The restriction of $C(M_G)$ to the forced edges (on the same vertex set V) is called forcing graph *for M_G, and it is denoted by $F(M_G)$.*

If M_G is clear from the context, we write C and F instead of $C(M_G)$ and $F(M_G)$.

We will illustrate this definition with the following example.

Example 12.6. Consider the following regular genotype matrix

$$M_G = \begin{pmatrix} 2 & 2 & 1 & 0 & 2 \\ 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 2 & 1 & 2 & 1 \end{pmatrix}.$$

The companion graph $C(M_G)$ for M_G is shown in Figure 12.1; the forcing graph $F(M_G)$ for M_G is the subgraph of $C(M_G)$ containing only the edges with label F. ◊

Obviously, the companion graph and forcing graph for a given regular genotype matrix can be easily constructed using the characterization of forcing

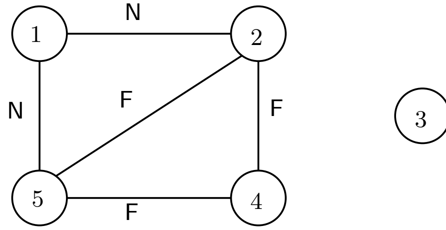


Fig. 12.1. The companion graph for the genotype matrix from Example 12.6

patterns from Lemma 12.6. We will show in the following how to infer a feasible indicator function for M_G from the companion graph, i.e., we will show how to define an indicator function $\mathcal{I}\colon E_P \to \{0,1\}$. The values of the indicator function can be easily set for any companion pair having a forcing pattern: If the forcing pattern forces the companion pair to be negated, then we set the value to 1; if it forces them to be equated, we set the value to 0.

It remains for us to show how to assign an indicator value to the edges of $C(M_G)$ labeled N. The algorithm for finding the value is based on the following idea: Consider an arbitrary triangle inside $C(M_G)$ where the indicator function has already been set for two of the three edges. Then the value of the third edge can be inferred as the exclusive-or value of these two values, following condition (c) from Definition 12.13. We show that we can solve our problem using this simple inference rule.

In the first step, we partition the companion graph with respect to the connected components of the forcing graph.

Definition 12.15. *Let $G = (V, E)$ be an (undirected) graph. The subgraph $H = (Z, E')$ induced by a subset $Z \subseteq V$ is called a* connected component *of G if any two vertices $z_1, z_2 \in Z$ are connected via a path in H, and if H is inclusion-maximal with this property.*

Every given graph can easily and very efficiently be partitioned into its connected components. For details, see the book by Cormen et al. [51].

Our algorithm computes the connected components of the forcing graph, and then considers the subgraphs of the companion graph induced by the vertex sets of the components.

Definition 12.16. *Let M_G be a genotype matrix, C be its companion graph, and F be its forcing graph. Let $F_1, \ldots, F_k$ be the connected components of F for some $k \in \mathbb{N}$. For any $i \in \{1, \ldots, k\}$, let C_i denote the subgraph of C induced by $V(F_i)$. We call the subgraphs $C_1, \ldots, C_k$ the* forcing components *of C.*

We will illustrate this definition with the following example.

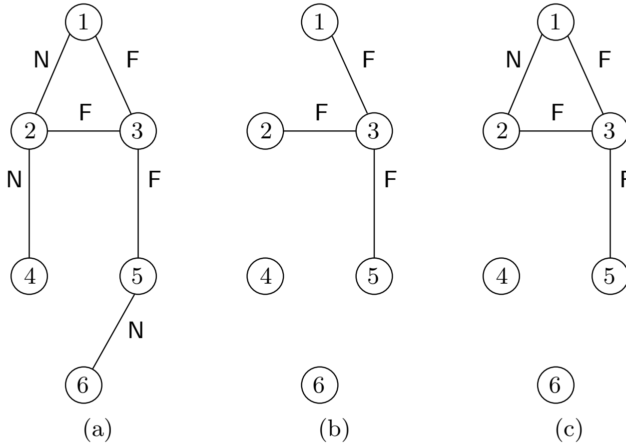


Fig. 12.2. The companion graph, forcing graph, and forcing components for the genotype matrix from Example 12.7

Example 12.7. Consider the following regular genotype matrix

$$M_G = \begin{pmatrix} 0 & 0 & 2 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 & 2 & 2 \\ 2 & 2 & 2 & 0 & 1 & 1 \\ 2 & 2 & 0 & 0 & 1 & 1 \\ 0 & 2 & 0 & 2 & 1 & 1 \end{pmatrix}.$$

The companion graph C for M_G is shown in Figure 12.2 (a), the forcing graph F for M_G is depicted in Figure 12.2 (b), and Figure 12.2 (c) shows the forcing components of C. $\diamondsuit$

In the following, we will see that the indicator function can be uniquely determined inside each forcing component using the inference rule based on condition (c) from Definition 12.13 as described above. The algorithm for computing the indicator function inside a forcing component of F is formalized in Algorithm 12.2.

To prove the correctness of Algorithm 12.2, we first need the following lemma.

Lemma 12.9. *Let M_G be a regular genotype matrix, and let C be its companion graph. Then, every cycle in C of length $\geqslant 4$ containing a non-forced edge e has a chord adjacent to e.*

Proof. Consider a non-forced edge $e = \{j_1, j_2\}$ inside a cycle H of length $\geqslant 4$ in C. There exist two vertices j_0 and j_3 such that j_0, j_1, j_2, and j_3 are pairwise distinct and the path $P = j_0, j_1, j_2, j_3$ is part of the cycle H.

Algorithm 12.2 Computing the indicator function inside a component

Input: A regular $(n \times m)$-genotype matrix M_G with set P of companion pairs, and a forcing component D in the companion graph C for M_G.

1. For all forced edges $e \in E(D) \cap E(F)$, set the value $\mathcal{I}(e)$ of the indicator function according to Lemma 12.6.
2. Let $E' := E(D) \cap E(F)$ be the set of edges for which the value of the indicator function is already set.
3. **while** $E' \neq E(D)$ **do**
   3.1. Find a triangle of edges e_1, e_2, e_3 in C such that $e_1, e_2 \in E'$ and $e_3 \notin E'$.
   3.2. Set $\mathcal{I}(e_3) := \mathcal{I}(e_1) \oplus \mathcal{I}(e_2)$.
   3.3. Set $E' := E' \cup \{e_3\}$.

Output: The indicator function $\mathcal{I}$ for the companion pairs inside D.

This implies that (j_0, j_1), (j_1, j_2), and (j_2, j_3) are companion pairs by the definition of the companion graph. Thus, there exist rows i_1, i_2, and i_3 in M_G such that $M_G(i_1, j_0) = M_G(i_1, j_1) = 2$, $M_G(i_2, j_1) = M_G(i_2, j_2) = 2$, and $M_G(i_3, j_2) = M_G(i_3, j_3) = 2$. The corresponding submatrix of M_G, under the assumption that i_1, i_2, i_3 are pairwise distinct, is

	j_0	j_1	j_2	j_3
i_1	2	2	y	
i_2		2	2	
i_3		x	2	2

We will now analyze the possible values for the entries $x = M_G(i_3, j_1)$ and $y = M_G(i_2, j_2)$. If both x and y are from the set $\{0, 1\}$, then $M_G^{[j_1, j_2]}$ contains a forcing pattern, contradicting our assumption that $\{j_1, j_2\}$ is a non-forced edge. Thus, at least one of x and y has to be a 2. (Note that it is immediately clear that x or y has to be 2 if i_1, i_2, and i_3 are not pairwise distinct.) If $x = 2$, then j_1 and j_3 are companions; if $y = 2$, then j_0 and j_2 are companions. Thus, at least one of the edges $\{j_0, j_2\}$ and $\{j_1, j_3\}$ is contained in $E(C)$, and so the cycle H has a chord. □

We are now ready to prove that Algorithm 12.2 outputs a feasible indicator function for a single forcing component, and that this indicator function is unique.

Lemma 12.10. *For a given genotype matrix M_G, which admits a feasible solution to the PPH problem, and a given forcing component D of its companion graph C, Algorithm 12.2 outputs the unique feasible indicator function for D.*

Proof. To prove the correctness of the algorithm, it suffices to show that the inference rule is always applicable, i.e., that a triangle of edges can always be found in step 3.1. Inside the forcing component D, the two endpoints of

any non-forced edge $e = \{x, y\}$ obviously have to be connected via a path in F, and thus via a path in $D' = (V(D), E')$, by the definition of a forcing component. Now, choose such a non-forced edge e from $E(D) - E'$, i.e., an edge that has not yet been assigned an indicator value, such that the shortest path in D' between its endpoints is of minimal length. If this path is of length 2, we have found an appropriate triangle (see Figure 12.3 (a)). Otherwise, e lies on a cycle of length at least 4 in D'. According to Lemma 12.9, this cycle has a chord e' adjacent to e. If this chord e' belongs to E', there is a shorter path in D' between the endpoints of e contradicting our assumption (see Figure 12.3 (b)); if e' belongs to $E(D) - E'$, there is a path in D' connecting the endpoints of e' that is shorter than the one connecting the endpoints of e, again a contradiction (see Figure 12.3 (c)). Thus, in every iteration of the while loop in step 3, an appropriate triangle can be found.

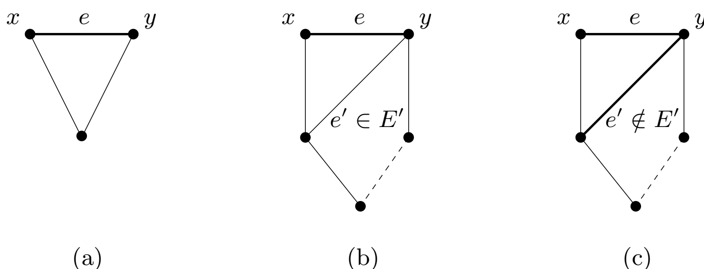


Fig. 12.3. Situations considered in the proof of Lemma 12.10. In (a) an appropriate triangle was found, while (b) and (c) illustrate that, otherwise, there is a contradiction in the choice of the edge e. Edges drawn as thin lines belong to E' and have thus been assigned an indicator value.

Now we prove that the indicator function found by Algorithm 12.2 is uniquely determined and does not depend on the order in which the edges are chosen. Since the input admits a solution to the PPH problem, there exists a feasible indicator function for M_G according to Lemma 12.8. Let $\mathcal{I}_0$ be such a feasible indicator function. We show that the indicator function $\mathcal{I}$ determined by the algorithm coincides with $\mathcal{I}_0$ on all edges inside D. It is clear that the two functions coincide on the forced edges. We proceed by induction on the number of edges that have received their indicator value by the algorithm. Consider the currently chosen triangle e_1, e_2, e_3. By the induction hypothesis, $\mathcal{I}(e_1) = \mathcal{I}_0(e_1)$ and $\mathcal{I}(e_2) = \mathcal{I}_0(e_2)$. By condition (c) of Definition 12.13, $\mathcal{I}_0(e_3) = \mathcal{I}_0(e_1) \oplus \mathcal{I}_0(e_2) = \mathcal{I}(e_1) \oplus \mathcal{I}(e_2)$, and this is exactly the value assigned to e_3 in Step 3.2 of the algorithm. □

We have now seen how to determine the values of the indicator function inside a forcing component. In the following, we describe how we can extend the indicator function across the forcing components. Obviously, every connected

Algorithm 12.3 Computing the indicator function

Input: A regular $(n \times m)$-genotype matrix M_G with set P of companion pairs, the (connected) companion graph C for M_G, and the forcing graph F for M_G.

1. Compute the set $\mathcal{D} = \{D_1, \ldots, D_k\}$ of forcing components of C.
2. Determine the component graph $\mathcal{G}$ for M_G.
3. Compute a spanning tree $\mathcal{T}$ of $\mathcal{G}$.
4. Let $\eta_1, \ldots, \eta_r$ be the edges of $\mathcal{T}$.
5. **for** $i = 1$ **to** r **do**
   5.1. Let e_i be some arbitrary edge from $E(\eta_i)$.
   5.2. Set $\mathcal{I}(e_i)$ arbitrarily.
   5.3. $E(F) := E(F) \cup \{e_i\}$.
6. Set the other indicator values using Algorithm 12.2.

Output: An indicator function $\mathcal{I}$ for M_G.

---

component of the companion graph can be handled independently. Thus, we assume in the following that C is connected.

The main idea is to connect the forcing components with each other via a minimum-cardinality set of (non-forced) edges. The edges from this set can be labeled with arbitrary indicator values, and afterwards we can view them as forced edges and apply Algorithm 12.2. To make this argument more formal, we define the following auxiliary graph whose vertices are the forcing components, and where two vertices are connected if the respective components are connected (by non-forced edges) in C.

Definition 12.17. *Let M_G be a regular genotype matrix with set P of companion pairs, let C be the (connected) companion graph for M_G, and let $\mathcal{D} = \{D_1, \ldots, D_k\}$ be the set of forcing components of C for some k.*

The component graph *for C is an undirected graph $\mathcal{G} = (\mathcal{D}, \mathcal{E})$ where $\mathcal{E} = \{\{D_i, D_j\} \mid D_i, D_j \in \mathcal{D}$ and there exist $x \in D_i$ and $y \in D_j$ such that $\{x, y\} \in E(C)\}$.*

For each edge $\eta = \{D_i, D_j\} \in \mathcal{E}$, denote by $E(\eta)$ the set of edges in C between D_i and D_j, i.e., let $E(\eta) = \{\{x, y\} \in E(C) \mid x \in D_i$ and $y \in D_j\}$.

Since we have assumed that C is connected, $\mathcal{G}$ is also connected. We will see in the following that the values of the indicator function can be arbitrarily chosen on a spanning tree of $\mathcal{G}$, and that this choice unambiguously determines the indicator values on all edges of C. So, our algorithm computes a spanning tree $\mathcal{T}$ on the component graph; chooses, for each edge of this tree, one corresponding edge of C; and assigns the values on these edges arbitrarily. After this, the edges can be handled as if they were additional forced edges (now their value is forced by the decision of the algorithm), and Algorithm 12.2 can be applied to determine the remaining values of the indicator function. We summarize this procedure in Algorithm 12.3.

To show the correctness of Algorithm 12.3, we start by proving that any choice of indicator values for the edges from $\mathcal{T}$ leads to a feasible indicator function.

Lemma 12.11. *Let M_G be a regular genotype matrix that admits a feasible solution to the PPH problem, let C be the connected companion graph for M_G, let $\mathcal{G}$ be the corresponding component graph, and let $\mathcal{T}$ be a spanning tree of $\mathcal{G}$. For each edge η_i of $\mathcal{T}$, let $e_i \in E(\eta_i)$ be the edge chosen in step 5.1 of Algorithm 12.3 for some $i \in \{1, \ldots, |E(\mathcal{T})|\}$. Then, there exist two feasible indicator functions $\mathcal{I}_1$ and $\mathcal{I}_2$ for M_G satisfying $\mathcal{I}_1(e_i) = 1$ and $\mathcal{I}_2(e_i) = 0$.*

Proof. Since M_G admits a solution to the PPH problem, there exists a feasible indicator function $\mathcal{I}_0$ for M_G. Let $\mathcal{I}_0(e_i) = a \in \{0, 1\}$. We show the existence of a feasible indicator function $\mathcal{I}$ satisfying $\mathcal{I}(e_i) = 1 - a$. We define $\mathcal{I}(e) = \mathcal{I}_0(e)$ for all $e \notin E(\eta_i)$ and $\mathcal{I}(e) = 1 - \mathcal{I}_0(e)$ for all $e \in E(\eta_i)$. To prove the feasibility of $\mathcal{I}$, we have to show that, for any triangle of edges e, e', e'' in C, the condition

$$\mathcal{I}(e) = \mathcal{I}(e') \oplus \mathcal{I}(e'') \tag{12.2}$$

holds. If none of these edges is contained in $E(\eta_i)$, Equation (12.2) immediately follows from the feasibility of $\mathcal{I}_0$. Since the edges in $E(\eta_i)$ connect two forcing components of C, it is not possible that exactly one or all three edges belong to $E(\eta_i)$. Let us assume now that exactly two edges of $\{e, e', e''\}$ belong to $E(\eta_i)$; without loss of generality, let these be the edges e' and e''. From the feasibility of $\mathcal{I}_0$, we know that $\mathcal{I}_0(e) = \mathcal{I}_0(e') \oplus \mathcal{I}_0(e'') = (1 - \mathcal{I}(e')) \oplus (1 - \mathcal{I}(e'')) = \mathcal{I}(e') \oplus \mathcal{I}(e'')$, which proves (12.2). □

From Lemma 12.11 we can now directly conclude the correctness of Algorithm 12.3.

Lemma 12.12. *Let M_G be a regular $(n \times m)$-genotype matrix that admits a solution to the PPH problem, with set P of companion pairs; let C and F be the (connected) companion graph and forcing graph for M_G, respectively. Then, Algorithm 12.3 computes a feasible indicator function $\mathcal{I}$ for M_G.*

Proof. The correctness of the algorithm follows directly from the correctness of Algorithm 12.2, as proved in Lemma 12.10, and from Lemma 12.11. □

Algorithm 12.1 and Algorithm 12.3 together imply a polynomial-time algorithm for constructing a haplotype matrix for a given $(n \times m)$-genotype matrix. This algorithm is summarized in Algorithm 12.4.

Theorem 12.1. *For any given genotype matrix M_G, Algorithm 12.4 solves the PPH problem in polynomial time.*

Proof. The correctness of Algorithm 12.4 directly follows from Lemma 12.12 and Lemma 12.8, and its running time is obviously polynomial. □

Algorithm 12.4 Solving the PPH problem

Input: A genotype matrix M_G.

1. Compute the companion graph C and the forcing graph F for M_G.
2. Compute the connected components $C_1, \ldots, C_l$ of C.
3. **for** $i := 1$ **to** l **do**
   3.1. Let $M_G^{(i)}$ denote the restriction of M_G to the rows corresponding to $V(C_i)$.
   3.2. Compute an indicator function $\mathcal{I}_i$ for $M_G^{(i)}$ using Algorithm 12.3.
   3.3. Construct a haplotype matrix $M_H^{(i)}$ from $\mathcal{I}_i$ using Algorithm 12.1.
4. Construct a haplotype matrix M_H for M_G by joining $M_H^{(1)}, \ldots, M_H^{(l)}$.
5. Check if M_H is a feasible solution, i.e., if it does not contain a complete pair matrix as a submatrix.

Output: The constructed haplotype matrix M_H if it is feasible; an error message indicating that there is no solution otherwise.

Using appropriate data structures, it can be shown that it is possible to implement this algorithm to run in $O(nm^2)$ time. From a biological point of view, it is interesting to be able to compute not only one feasible haplotype matrix, but the set of all feasible matrices. It has been shown that this can be done using the algorithm, i.e., every possible choice of indicator values for the edges connecting the forcing components yields one feasible haplotype matrix, and these are all the possible matrices, independent of the choice of the spanning tree. For details we refer the reader to the literature referenced in Section 12.4.

12.2 Haplotyping a Single Individual

We have seen in the previous section how to determine the set of haplotypes occurring in a population of several individuals. In this section, we present an approach for computing the haplotypes of a single individual. Of course, it is impossible to guess a meaningful pair of haplotypes if we are given only the genotype sequence of the individual. Instead, we try to infer the haplotypes from DNA fragments obtained by some shotgun sequencing experiment. Recall that the idea behind the shotgun sequencing approach is to cut the given DNA into short fragments that can be directly sequenced and to assemble them into one DNA sequence using combinatorial methods. The method was described in detail in Chapter 8. Although it is very hard to separate the two haplotypes when extracting the DNA from the organism, we may well assume that every single fragment is part of only one of the haplotypes. Thus, our goal is to partition the set of fragments into two sets according to the haplotype they were taken from.

To describe the approach more formally, we have to rely on the following assumptions: As in the previous section, we consider SNPs to be the only

differences between the two haplotypes. We assume that all SNP sites are known to us and that we can efficiently determine which SNPs are contained in each of the fragments. More formally, we assume that our input describes the values the fragments show at each SNP site.

Definition 12.18. *Let $F = \{f_1, \ldots, f_n\}$ denote a set of fragments, and let $S = \{s_1, \ldots, s_m\}$ denote a set of SNP sites. A* SNP matrix *for F and S is an $(n \times m)$-matrix M over $\{0, 1, \text{-}\}$, where $M(i, j) = \text{-}$ if the fragment f_i does not contain the SNP site s_j, and $M(i, j) = x \in \{0, 1\}$ states that fragment f_i takes the value x at SNP site s_j.*

We illustrate Definition 12.18 with the following example.

Example 12.8. Consider the set $F = \{f_1, \ldots, f_5\}$ of fragments and the set $S = \{s_1, \ldots, s_6\}$ of SNPs. A possible SNP matrix M for F and S is shown below:

	s_1	s_2	s_3	s_4	s_5	s_6
f_1	-	0	1	1	-	1
f_2	1	0	0	-	-	0
f_3	1	0	1	-	1	1
f_4	1	-	-	0	-	0
f_5	-	0	1	-	1	-

◊

As the first step, we consider the case where the input data does not contain any errors. In this case, we call the given SNP matrix *error-free*, and our task turns out to be quite easy. Obviously, two fragments have to be assigned to different haplotypes if there exists a SNP where they have different values from the set $\{0, 1\}$. We call such a situation a *fragment conflict*. We can construct a graph with the fragments as its vertices and the fragment conflicts as its edges. We make this formal in the following definition.

Definition 12.19. *Let M be an $(n \times m)$-SNP matrix for a set $F = \{f_1, \ldots, f_n\}$ of fragments and a set $S = \{s_1, \ldots, s_m\}$ of SNP sites. A* fragment conflict *in M is an (unordered) pair $\{f_{i_1}, f_{i_2}\}$ of fragments such that there exists a SNP site $s_j \in S$ with $\{M(i_1, j), M(i_2, j)\} = \{0, 1\}$.*

The fragment conflict graph *for M is an undirected graph $G_F = (F, E)$ whose vertices are the fragments from F, and whose edges are the fragment conflicts in M, i.e., $E = \{\{f_{i_1}, f_{i_2}\} \mid$ there exists an $s_j \in S$ with $\{M(i_1, j), M(i_2, j)\} = \{0, 1\}\}$.*

We will illustrate this definition with the following example.

Example 12.9. Consider the SNP matrix M from Example 12.8. The fragment conflict graph for M is shown in Figure 12.4. ◊

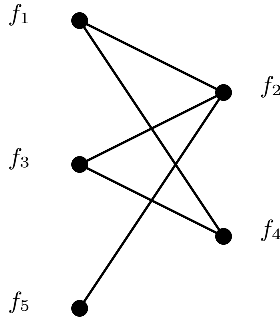


Fig. 12.4. The fragment conflict graph for the SNP matrix from Example 12.8

Thus, for error-free data, the genotype has to be resolvable into two haplotypes; we present a formal condition for this in the following. We start with a formal description of our haplotyping problem in the error-free case.

Definition 12.20. *The* error-free single individual haplotyping problem *is the following computing problem:*

Input: An error-free $(n \times m)$-SNP matrix M for a set $F = \{f_1, \dots, f_n\}$ of fragments and a set $S = \{s_1, \dots, s_m\}$ of SNP sites.
Output: A partition of F into two sets (haplotypes) H_1 and H_2, such that conflicts occur only between fragments from different haplotypes, i.e., for any conflict $\{f, f'\}$, $f \in H_1$ and $f' \in H_2$, or vice versa.

To solve this problem, we will use the following observation. We see that the conflict graph from Example 12.9 is bipartite. The next lemma states that this is the case for all fragment conflict graphs corresponding to error-free SNP matrices.

Lemma 12.13. *Let M be an error-free SNP matrix and let G_F be the fragment conflict graph for M. Then, G_F is bipartite.*

Proof. Assume to the contrary that G_F contains a cycle $C = f_1, \dots, f_{2i+1}, f_1$ of odd length. Obviously, if f_1 belongs to the first haplotype, then f_2 has to belong to the second haplotype, because $\{f_1, f_2\}$ is a conflict. Analogously, f_3 has to belong to the first haplotype, f_4 to the second, and so on. Finally, f_{2i+1} again has to be part of the first haplotype. But then, f_1 and f_{2i+1} belong to the same haplotype, a contradiction to our assumption that $\{f_{2i+1}, f_1\}$ is a fragment conflict. □

Although the fragment conflict graph might be bipartite also in the presence of errors, we expect most errors to destroy the bipartiteness of the graph. In the remainder of this section, we thus assume that a SNP matrix is error-free if and only if the corresponding fragment conflict graph is bipartite.

Algorithm 12.5 Solving the error-free single individual haplotyping problem

Input: An error-free $(n \times m)$-SNP matrix M for a set $F = \{f_1, \ldots, f_n\}$ of fragments and a set $S = \{s_1, \ldots, s_m\}$ of SNP sites.

1. Construct the fragment conflict graph G_F for M.
2. Compute a bipartition of G_F into the two vertex sets H_1 and H_2, using, for example, breadth-first search.

Output: The two haplotypes H_1 and H_2.

Definition 12.21. *Let M be an $(n \times m)$-SNP matrix. We call M* error-free *if its fragment conflict graph is bipartite.*

Lemma 12.13 now gives us a simple algorithm for the error-free single individual haplotyping problem: We just have to construct the fragment conflict graph, and to find a bipartition of it. This is summarized in Algorithm 12.5.

Theorem 12.2. *Algorithm 12.5 computes a solution to the error-free single individual haplotyping problem in $O(n^2 \cdot m)$ time.*

Proof. From the above discussion, it is obvious that the algorithm solves the problem. We now analyze its running time. The fragment conflict graph can be constructed by checking, for each SNP site and for each pair of fragments, whether the SNP induces a conflict on this pair of fragments. Thus, its construction takes time in $O(n^2 \cdot m)$. Computing the bipartition can be done using breadth-first search. This is possible in $O(n + k)$ time, where k is the number of fragment conflicts in M. For details see for instance the book by Cormen et al. [51]. Since $k < n^2$, the second step of the algorithm can be performed in $O(n^2)$ time. □

Please note that the solution to our problem is unique if and only if the fragment conflict graph is connected. Otherwise, the information gathered in the SNP matrix does clearly not suffice to completely determine the haplotypes.

Unfortunately, the data generated by biological experiments is rarely error-free. Thus we devote the remainder of this section to dealing with errors in the SNP matrix. As mentioned above, if there are false entries in the matrix, the fragment conflict graph will most likely become non-bipartite. So our goal will be to find a minimal set of changes which, applied to the SNP matrix, make the resulting fragment conflict graph bipartite again.

One can imagine several sources of errors occurring during the shotgun sequencing experiments. We will concentrate on two types of errors here: First, the examined DNA could be contaminated, leading to some fragments that do not belong to one of the two haplotypes we are looking for. Second, although one has a map of the SNP sites of the organism, it might contain errors; especially, it might propose SNPs at some sites where in reality there are none. These two types of errors lead to the following two optimization problems.

Definition 12.22. *The* minimum fragment removal problem*, the MFR problem for short, is the following optimization problem:*

Input: An $(n \times m)$-SNP matrix M for a set $F = \{f_1, \ldots, f_n\}$ of fragments and a set $S = \{s_1, \ldots, s_m\}$ of SNP sites.
Feasible solutions: All subsets $F' \subseteq F$ of fragments such that the SNP matrix for $F - F'$ and S is error-free.
Costs: The cardinality of the subset F'.
Optimization goal: Minimization.

The minimum SNP removal problem*, the MSR problem for short, is the following optimization problem:*

Input: An $(n \times m)$-SNP matrix M for a set $F = \{f_1, \ldots, f_n\}$ of fragments and a set $S = \{s_1, \ldots, s_m\}$ of SNP sites.
Feasible solutions: All subsets $S' \subseteq S$ of SNP sites such that the SNP matrix for F and $S - S'$ is error-free.
Costs: The cardinality of the subset S'.
Optimization goal: Minimization.

Unfortunately, the MFR and MSR problems are hard in their general formulation. We prove the NP-hardness of the MFR problem by a reduction from a graph problem. To do this, we first need to show that every undirected graph can be transformed into a SNP matrix.

Lemma 12.14. *Let $G = (V, E)$ be an undirected graph with n vertices and m edges. Then there exists an $(n \times m)$-SNP matrix such that G equals the fragment conflict graph G_F for M.*

Proof. For a graph $G = (V, E)$ with vertex set $V = \{v_1, \ldots, v_n\}$ and edge set $E = \{e_1, \ldots, e_m\}$, we associate a fragment with each of the vertices, i.e., we set $f_i = v_i$ for all $i \in \{1, \ldots, n\}$, and we associate a SNP with each edge of G, i.e., we set $s_j = e_j$ for all $j \in \{1, \ldots, m\}$. Now we can define the SNP matrix M as follows: For each edge $e_j = \{v_i, v_{i'}\}$, where $i < i'$, we set $M(i, j) = 0$, $M(i', j) = 1$, and $M(l, j) = -$ for all $l \in \{1, \ldots, n\} - \{i, i'\}$.

Then, obviously, every edge from G corresponds to an edge in G_F, and vice versa. Note that this transformation requires only polynomial time. □

The transformation from the proof of Lemma 12.14 is illustrated with the following example.

Example 12.10. Consider the graph G from Figure 12.5. We construct for it the following SNP matrix M:

	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_1	0	0	0	-	-	-	-
v_2	1	-	-	0	0	0	-
v_3	-	1	-	-	1	-	0
v_4	-	-	1	1	-	-	-
v_5	-	-	-	-	-	1	1

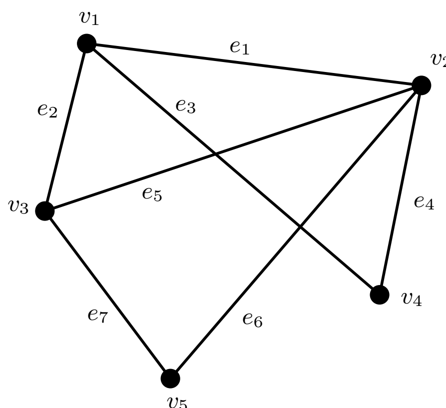


Fig. 12.5. The graph from Example 12.10

Then, obviously, $G = G_F$ holds. ◊

We are now ready to prove the NP-hardness of the MFR problem. We use a reduction from the following problem.

Definition 12.23. *The* minimum vertex bipartizer problem *is the following optimization problem:*

Input: An undirected graph $G = (V, E)$.
Feasible solutions: All subsets $B \subseteq V$ such that the subgraph of G induced by $V - B$ is bipartite.
Costs: The cost of a feasible solution is the cardinality of the set B.
Optimization goal: Minimization.

Lemma 12.15 ([208]). *The minimum vertex bipartizer problem is NP-hard.* □

Now we can easily show that the MFR problem is also hard.

Theorem 12.3. *The MFR problem is NP-hard.*

Proof. Consider a graph $G = (V, E)$ as an input for the minimum vertex bipartizer problem. We can construct a SNP matrix M from G using the construction from the proof of Lemma 12.14. Since each fragment in this matrix M corresponds to exactly one vertex of G, removing a minimum set of fragments from M is equivalent to removing a minimum set of vertices from G. Thus, Lemma 12.15 implies the NP-hardness of the MFR problem. □

Lancia et al. have shown that the MSR problem is NP-hard [125]; we will skip the proof here.

Theorem 12.4 ([125]). *The MSR problem is NP-hard.* □

In the following, we will consider a special case of the MFR and MSR problems that admits a polynomial-time solution. In the traditional shotgun approach for DNA sequencing, as described in Section 8.1, the fragments represent contiguous regions of the DNA, and thus also include a contiguous set of SNP sites. In terms of the matrix representation, this means that the non-gap entries in each row of the SNP matrix will be consecutive. For this special case, both the MFR and the MSR problems can be solved by polynomial-time dynamic programming algorithms. We focus on the algorithm for the MSR problem. The algorithm for the MFR problem is slightly more involved; we refer to the literature given in Section 12.4 for the details.

We first define the special case more formally.

Definition 12.24. *Let $S = \{s_1, \ldots, s_m\}$ be a set of SNPs, let $F = \{f_1, \ldots, f_n\}$ be a set of fragments, and let M be a SNP matrix for S and F. We say that M is* gapless *if, in each row of M, all non-gap entries appear consecutively, i.e., if for each fragment (row) f_i there exist two SNPs (columns) s_{l_i} and s_{r_i} such that $M(i,j) = \text{-}$ for all $j < l_i$ and for all $j > r_i$, and $M(i,j) \in \{0,1\}$ for all $l_i \leqslant j \leqslant r_i$.*

In the following, we restrict our attention to SNP matrices where every column contains at least one 1 and at least one 0. This is no real restriction, since a SNP site, in which all fragments agree, obviously does not help in distinguishing the two haplotypes.

Definition 12.25. *Let $S = \{s_1, \ldots, s_m\}$ be a set of SNPs, let $F = \{f_1, \ldots, f_n\}$ be a set of fragments, and let M be a SNP matrix for S and F. If, for any SNP s_i, there exist two fragments f_{j_i} and $f_{j'_i}$ such that $M(j_i, i) = 1$ and $M(j'_i, i) = 0$, we call M* reduced.

To describe the algorithm for the MSR problem on gapless SNP matrices, we first need the notion of conflicts between SNPs.

Definition 12.26. *Let $S = \{s_1, \ldots, s_m\}$ be a set of SNPs, let $F = \{f_1, \ldots, f_n\}$ be a set of fragments, and let M be a SNP matrix for S and F. Then, an (unordered) pair $\{s, s'\}$ of SNPs is called a* SNP conflict *if there exist two fragments f and f' such that*

- $M(f,s), M(f,s'), M(f',s), M(f',s') \in \{0,1\}$, *and*
- $M(f,s) = M(f,s')$ *if and only if* $M(f',s) \neq M(f',s')$.

We will also say that the SNPs s and s' are in conflict*.*

In other words, s and s' are in conflict, if the submatrix defined by s, s', f, and f' contains three 1-symbols and one 0-symbol, or vice versa.

Definition 12.26 also implies that, for a SNP matrix M and a SNP conflict $\{s, s'\}$ with the corresponding fragments f and f', $M(f,s) = M(f',s)$ if and only if $M(f,s') \neq M(f',s')$.

The following lemma states that a reduced gapless SNP matrix M cannot be error-free if it contains two SNPs that are in conflict.

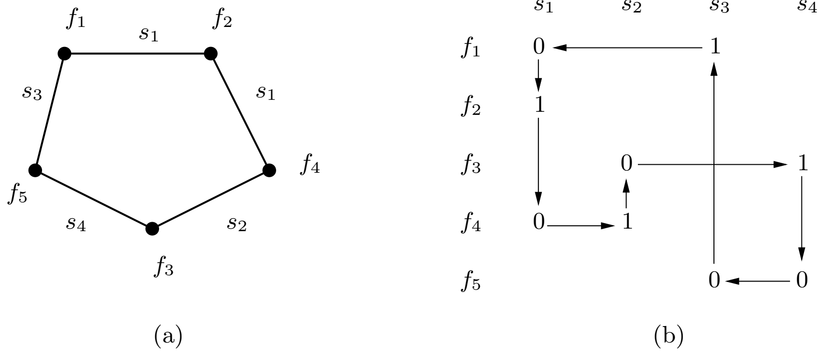


Fig. 12.6. An example for the fragment cycle in the proof of Lemma 12.16. (a) An odd cycle C in the fragment conflict graph; the edge labels indicate the SNP indices on which the respective fragments disagree; (b) the corresponding directed cycle D in the SNP matrix

Lemma 12.16. *Let M be a reduced gapless SNP matrix. Then M is error-free if and only if it does not contain two SNPs that are in conflict.*

Proof. We first show that a SNP matrix containing a SNP conflict cannot be error-free. Consider two SNPs s and s' that are in conflict, and let f and f' be two fragments responsible for that conflict. Then, without loss of generality, we may assume that $M(f,s) = M(f',s) = M(f,s') = 0$ and $M(f',s') = 1$. Since M is reduced, this implies that there exists a third fragment f'' such that $M(f'',s) = 1$. This leads to the following submatrix:

	s	s'
f	0	0
f'	0	1
f''	1	

Now we see that $\{f, f'\}$ is a fragment conflict due to s', and $\{f, f''\}$ and $\{f', f''\}$ are fragment conflicts due to s. Thus, there exists a triangle in G_F, i.e., G_F is not bipartite. According to Lemma 12.13, M is not error-free.

For the other direction, we again prove its negation, namely, that a non-error-free matrix implies a SNP conflict. To do so, we need some additional notation. Consider a cycle $C = f_1, \ldots, f_k, f_1$ in the fragment conflict graph. For simplicity of notation, we define $f_{k+1} = f_1$. For each edge $\{f_i, f_{i+1}\}$ in C, we find a SNP with index s_i such that $M(f_i, s_i) \neq M(f_{i+1}, s_i)$. The cycle C together with the SNPs defines a directed cycle D in the matrix M in the following sense: We consider the matrix entries as vertices; then, the cycle D is composed of horizontal arcs from $M(f_i, s_{i-1})$ to $M(f_i, s_i)$ (if $s_{i-1} \neq s_i$) and vertical arcs from $M(f_i, s_i)$ to $M(f_{i+1}, s_i)$. An example is shown in Figure 12.6. A maximal sequence of consecutive vertical arcs is

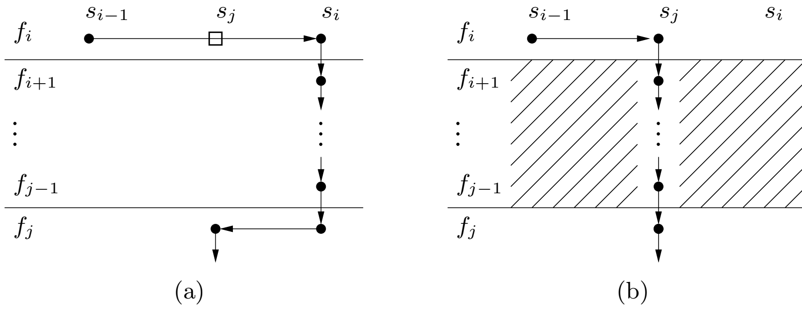


Fig. 12.7. The situation in the proof of Lemma 12.16. (a) A part of the assumed cycle D; (b) the respective part of the new cycle D' of lesser weight

called a *vertical run*. We call the minimum number of vertical runs in any cycle D in M corresponding to an odd cycle in G_F the *weight* of M.

Assume that there exists a matrix that is not error-free and does not contain any SNP conflicts. Choose such a matrix M of minimum weight. Consider a cycle D in M with the minimum number of vertical runs that corresponds to an odd cycle $C = f_1, \ldots, f_k, f_1$ in G_F. D has to contain at least two vertical runs, since otherwise D would be a cycle within only one column of M and would have the same length as C. But on such a cycle D, the values 0 and 1 would alternate, which contradicts the odd length of cycle C. The idea is to construct an alternative cycle D' together with a corresponding SNP matrix M' under the assumption that there are no SNP conflicts, such that D' in M' has a smaller number of vertical runs, but still corresponds to an odd cycle C' in G_F, hence contradicting our choice of M. We want to consider the rightmost vertical run of D. Since the order of the fragments in M does not affect the structure of the fragment conflict graph, we may assume without loss of generality, that the rightmost vertical run starts in fragment f_i, ends in f_j, and contains all fragments $f_{i+1}, \ldots, f_{j-1}$ as vertices for some $i < j$. This implies that $s_{i-1} \leqslant s_i$ and $s_j \leqslant s_i$.

Assume without loss of generality that $s_{i-1} \leqslant s_j$ holds; the proof for the other case is analogous. Then we have the situation shown in Figure 12.7 (a). We know that $M(f_i, s_j) = M(f_j, s_j)$ if and only if $M(f_i, s_i) = M(f_j, s_i)$, since otherwise M would contain a SNP conflict. Now we construct a new cycle D' from D as follows: First, we replace all entries in rows $f_{i+1}, \ldots, f_{j-1}$ by gaps, except for the entries in column s_j, which are set to yield a sequence of alternating zeros and ones. Then, we replace the horizontal arc from $M(f_i, s_{i-1})$ to $M(f_i, s_i)$ with the arc from $M(f_i, s_{i-1})$ to $M(f_i, s_j)$; we delete the horizontal arc from $M(f_j, s_i)$ to $M(f_j, s_j)$; and we replace the vertical run from $M(f_i, s_i)$ to $M(f_j, s_i)$ by the vertical run from $M(f_i, s_j)$ to $M(f_j, s_j)$. This operation defines a new matrix M' and leads to the new cycle D' shown in Figure 12.7 (b). Obviously, we have not induced any new SNP conflicts by this transformation, and the number of vertical runs in D' is decreased by

one with respect to the number of vertical runs in D, since the vertical run starting at $M(f_i, s_j)$ in D' is joined with the vertical run starting at $M(f_j, s_j)$. Furthermore, the cycle C' in the fragment conflict graph G'_F for M' is of odd length, since it contains exactly as many vertices as C. This gives us a contradiction to our assumption of M being of minimum weight. Thus, we have proved that any gapless matrix that is not error-free has to contain a SNP conflict, or, in other words, that no error-free gapless SNP matrix can contain a SNP conflict. □

Before we present an algorithm for the MSR problem, we first prove another property of SNP conflicts.

Lemma 12.17. *Let M be a gapless reduced SNP matrix with SNP set $S = \{s_1, \ldots, s_m\}$. Let $1 \leqslant i < j < k \leqslant m$. If $\{s_i, s_j\}$ and $\{s_j, s_k\}$ are no SNP conflicts, then $\{s_i, s_k\}$ is also not a SNP conflict.*

Proof. Assume to the contrary that s_i and s_k are in conflict. Then there exist two fragments f and f' such that $M(f, s_i), M(f, s_k), M(f', s_i), M(f', s_k) \in \{0, 1\}$ and that $M(f, s_i) = M(f', s_i)$ if and only if $M(f, s_k) \neq M(f', s_k)$. Assume without loss of generality that $M(f, s_i) = M(f', s_i)$, and thus $M(f, s_k) \neq M(f', s_k)$. Since $i < j < k$ and M is gapless, $M(f, s_j), M(f', s_j) \in \{0, 1\}$ holds. If $M(f, s_j) = M(f', s_j)$, then s_j and s_k are in conflict; otherwise s_j and s_i are in conflict. So, our assumption leads to a contradiction; and we have also shown that s_i and s_k are not in conflict. □

We are now ready to present a polynomial-time algorithm for the MSR problem on gapless SNP matrices. According to Lemma 12.16, it is sufficient to remove a minimum cardinality set of SNPs such that the resulting submatrix does not contain any SNP conflicts. For convenience, we do not try to minimize the number of SNPs to remove, but we instead try to find a maximum cardinality set of SNPs that do not have any SNP conflicts between them. The algorithm is based on dynamic programming. Recall that dynamic programming algorithms construct the solution for a problem instance from partial solutions, i.e., from solutions for parts of the given problem instances. In our case, as partial solutions for an input matrix with SNP set $\{s_1, \ldots, s_m\}$, the algorithm will calculate the maximum non-conflicting SNP sets $P_j \subseteq \{s_1, \ldots, s_j\}$ containing s_j, for all $j \in \{1, \ldots, m\}$.

We first formally fix our notation in the following definition.

Definition 12.27. *Let M be a gapless reduced $(n \times m)$-SNP matrix for the SNP set $S = \{s_1, \ldots, s_m\}$. For each SNP s_j, for $1 < j < m$, we denote by $\text{Good}(s_j)$ the set of all SNPs with a smaller index, that are not in conflict with s_j, i.e., $\text{Good}(s_j) = \{s_i \mid i < j \text{ and } \{s_i, s_j\} \text{ is not a SNP conflict}\}$.*

Furthermore, for each $j \in \{1, \ldots, m\}$, we denote by P_j a maximum-cardinality subset of non-conflicting SNPs from $\{s_1, \ldots, s_j\}$ that contains s_j. Moreover, we set $P_0 = \emptyset$.

Algorithm 12.6 Solving the MSR problem on gapless SNP matrices

Input: A gapless reduced SNP matrix for a SNP set $S = \{s_1, \dots, s_m\}$ and a fragment set $F = \{f_1, \dots, f_n\}$.

1. Construct the sets $\text{Good}(s_j)$ for all $s_j \in \{s_1, \dots, s_m\}$.
2. Set $P_0 := \emptyset$.
3. **for** $j := 1$ **to** m **do**
   3.1 Set $A_j := \text{argmax}_{P_i : s_i \in \text{Good}(s_j)} |P_i|$.
   3.2 Set $P_j := \{s_j\} \cup A_j$.
4. Set $P := \text{argmax}_{P_j : j \in \{1,\dots,m\}} |P_j|$.

Output: The computed maximum non-conflicting SNP set P.

We now show how we can recursively compute the sets P_j.

Lemma 12.18. *Let M be a gapless reduced $(n \times m)$-SNP matrix for the SNP set $S = \{s_1, \dots, s_m\}$. Then the partial solution P_j to the MSR problem can be computed as follows: Let $A_j = \text{argmax}_{P_i : s_i \in \text{Good}(s_j)} |P_i|$ be one of the largest of all sets P_i where $s_i \in \text{Good}(s_j)$. Then $P_j = \{s_j\} \cup A_j$.*

Proof. Let $s_i \in \text{Good}(s_j)$. Then, $\{s_j\} \cup P_i$ is a set of non-conflicting SNPs for all $i < j$, due to Lemma 12.17. It remains for us to show that $\{s_j\} \cup A_j$ is the largest non-conflicting subset of $\{s_1, \dots, s_j\}$ containing s_j. Assume that there is a larger non-conflicting set $X \subseteq \{s_1, \dots, s_j\}$ containing s_j. Let $i' < j$ be the largest index such that $s_{i'} \in X$. Then, $X - \{s_j\} \subseteq \{s_1, \dots, s_{i'}\}$ is a set of non-conflicting SNPs containing $s_{i'}$, and $|X - \{s_j\}| > |P_{i'}|$, contradicting the definition of $P_{i'}$. □

Based on Lemma 12.18, we can now formulate an algorithm for solving the MSR problem on gapless SNP matrices; it is given as Algorithm 12.6.

Theorem 12.5. *Algorithm 12.6 finds a maximum-cardinality set of SNPs for a given gapless reduced $(n \times m)$-SNP matrix in time $O(n \cdot m^2)$.*

Proof. The correctness of the algorithm immediately follows from Lemma 12.18. Constructing the sets $\text{Good}(s_j)$ for all SNPs can be done in $O(n \cdot m^2)$ time, and computing all partial solutions using the sets $\text{Good}(s_j)$ can be implemented in $O(m^2)$ time. □

Note that, using Algorithm 12.6, we are able to solve the MSR problem in polynomial time not only on gapless matrices, but also on matrices for which a permutation of the SNPs exists that makes the matrix gapless. The existence of such a permutation can be efficiently tested; and if such a permutation exists, it can also be found efficiently. The algorithm is analogous to the algorithm for transforming a binary matrix into consecutive-ones form, presented in Section 7.2.1.

In concluding this section, let us briefly discuss the relevance of gapless inputs. At first glance, they appear to be the only meaningful ones, as fragments are usually considered to be sequenced completely, resulting in a contiguous region in the SNP matrix. However, errors occurring during these reading techniques, as well as the application of concepts like mate pairs (as shortly discussed in Chapter 6), where only a prefix and a suffix of the fragment is actually sequenced, may lead to the presence of gaps in real data.

If the number of gaps is bounded by a constant, then the dynamic programming approach presented above can be extended in an appropriate way to yield good results for this case also.

12.3 Summary

Many organisms have a diploid genome, i.e., two slightly different copies of each genome, called haplotypes. The most frequent differences between the two haplotypes of an organism are the single nucleotide polymorphisms, or SNPs for short. Standard sequencing techniques only reveal the genotype, i.e., a sort of consensus of the two haplotypes. Since haplotype information is important, for example, for tracking genetic diseases, but experimentally difficult to obtain, computational methods are used for inferring the haplotypes from the given genotypes.

One approach proposed in this context uses the genotypes from a population of individuals and tries to resolve them into a set of haplotypes that, on the one hand, are consistent with the genotypes, and that, on the other hand, adhere to a perfect phylogeny model. From a computational point of view, this leads to the PPH problem solvable in polynomial time.

Another approach tries to infer the haplotypes of a single individual. Here, one tries to partition the DNA fragments from a shotgun sequencing experiment into the two haplotypes. The resulting combinatorial problem is to find a bipartition of a graph representing the fragments as vertices and the incompatibilities between them as edges. It is actually easily solvable for error-free data, where the input data is a bipartite graph itself, but becomes hard in its most general formulation, where, in the presence of errors, we seek to remove a minimal number of fragments (or SNPs) from the input data such that the induced graph becomes bipartite again. A special case, where the considered fragments are contiguous strips of the DNA and where all SNP sites inside the fragments have been detected, becomes polynomially solvable by a dynamic programming approach.

12.4 Bibliographic Notes

An overview of the different models for haplotyping is given in the survey papers by Bonizzoni et al. [35] and Halldorsson et al. [95], as well as in the book edited by Istrail et al. [107].

The problem of computing the haplotypes in a population from genotype information was introduced by Clark [48], who gave a partial solution based on a simple inference rule. The problem was further investigated by Gusfield in [92], who also proposed the perfect phylogeny approach [93]. The algorithm we presented in Section 12.1 goes back to a paper by Bafna et al. [19], a very similar algorithm was independently proposed by Eskin et al. [66]. Ding et al. presented a linear-time algorithm for the PPH problem in [61]. A good overview of other approaches to the haplotyping problem in populations is given by Gusfield [94].

The problem of haplotyping a single individual was introduced by Lancia et al. [125]. In this paper, the MFR and MSR problems were defined and their hardness was proven. The paper also proposed the first polynomial (but involved) algorithms for the gapless case; we have taken the proof of Lemma 12.16 from it. Rizzi et al. [167] presented simple dynamic-programming algorithms for the MSR and MFR problems; our presentation of Algorithm 12.6 is based on this paper. The algorithms were also extended to the case of a constant number of gap symbols per fragment in [167]. Another fast (but inexact) heuristic for the MFR problem was proposed by Panconesi and Sozio [152].

13

Higher-Dimensional Structures of Biomolecules

In the previous chapters, we more or less exclusively focussed on the primary structure, i.e., we dealt with the linear sequence of the basic units of the considered molecules only. But for the function of the molecules in living beings, their spatial structure is of essential significance. The spatial structure enables or prohibits binding to other molecules and in this way determines the particular function of the molecule. Because of this important fact, we will devote the subsequent sections to certain topics dealing with higher structural levels of molecules.

Although determining the spatial structure of a molecule directly is possible by X-ray crystallography and other methods, the procedures are very costly and require a lot of effort. Hence, one tries to infer information about the spatial structure of the molecule by analyzing its known primary structure. The methods applied in this context are manifold and complex. The difficulty of this task is due to many different factors; for instance, there may be several exceptions from the "rules" describing the folding process of the particular molecule; thus, considering all rules is not possible. Furthermore, the spatial structure may depend not only on the primary structure of the molecule, but also on other influencing factors, such as the surrounding medium or specific substrates. Due to this, all structures derived from the primary structure of the molecule must be viewed as candidates only, and must subsequently be verified (or falsified) by further experiments.

For all molecules considered in the previous chapters, i.e., nucleic acids and proteins, their higher-dimensional structures are divided into a hierarchical system. The basis is in all cases formed by the above mentioned primary structure, i.e., the sequence of nucleotides or amino acids, respectively, along the molecule. Based on this, we distinguish between secondary, tertiary, and, in the case of proteins, quaternary structures. The particular levels of structure are defined for certain types of molecules in a slightly different way; we discuss this topic in more detail later.

In Section 13.1 we take a closer look at the structural hierarchy of RNA and present some approaches to obtain its secondary structure. The knowledge

of the higher-dimensional structure of molecules can also serve as additional information (besides the primary structure) when comparing two molecules. Basic approaches, showing how this is possible in the case of RNA secondary structures, is discussed in Section 13.2. Finally, Section 13.3 is devoted to methods and approaches for determining higher-dimensional structures of proteins. The chapter is closed with a summary in Section 13.4 and references to further reading in Section 13.5.

13.1 RNA Secondary Structure Prediction

Before we start describing the structural hierarchy of RNA, we briefly review its composition. An RNA molecule consists of a chain of nucleotides, that may contain the bases adenine, cytosine, guanine, and uracil. In contrast to DNA molecules, which consist of two complementary chains of nucleotides connected to each other by hydrogen bonds between complementary bases, RNA usually occurs as a single-stranded molecule. There exists the possibility of pairings between complementary bases of the *same* RNA strand, which results in various different spatial structures.

As mentioned in the introduction to this chapter, we call the series of nucleotides along the RNA molecule its *primary structure*. This corresponds to the representation of RNA in terms of a string over the alphabet $\Sigma_{\text{RNA}} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$, as considered in previous chapters.

On the other hand, we call the folding of the RNA strand with itself by means of hydrogen bonds between bases at different positions in the same RNA strand its *secondary structure*. We will make this precise in the following definition and propose a representation for it.

Definition 13.1. *Let $r = r_1 r_2 \dots r_n$, where $r_i \in \Sigma_{\text{RNA}}$ for $1 \leqslant i \leqslant n$, be the primary structure of an RNA in its string representation. A* secondary structure *of RNA can then be represented in terms of a set* SecStruct_r *of pairs of indices from* $\{1, \dots, n\}$,

$$\text{SecStruct}_r \subseteq \{(i, j) \mid 1 \leqslant i < j \leqslant n\},$$

where base r_i is paired with base r_j.

Moreover, SecStruct_r *should satisfy the following properties:*

(i) Each index $k \in \{1, \dots, n\}$ occurs at most once in a pair from SecStruct_r*.*
(ii) For each pair (i, j) from SecStruct_r*, either (r_i, r_j) is a Watson-Crick pair, i.e., $(r_i, r_j) \in \{(\mathtt{A}, \mathtt{U}), (\mathtt{U}, \mathtt{A}), (\mathtt{C}, \mathtt{G}), (\mathtt{G}, \mathtt{C})\}$, or $(r_i, r_j) \in \{(\mathtt{G}, \mathtt{U}), (\mathtt{U}, \mathtt{G})\}$. We call these base pairs also* valid base pairs *in the following.*
(iii) For each pair (i, j) from SecStruct_r*, $j - i \geqslant 4$ holds.*

These properties impose some rather weak constraints on the general definition of secondary structures; they ensure, to some extent, the feasibility of

computation. Intuitively, property (i) requires that each base can be paired with at most one other base. Hence, a base is either paired or unpaired. Property (ii) ensures that a base pair is either a Watson-Crick pair or one of the also relatively stable pairs (G, U) or (U, G). While in reality other pairings may also occur, they are so rare that we may ignore them here. To allow for pairings in the same molecule, the RNA strand has to fold to a certain degree. Property (iii) describes the fact that the bends within such foldings cannot be too sharp, since this is prohibited by the binding angles between the atoms. We thus assume in particular that no pair should occur in the secondary structure where the corresponding bases are at distance less than 4 in the primary structure.

The real spatial conformation of RNA, i.e., the positions of the single atoms in space, the angle of the bindings, and so on, is referred to as the *tertiary structure* of RNA. According to this, the secondary structure represents a kind of transition on the way from the primary structure of the molecule to its actual spatial form. That it is indeed a hierarchical intermediate, becomes clear from Figure 13.1, where the secondary structure of a tRNA, often described as trefoil-shaped, contrasts with the tertiary structure, which rather resembles an "L". Nevertheless, it is useful to try to figure out the secondary structure of an RNA first and then utilize the information obtained to gain a hypothesis of the tertiary structure of the molecule, if the secondary structure does not reveal the molecule's fundamental characteristics.

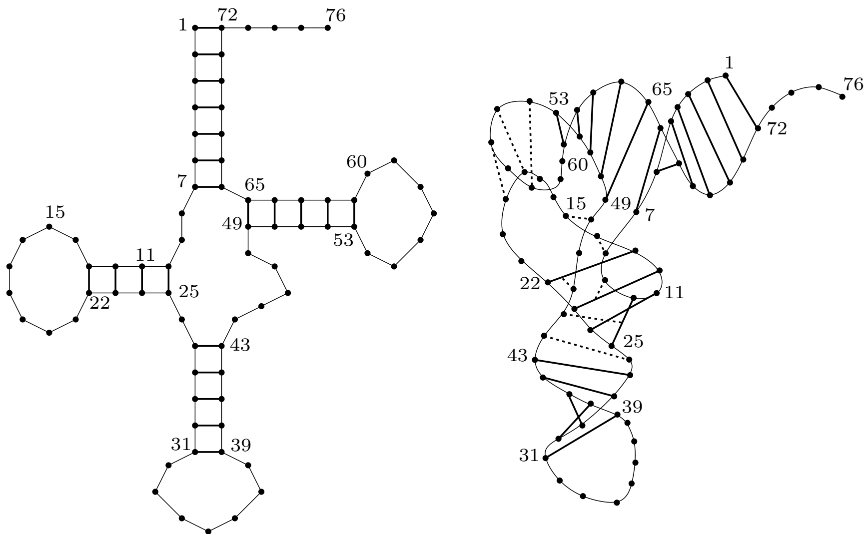


Fig. 13.1. Secondary and tertiary structure of a tRNA. Bold lines represent base pairings, dotted lines represent other intermolecular forces

In the following, we present various approaches to deriving the secondary structure of RNA from its primary structure.

13.1.1 Minimizing the Free Energy

Many algorithmically oriented approaches are based on the idea of computing the secondary structure that minimizes the free energy of the molecule. In this context, the free energy of a molecule refers to its potential to further release energy by participating in additional chemical bonds. Since in nature the molecules that have the least possible free energy are known to be the most stable ones, we try to figure out the secondary structure having the least free energy. Differences in the free energy of the molecules depend on the bindings between the bases of the RNA and are usually computed experimentally. We now specify the possible types of bindings and the resulting substructures in more detail.

To do so, we first define when a base, or a base pair, is *reachable* from a particular base pair.

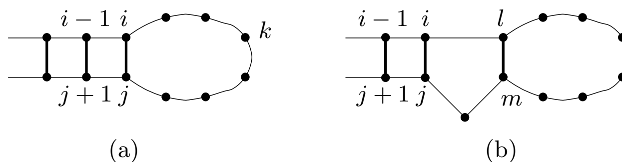


Fig. 13.2. Reachability of bases and base pairs from a particular base pair (i,j)

Based on this definition, we can now describe the above mentioned substructures of a secondary structure of RNA. We distinguish the following types of substructures that can be inferred by base pairings in the secondary structure of an RNA.

Definition 13.2. *Let r be the primary structure of an RNA, let* SecStruct_r *be a corresponding secondary structure, and let* $(i,j) \in \text{SecStruct}_r$ *be a base pair in the secondary structure. Then,* (i,j)*, or the induced substructure (see Figure 13.3) of the RNA, is called:*

- Stacked pair *if* $(i+1, j-1) \in \text{SecStruct}_r$*, i.e., if the reachable base pair is directly adjacent. A series of consecutive stacked pairs is called a* stem.
- Hairpin loop *if no base pair in* SecStruct_r *is reachable from* (i,j)*, i.e., a loop without any further base pair emerges from the base pair* (i,j)*.*
- Bulge *if there exists a base pair* $(i',j') \in \text{SecStruct}_r$ *reachable from* (i,j) *such that either* $i'-i > 1$ *or* $j-j' > 1$ *holds (but not both at the same time). Intuitively, there emerges a kind of bulb on one side of the double strand induced by the base pairs* (i,j) *and* (i',j')*.*

- Interior loop *if there exists a base pair* $(i', j') \in \text{SecStruct}_r$ *reachable from* (i, j) *such that* $i' - i > 1$ *as well as* $j - j' > 1$ *hold. In this case, a bulb emerges between* (i, j) *and the reachable base pair* (i', j') *that does not contain a further base pair.*
- Multiple loop *if there is more than one other base pair reachable from* (i, j)*. These structures may have various different forms.*

We denote the number of bases reachable from base pair (i, j) *as the size of the substructure.*

We refer to the collection of the substructures hairpin loop, bulge, interior loop, and multiple loop as *loops* for short.

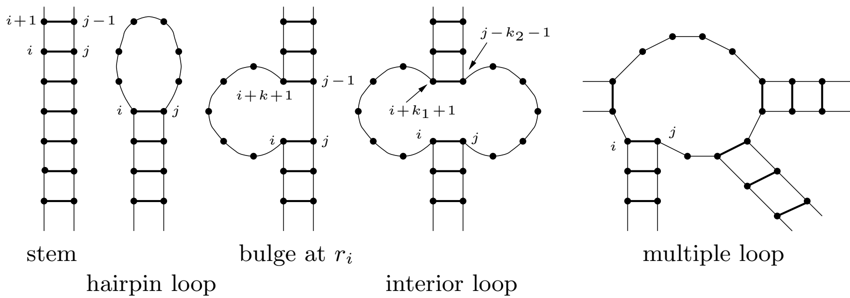


Fig. 13.3. Schematic view of possible substructures within an RNA secondary structure. Bases are shown as dots, connections within the RNA strand as lines, and base pairings as bold lines

It is easy to imagine that a substructure consisting of many stacked pairs will have a stabilizing effect. On the other hand, the other substructures will have, to a certain extent, a destabilizing effect on the secondary structure, where the degree of destabilization corresponds to the number of reachable (unpaired) bases. Due to this, the free energies of stems are negative, while those of loops are positive.[1]

Before we discuss some algorithmic approaches for addressing our problem, we define another complex substructure, called *pseudoknot*.

Definition 13.3. *Let* r *be the primary structure of an RNA. A corresponding secondary structure* SecStruct_r *contains a* pseudoknot *if there exist two base pairs* (i, j) *and* (k, l) *in* SecStruct_r *such that* $i < k < j < l$ *holds.*

Such a pseudoknot is shown in Figure 13.4. Pseudoknots occur in real secondary structures of RNA. But since they often essentially complicate the

[1] Recall that as little as possible free energy is assumed to correspond to a conformation of high stability.

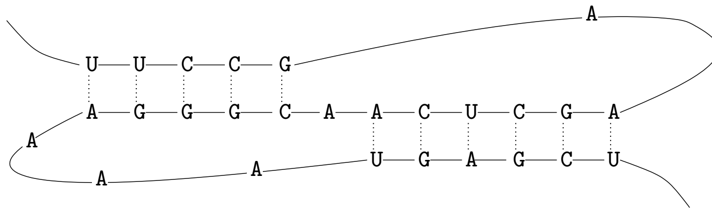


Fig. 13.4. Example of a pseudoknot

prediction of a secondary structure, they are omitted in most approaches for solving this problem; but they may be reconsidered in the following step for computing the tertiary structure of the RNA. Next, we present some algorithms that depend on the assumption that no pseudoknots occur inside the desired secondary structure. References to methods to actually introduce pseudoknots are given in the bibliographic notes in Section 13.5.

The Algorithm of Nussinov

One of the first approaches to determine the secondary structure of RNA from its primary structure was presented by Nussinov et al. [151] in 1978. The basic idea is to try maximizing the number of base pairs in the computed secondary structure. That is, the focus is on the number of stabilizing bindings only, while the kind of resulting substructures are left out of consideration. The algorithm is based on the method of *dynamic programming* we used in Chapter 5 for computing similarities between sequences. To apply this method to our task here, we consider substrings of different lengths of the string corresponding to the primary structure of the RNA. For each of the substrings, we compute an optimal (with respect to the measure described above) secondary structure, and the use secondary structures already computed for shorter substrings to successively derive secondary structures of longer substrings. We use the following notations. Let $r = r_1 r_2 \dots r_n$ be the string representation of the primary structure of the considered RNA. By $S_{i,j}$ we denote the optimal secondary structure for substring $r_i \dots r_j$. We refer to the number of base pairs contained in this secondary structure as $\mathrm{BP}(S_{i,j})$, i.e., $\mathrm{BP}(S_{i,j}) = |S_{i,j}|$. Now, the algorithm in principle fills up an $(n \times n)$-matrix with value $\mathrm{BP}(S_{i,j})$ at position (i, j). Finally, the value at position $(1, n)$ gives the desired result, the number of base pairs in the computed secondary structure. The particular secondary structure can then, starting from the final value, be reconstructed using a trace-back approach in a way similar to that shown in Chapter 5 for determining the alignment from the similarity matrix. The algorithm is presented in Figure 13.1; here, $\delta : \Sigma_{\mathrm{RNA}} \times \Sigma_{\mathrm{RNA}} \to \{0, 1\}$ denotes a function that yields 1 if the argument is a valid base pair, and 0 otherwise.

The cases considered in the recurrences are visualized in Figure 13.5 to show the intention behind them in more detail. Case (i) corresponds to the

Algorithm 13.1 Nussinov's Algorithm

Input: A string $r = r_1 \ldots r_n$.

1. Initialization:

  for $i = 2$ **to** n **do** $\text{BP}(S_{i,i-1}) := 0$
  for $i = 1$ **to** n **do** $\text{BP}(S_{i,i}) := 0$

2. Recurrence:

  for $l = 1$ **to** $n - 1$ **do**
    for $i = 1$ **to** $n - l$ **do**
      $j := i + l$;

$$\text{BP}(S_{i,j}) := \max \begin{cases} \text{BP}(S_{i+1,j}) & \text{(i)} \\ \text{BP}(S_{i,j-1}) & \text{(ii)} \\ \text{BP}(S_{i+1,j-1}) + \delta(r_i, r_j) & \text{(iii)} \\ \max_{k, i<k<j}\{\text{BP}(S_{i,k}) + \text{BP}(S_{k+1,j})\} & \text{(iv)} \end{cases}$$

Output: $\text{BP}(S_{1,n})$

possibility that base r_i will not be paired, and thus there is no change in the number of base pairs from substructure $S_{i+1,j}$ to $S_{i,j}$. The analogous case where r_j is not paired, is depicted as case (ii), and we thus obtain the same number of base pairs as in $S_{i,j-1}$. The possibility that bases r_i and r_j bind to each other is shown as case (iii). Accordingly, the number of base pairs in the resulting secondary structure $S_{i,j}$ is increased by 1 compared with $\text{BP}(S_{i+1,j-1})$ if $\delta(r_i, r_j) = 1$, i.e., if (r_i, r_j) is a valid base pair. Case (iv) represents the scenario where the optimal secondary structure is composed of two parts, $S_{i,k}$ and $S_{k+1,j}$. This type of composition implies, in particular, that pseudoknots cannot be considered here.

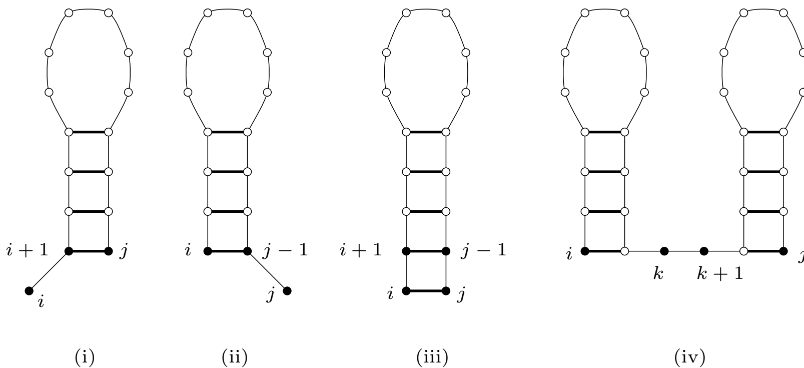


Fig. 13.5. Possible cases in the recurrence of Nussinov's Algorithm (Algorithm 13.1)

Algorithm 13.1 computes along this line only the optimal number of base pairs in a secondary structure; the structure itself can be inferred via a traceback approach as presented in the context of alignment algorithms.

Clearly, the procedure is a very rough simplification of reality, since only the number of base pairs in the secondary structure is considered here. Therefore, it is appropriate to extend this algorithm, at least in such a way that the particular free energy of the single base pairings is also considered. To do so, we denote by $fe(r_i, r_j)$ the free energy of base pair (r_i, r_j) and refine the above algorithm by including function *fe* instead of δ in case (iii). Since the free energy of a binding is negative, we then search for the structure that minimizes the energy. The modified algorithm then uses the recurrence

$$\mathrm{E}(S_{i,j}) = \min \begin{cases} \mathrm{E}(S_{i+1,j}) & \text{(i)} \\ \mathrm{E}(S_{i,j-1}) & \text{(ii)} \\ \mathrm{E}(S_{i+1,j-1}) + fe(r_i, r_j) & \text{(iii)} \\ \min_{k, i<k<j}\{\mathrm{E}(S_{i,k}) + \mathrm{E}(S_{k+1,j})\} & \text{(iv)} \end{cases} \tag{13.1}$$

where E refers to the free energy of the secondary structure (based on base pairs) and does not only count the number of base pairs BP as Algorithm 13.1 does. Using the function *fe*, we can furthermore do without the assumption of considering valid base pairs only, since valid base pairs will, by evaluation of the experimentally obtained function *fe*, yield lower free energies than invalid base pairs.

At this point, let us briefly consider the running time of Algorithm 13.1. Essentially, the algorithm computes values in the above right triangle of an $(n \times n)$-matrix (see Figure 13.6); this requires $O(n^2)$ steps. The computation of values for the cases (i), (ii), and (iii) in the recurrence only needs a constant effort. On the other hand, up to n different possibilities for the parameter k must be tested in case (iv). This requires $O(n)$ time, which altogether leads to a running time of the algorithm in $O(n^3)$.

Additionally, we would like to mention that Algorithm 13.1 in its present form does not take into account the bending property of Definition 13.1 (iii), but it is easy to come up with an appropriate extension (simply check whether the two bases of a possible pair are at the required distance).

The Algorithm of Zuker

Until now we have considered the free energy on the level of base pairs only for predicting the secondary structure of RNA, without taking into account what kinds of substructures with respect to Definition 13.2 result from such pairings. But the free energy of the secondary structure essentially depends on the substructures; stems have stabilizing effects (negative free energy), while loops have destabilizing effects (positive free energy). To take these effects into account, we now present a refined procedure that is also based on the method of dynamic programming and was proposed by Zuker and Stiegler [212].

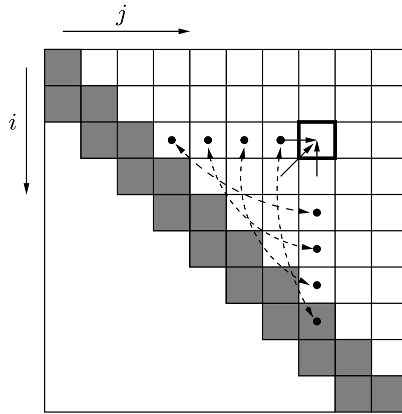


Fig. 13.6. Schematic view of how entries $\mathrm{BP}(S_{i,j})$ (or $\mathrm{E}(S_{i,j})$) are calculated by Nussinov's Algorithm. The left lower area of the matrix is not needed for the computation. The grey shaded area shows the fields initialized by step 1. An entry is obtained as the maximum (or minimum) of the values in the fields below (i), to the left of (ii), and to the left of and below (iii) the searched entry, and the sum of the values marked by a dot that are connected by dashed arcs (iv)

Algorithm 13.2 Zuker's Algorithm - Basic recurrence

Input: A string $r = r_1 \dots r_n$.

Recurrence:

for $l = 1$ **to** $n - 1$ **do**
  for $i = 1$ **to** $n - l$ **do**
    $j := i + l$;

$$\mathrm{E}(S_{i,j}) := \min \begin{cases} \mathrm{E}(S_{i+1,j}) & \text{(i)} \\ \mathrm{E}(S_{i,j-1}) & \text{(ii)} \\ \mathrm{E}(\mathrm{L}_{i,j}) & \text{(iii)} \\ \min_{k, i<k<j}\{\mathrm{E}(S_{i,k}) + \mathrm{E}(S_{k+1,j})\} & \text{(iv)} \end{cases}$$

Output: $\mathrm{E}(S_{1,n})$

We restrict ourselves here to the presentation of the recurrence; the initialization corresponds to that from Algorithm 13.1. We denote by $\mathrm{E}(S_{i,j})$, for each secondary structure $S_{i,j}$, the minimal free energy, where we additionally consider the different possibilities of substructures as well. The basic recurrence of Zuker's Algorithm is similar to recurrence (13.1) in the modified algorithm by Nussinov, and is presented as Algorithm 13.2.

Case (iii) in the recurrence corresponds to a pairing between bases r_i and r_j, as in Algorithm 13.1. But here we additionally consider the substructure induced by this base pair for determining the free energy. Therefore, we denote by $\mathrm{L}_{i,j}$ a substructure with minimal free energy that results from a base pairing (r_i, r_j). Before we actually give a recurrence for the computation of $\mathrm{E}(\mathrm{L}_{i,j})$,

we introduce some notation for the experimentally determined free energy of substructures.

Definition 13.4.
- $\text{fe}_{\text{stacked}}$ *denotes the free energy of a stacked pair, namely, the additional free energy released by another base pairing within a stem (stabilizing, thus negative).*
- $\text{fe}_{\text{hairpin}}(k)$ *denotes the free energy of a hairpin loop of size k (destabilizing, thus positive).*
- $\text{fe}_{\text{bulge}}(k)$ *denotes the free energy of a bulge of size k (destabilizing, thus positive).*
- $\text{fe}_{\text{interior}}(k)$ *denotes the free energy of an interior loop of size k (destabilizing, thus positive).*

Recall that we referred to the free energy of base pair (r_i, r_j) as $fe(r_i, r_j)$. Hence, except the formation of stacked pairs, all the substructures have a destabilizing effect, and thus the free energies will in general have positive values.

Based on the notations for specific free energy types introduced above, we can now formulate the recurrence for $\text{E}(\text{L}_{i,j})$ in Algorithm 13.3.

Algorithm 13.3 Zuker's Algorithm — Recurrence for $\text{E}(\text{L}_{i,j})$

$$\text{E}(\text{L}_{i,j}) := \begin{cases} fe(r_i, r_j) + \text{fe}_{\text{stacked}} + \text{E}(S_{i+1,j-1}), & \\ \quad \text{if } \text{L}_{i,j} \text{ is a stem} & \text{(a)} \\ fe(r_i, r_j) + \text{fe}_{\text{hairpin}}(j-i-1), & \\ \quad \text{if } \text{L}_{i,j} \text{ is a hairpin loop} & \text{(b)} \\ \min_{k \geqslant 1}\{fe(r_i, r_j) + \text{fe}_{\text{bulge}}(k) + \text{E}(S_{i+k+1,j-1})\}, & \\ \quad \text{if } \text{L}_{i,j} \text{ is a bulge at } r_i & \text{(c)} \\ \min_{k \geqslant 1}\{fe(r_i, r_j) + \text{fe}_{\text{bulge}}(k) + \text{E}(S_{i+1,j-k-1})\}, & \\ \quad \text{if } \text{L}_{i,j} \text{ is a bulge at } r_j & \text{(d)} \\ \min_{k_1,k_2 \geqslant 1}\{fe(r_i, r_j) + \text{fe}_{\text{interior}}(k_1+k_2) + \text{E}(S_{i+k_1+1,j-k_2-1})\}, & \\ \quad \text{if } \text{L}_{i,j} \text{ is a interior loop} & \text{(e)} \end{cases} \tag{13.2}$$

For a better understanding of the particular cases we refer the reader to Figure 13.3.

After describing Algorithm 13.2 completely, we now consider its running time. Analogously to the analysis of Algorithm 13.1, we want to fill up a matrix of size $O(n^2)$ with values for the corresponding free energy for a given input string $r = r_1 \dots r_n$. As in Algorithm 13.1, cases (i) and (ii) in the basic recurrence in Algorithm 13.2 require a constant effort, case (iv) requires a linear effort. To determine the effort in case (iii) (computing $\text{E}(L_{i,j})$) we have to consider the recurrence (13.2). Here, (a) and (b) again imply a constant effort,

and (c) and (d) a linear one. On the other hand, a naive implementation[2] of (e) will lead to an effort in $O(n^2)$, since all possible values for parameters k_1 and k_2 have to be tested ($1 \leqslant k_1, k_2 \leqslant n$). Altogether, we can conclude that Algorithm 13.2 can be implemented to require a running time in $O(n^4)$.

Also, Algorithm 13.2 is based on simplifying assumptions: On the one hand, no pseudoknots are considered, and on the other hand, multiple loops and other conformations influencing the free energy are not taken into account. Nevertheless, this algorithm constitutes the basis for many extensions that also account for these. We come back to this topic in Section 13.5.

The algorithm can be modified in such a way that not only optimal (with respect to the considered model) secondary structures can be computed, but also approximate ones. This set of similarly good hypotheses can then be further tested for relevance in biological experiments.

13.1.2 Stochastic Context-Free Grammars

In this section we deal with another approach to compute good hypotheses for the secondary structure of RNA. The idea is to describe the secondary structure of RNA with a set of context-free rules, which are applied with a certain probability. The possibility of a base pairing ($\mathtt{C}, \mathtt{G}$) may, for instance, be modeled by using a rule of type $A \to \mathtt{C}A\mathtt{G}$. We make this approach concrete in the following. To start, we give the definition of a context-free grammar.

Definition 13.5. *A* context-free grammar *G is a 4-tuple $G = (N, T, P, S)$, where N and T are alphabets, with $N \cap T = \emptyset$.*

- *N is called the alphabet of non-terminals,*
- *T is called the alphabet of terminals,*
- *$S \in N$ is called the start symbol, and*
- *$P \subseteq N \times (N \cup T)^*$ is called the set of productions (derivation rules), where P is a finite set. We also represent an element $(u, v) \in P$ as $u \to v$.*

Let $u, v \in (N \cup T)^*$, and $u = w_1 A w_2$ and $v = w_1 v' w_2$ for some $w_1, w_2 \in (N \cup T)^*$. If there exists a production $(A, v') \in P$, then we say that v is derived from *u* in *G and denote this by $u \Rightarrow_G v$. By $\Rightarrow_G^*$ we refer to the reflexive and transitive closure of $\Rightarrow_G$. (That is, in a derivation we allow for an arbitrary number of arbitrary productions.) A series of the form $w_0 \Rightarrow_G w_1 \Rightarrow_G \ldots \Rightarrow_G w_n$ will be called a* derivation in *G.*

A string $w \in T^*$ is generated *by a grammar G if there exists a derivation of the form $S \Rightarrow_G^* w$. The set of strings generated by G is called the* language *of G (or generated by G)*

$$L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}.$$

[2] References to a more efficient implementation can be found in Section 13.5.

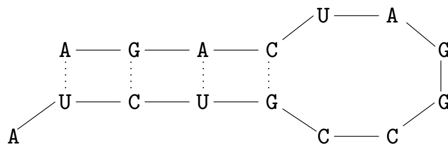


Fig. 13.7. Schematic view of the secondary structure from Example 13.1

In the previous definition, context-freeness refers to the kind of rules allowed in the grammar, where the left-hand side consists of a single non-terminal only, and thus the application of a production does not rely on the context, i.e., the surrounding symbols, of the non-terminal.

As this brief definition suffices for our purposes, we do not further discuss the theory of grammars here. Instead, we motivate it by a short example how we would like to model the secondary structure of an RNA by means of a context-free grammar.

Example 13.1. Let $r = \texttt{AGACUAGGCCGUCUA}$ be the string representation of an RNA and let a corresponding secondary structure be given by $\text{SecStruct}_r = \{(1, 14), (2, 13), (3, 12), (4, 11)\}$ (see also Figure 13.7). Let us now consider the following context-free grammar $G = (\{S\}, \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}, P, S)$, with

$$\begin{aligned} P = \{&S \to \texttt{A}S \mid \texttt{C}S \mid \texttt{G}S \mid \texttt{U}S, \\ &S \to S\texttt{A} \mid S\texttt{C} \mid S\texttt{G} \mid S\texttt{U}, \\ &S \to \texttt{A}S\texttt{U} \mid \texttt{U}S\texttt{A} \mid \texttt{C}S\texttt{G} \mid \texttt{G}S\texttt{C}, \\ &S \to \texttt{A} \mid \texttt{C} \mid \texttt{G} \mid \texttt{U}\}. \end{aligned}$$

Here, the notation $S \to u \mid v$ is used as an abbreviation for the two rules $S \to u$ and $S \to v$. Note that grammar G (like every grammar) is independent from the actual string we may derive, and thus universally applicable to all possible strings.

Next, we show that r is contained in the language $L(G)$ by exposing a derivation. To do this, we successively apply productions from the first line of the above set of productions to subsequently generate the particular characters in r.[3]

$$\begin{aligned} S &\Rightarrow_G \texttt{A}S \Rightarrow_G \texttt{AG}S \Rightarrow_G \texttt{AGA}S \Rightarrow_G \texttt{AGAC}S \Rightarrow_G \texttt{AGACU}S \Rightarrow_G \texttt{AGACUA}S \\ &\Rightarrow_G \texttt{AGACUAG}S \Rightarrow_G \texttt{AGACUAGG}S \Rightarrow_G \texttt{AGACUAGGC}S \Rightarrow_G \texttt{AGACUAGGCC}S \\ &\Rightarrow_G \texttt{AGACUAGGCCG}S \Rightarrow_G \texttt{AGACUAGGCCGU}S \Rightarrow_G \texttt{AGACUAGGCCGUC}S \\ &\Rightarrow_G \texttt{AGACUAGGCCGUCU}S \Rightarrow_G \texttt{AGACUAGGCCGUCUA} \end{aligned}$$

[3] Actually, $L(G)$ contains all strings over the alphabet Σ_{RNA}, i.e., $L(G) = \Sigma_{\text{RNA}}^*$, as each string can easily be derived by applying the productions in line 1 (or line 2) only.

But there also exists a derivation for r in G whose sequence of productions mirrors the secondary structure SecStruct_r.

$$S \Rightarrow_G S\texttt{A} \Rightarrow_G \texttt{A}S\texttt{UA} \Rightarrow_G \texttt{AG}S\texttt{CUA} \Rightarrow_G \texttt{AGA}S\texttt{UCUA} \Rightarrow_G \texttt{AGAC}S\texttt{GUCUA}$$
$$\Rightarrow_G \texttt{AGACU}S\texttt{GUCUA} \Rightarrow_G \texttt{AGACUA}S\texttt{GUCUA} \Rightarrow_G \texttt{AGACUAG}S\texttt{GUCUA}$$
$$\Rightarrow_G \texttt{AGACUAGG}S\texttt{GUCUA} \Rightarrow_G \texttt{AGACUAGGC}S\texttt{GUCUA} \Rightarrow_G \texttt{AGACUAGGCCGUCUA}$$

In this derivation, we used, after applying production $S \rightarrow S\texttt{A}$ from the second line of P to obtain the overhanging A, derivation rules of type $S \rightarrow \texttt{X}S\texttt{Y}$ from the third line of P to describe the base pairings of the secondary structure. From the rules used in the derivation, we can directly infer the secondary structure. ◊

We have seen in the previous example how to draw conclusions on the secondary structure of an RNA from the application of certain productions in a derivation of the RNA's string representation. We now extend the notion of context-free grammars by a stochastic component to evaluate positively those derivations that represent a good hypothesis for the secondary structure of the considered RNA.

Definition 13.6. *A* stochastic context-free grammar $G_{\text{stoch}} = (N, T, P, S, \rho)$ *is a context-free grammar* $G = (N, T, P, S)$*, with an additional function* $\rho : P \rightarrow [0, 1]$*, that assigns a probability to each production in* P *such that, for all* $A \in N$*,*

$$\sum_{\substack{x \in (N \cup T)^* \\ A \rightarrow x \in P}} \rho(A \rightarrow x) = 1.$$

A stochastic context-free grammar thus assigns a probability to each production. Hence, also each derivation of a string w gets assigned a probability, namely the product of probabilities of the productions used in the derivation. Let α be a derivation in a stochastic context-free grammar $G_{\text{stoch}} = (N, T, P, S, \rho)$; then,

$$\text{Prob}(\alpha) = \rho(p_1) \cdot \rho(p_2) \cdot \ldots \cdot \rho(p_k)$$

gives the probability of derivation α in G_{stoch}, where $p_1, \ldots, p_k$ is the series of production used in α.[4] The probability of a string w to be generated by a stochastic context-free grammar $G_{\text{stoch}} = (N, T, P, S, \rho)$ is then the sum of all probabilities of all derivations for w in G_{stoch}:

$$\text{Prob}(w) = \sum_{\alpha \text{ is derivation for } w \text{ in } G_{\text{stoch}}} \text{Prob}(\alpha).$$

[4] Note that some productions may be applied several times within a derivation, accordingly, there may exist i and j, $i \neq j$, where $p_i = p_j$.

We have seen in Example 13.1 how a derivation can correspond to the secondary structure of an RNA. Our next goal is to assign a high probability to a derivation, that leads to a good hypothesis for the secondary structure. This implies the following tasks.

1. *Generating a stochastic context-free grammar:* Let D be a set of training data consisting of pairs of primary and secondary structures of RNA. From this, a stochastic context-free grammar should be constructed whose derivations of a primary structure r in D have high probability if these derivations mirror the corresponding secondary structure of r.
2. *Determining the most probable derivation:* Let G_{stoch} be a stochastic context-free grammar as constructed in step 1. For the primary structure r of an RNA with a yet unknown secondary structure, we want to compute the most probable derivation of r in G_{stoch}. This is based on the hope that structures in D learned in step 1 can be generalized to the unknown secondary structure sequence r, to obtain a good hypothesis of the secondary structure of r by finding a derivation with high probability. Formally, we would like to determine β as

$$\beta = \underset{\alpha}{\operatorname{argmax}}\{\text{Prob}(\alpha) \mid \alpha \text{ is a derivation of } r\}.$$

In this book, we do not further discuss the construction of a grammar as required in step 1, instead we will describe a procedure for step 2. Regarding approaches to perform step 1, we refer the reader to the bibliographic notes in Section 13.5.

First, let us restrict the general form of a context-free grammar in such a way that we only allow a particular type of derivation rule.

Definition 13.7. *A context-free grammar $G = (N, T, P, S)$ is said to be in* Chomsky normal form*,* CNF *for short, if all productions in P are of type*

- *$A \to BC$, where $A, B, C \in N$, or*
- *$A \to a$, where $A \in N$ and $a \in T$.*

It is possible to show that all context-free grammars can be transformed into a grammar in CNF, such that the languages generated are the same. We will not prove this in general, but exemplarily consider production $S \to \mathtt{C}S\mathtt{G}$ from Example 13.1. Recall, $S \in N$ denotes a non-terminal, and $\mathtt{C}, \mathtt{G} \in T$ are terminals of our grammar. Production $S \to \mathtt{C}S\mathtt{G}$ can now be replaced by productions $S \to XY$, $X \to \mathtt{C}$, $Y \to SZ$, and $Z \to \mathtt{G}$, where $X, Y, Z \in N$ are new non-terminals, without changing the language of the grammar.

Now, a stochastic context-free grammar is in Chomsky normal form, if the corresponding context-free grammar is in Chomsky normal form. Transforming an arbitrary stochastic context-free grammar into a stochastic CNF requires the transformation of the underlying grammar. Moreover, it is possible to transform the probabilities ρ of productions as well, such that corresponding derivations will have also the same probability. For our example above, let

the original probability of production $S \to \mathtt{C}S\mathtt{G}$ be given by $\rho(S \to \mathtt{C}S\mathtt{G})$. We denote the probabilities of productions in the corresponding grammar in CNF by ρ', and they may be set to $\rho'(S \to XY) = \rho(S \to \mathtt{C}S\mathtt{G})$, $\rho'(X \to \mathtt{C}) = 1$, $\rho'(Y \to SZ) = 1$, and $\rho'(Z \to \mathtt{G}) = 1$.

Next, we introduce a structure that nicely represents derivations of a grammar.

Definition 13.8. *Let $G = (N, T, P, S)$ be a context-free grammar. A directed ordered rooted tree D_G is called a* derivation tree *of G if*

- *The root is labeled with the start-symbol S.*
- *Each inner vertex is labeled with a symbol from N.*
- *Each leaf is labeled with a symbol from $T \cup \{\lambda\}$.*
- *Letting A be the label of an inner vertex n and $B_1, B_2, \ldots, B_k$ be the labeling of its children in left-to-right order,we have*

$$A \to B_1 B_2 \ldots B_k \in P.$$

- *For a vertex n labeled λ, n is a leaf and the only child of its parent vertex.*

A derivation tree $D_{G,w}$ of string w in grammar G is now a derivation tree of G such that the labeling of leaves of $D_{G,w}$ read in left-to-right order yields w. A derivation tree in this way specifies a certain selection of productions to derive a particular string. However, the order of applying these rules may vary. Let us consider the following example.

Example 13.2. Let G be the grammar from Example 13.1; i.e.,

$$\begin{aligned} P = \{&S \to \mathtt{A}S \mid \mathtt{C}S \mid \mathtt{G}S \mid \mathtt{U}S, \\ &S \to S\mathtt{A} \mid S\mathtt{C} \mid S\mathtt{G} \mid S\mathtt{U}, \\ &S \to \mathtt{A}S\mathtt{U} \mid \mathtt{U}S\mathtt{A} \mid \mathtt{C}S\mathtt{G} \mid \mathtt{G}S\mathtt{C}, \\ &S \to \mathtt{A} \mid \mathtt{C} \mid \mathtt{G} \mid \mathtt{U}\}. \end{aligned}$$

The derivation trees of the string $r = \mathtt{AGACUAGGCCGUCUA}$ that correspond to the derivations in Example 13.1 are shown in Figure 13.8 (a) and (b), respectively. If we now extend P by another production $S \to SS$, we can, for the string $r' = \{\mathtt{AGAC}\}$, get a derivation tree as shown in Figure 13.9. Here, $|r'|$ non-terminals S are produced first by multiple application of the rule $S \to SS$, and then the particular terminals are derived using productions of the form $S \to a$, $a \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$. This derivation tree represents a number of different derivations:

$$\begin{aligned} S &\Rightarrow_G SS \Rightarrow_G SS\,S \Rightarrow_G \mathtt{A}S\,S \Rightarrow_G \mathtt{AG}\,S \\ &\Rightarrow_G \mathtt{AG}\,SS \Rightarrow_G \mathtt{AG}\,\mathtt{A}S \Rightarrow_G \mathtt{AGAC} \end{aligned} \tag{13.3}$$

$$\begin{aligned} S &\Rightarrow_G SS \Rightarrow_G S\,SS \Rightarrow_G S\,S\mathtt{C} \Rightarrow_G S\,\mathtt{AC} \\ &\Rightarrow_G SS\,\mathtt{AC} \Rightarrow_G S\mathtt{G}\,\mathtt{AC} \Rightarrow_G \mathtt{AGAC} \end{aligned} \tag{13.4}$$

$$\begin{aligned} S &\Rightarrow_G SS \Rightarrow_G SS\,S \Rightarrow_G \mathtt{A}S\,S \Rightarrow_G \mathtt{A}S\,SS \\ &\Rightarrow_G \mathtt{AG}\,SS \Rightarrow_G \mathtt{AG}\,S\mathtt{C} \Rightarrow_G \mathtt{AGAC} \end{aligned} \tag{13.5}$$

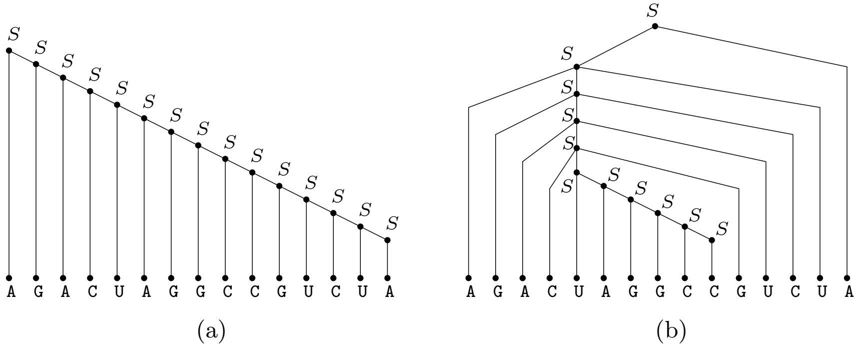


Fig. 13.8. Derivation trees for $r =$ AGACUAGGCCGUCUA and G from Example 13.2 (the direction of the edges is implicitly considered to be from top to bottom)

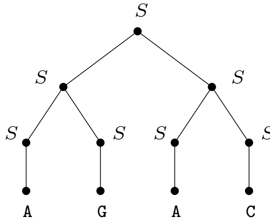


Fig. 13.9. Derivation tree for the grammar from Example 13.2, extended by the production $S \to SS$

Derivation (13.3) is also called left-most derivation, since always the left-most non-terminal is replaced; analogously, we refer to derivation (13.4) as the right-most derivation. Besides left- and right-most derivations, also combinations of both may occur, as in case of derivation (13.5). ◊

Thus, we have seen that a derivation tree may describe several different derivations, but each of them uses the same collection of productions. Since in stochastic context-free grammars probabilities are assigned to the productions, each derivation represented by a particular derivation tree of a stochastic context-free grammar has the same probability. Therefore, it is meaningful to look at the probability of a derivation tree for a string and a given stochastic context-free grammar instead of the probability of a particular derivation. We now aim at determining such a derivation tree with maximum probability.

To do so, we again use an approach that depends on the concept of dynamic programming. The algorithm in some sense resembles the Viterbi algorithm for HMMs presented in Section 9.4. We are given a string r and a stochastic context-free grammar G in CNF. Starting with the terminal symbols in r, we now try to go back to the starting symbol S by reversing the possible productions. During this process we store the probabilities of the productions,

Algorithm 13.4 Stochastic CYK algorithm

Input: A stochastic context-free grammar G_{stoch} in CNF and a string $w = w_1 \dots w_n$ over the alphabet T.

1. Initialization:

   for $i = 1$ **to** n **do**
     for $l = 1$ **to** m **do**
       $\gamma(i, i, l) = \rho(\mathtt{A}_l \to w_i)$;

2. Recurrence:

   for $d = 1$ **to** $n - 1$ **do**
     for $i = 1$ **to** $n - d$ **do**
       for $l = 1$ **to** m **do**
         $j := i + d$
         $\gamma(i, j, l) = \max_{\substack{A_l \to A_x A_y \in P \\ k = i, \dots, j-1}} \{\gamma(i, k, x) \cdot \gamma(k + 1, j, y) \cdot \rho(A_l \to A_x A_y)\}$;

Output: $\gamma(1, n, 1)$

---

to finally yield the derivation tree (the order of application of productions is not determined by this approach) with maximal probability. This algorithm is a modification of the CYK algorithm, named after its inventors Cocke, Younger, and Kasami, that was originally designed to compute whether or not a given string w can be generated by a given context-free grammar G in CNF, i.e., whether or not $w \in L(G)$ holds.

To describe the algorithm, we use the following notation. Let $G_{\text{stoch}} = (N, T, P, A_1, \rho)$ be the considered stochastic context-free grammar in CNF, where $N = \{A_1, \dots, A_m\}$.[5] For the sake of simplicity we may assume that productions not occurring in P are assigned probability 0 by ρ. Furthermore, let $w = w_1 \dots w_n$ be a string over the alphabet T. By $\gamma(i, j, l)$ we denote the maximal sum of probabilities of productions to generate the string $w_i \dots w_j$, starting with the non-terminal A_l in G_{stoch}. These probabilities are first computed for strings of length 1, and then successively for longer substrings of w until, finally, $\gamma(1, n, 1)$ yields the maximal probability of generating the string w using G_{stoch}. The algorithm is given as Algorithm 13.4. Again, it only computes the value of the maximal probability, not the desired derivation tree itself. Similarly to determining the optimal alignment in Chapter 5, we can apply a traceback approach here.

Due to the nested for loops in step 2 and the computation of the maximum over all possible values k, this algorithm requires a running time in $O(n^3 \cdot m)$. As a rule, it is implemented using the logarithms of the probabilities. The reasons for this were discussed in the context of the Viterbi algorithm in Section 9.4.

[5] P thus contains productions of type $A_h \to A_i A_j$ and $A_h \to b$ only.

Algorithm 13.5 Computing a most probable derivation based on Nussinov's algorithm

Input: The grammar G_{stoch} as described above and a string $r = r_1 \dots r_n$.

1. Initialization:

   for $i = 2$ **to** n **do**
   $\gamma(i, i-1) := 0$
   for $i = 1$ **to** n **do**
   $\gamma(i,i) := \max\{\rho(S \to r_i S), \rho(S \to S r_i)\}$

2. Recurrence:

   for all $j - i = 1$ **to** $n - 1$ **do**

$$\gamma(i,j) := \max \begin{cases} \gamma(i+1,j) \cdot \rho(S \to r_i S) \\ \gamma(i,j-1) \cdot \rho(S \to S r_j) \\ \gamma(i+1,j-1) \cdot \rho(S \to r_i S r_j) \\ \max_{k, i<k<j}\{\gamma(i,k) \cdot \gamma(k+1,j) \cdot \rho(S \to SS)\} \end{cases}$$

Output: $\gamma(1,n)$.

In concluding this section, we will Algorithm 13.1 described in Section 13.1.1 and show how this algorithm can be simulated using a stochastic context-free grammar. Let us therefore consider the extended stochastic context-free grammar from Example 13.2, namely, $G_{\text{stoch}} = (\{S\}, \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}, P, S, \rho)$, where

$$\begin{aligned} P = \{&S \to \mathtt{A}S \mid \mathtt{C}S \mid \mathtt{G}S \mid \mathtt{U}S, && (13.6)\\ &S \to S\mathtt{A} \mid S\mathtt{C} \mid S\mathtt{G} \mid S\mathtt{U}, && (13.7)\\ &S \to \mathtt{A}S\mathtt{U} \mid \mathtt{U}S\mathtt{A} \mid \mathtt{C}S\mathtt{G} \mid \mathtt{G}S\mathtt{C}, && (13.8)\\ &S \to SS, && (13.9)\\ &S \to \lambda\}. \end{aligned}$$

The productions in (13.6) correspond to the case in Nussinov's algorithm where base r_i does not perform a binding (see Figure 13.5 (i)); in an analogous way, the productions in (13.7) correspond to the case where r_j is unpaired (see Figure 13.5 (ii)). The pairing of bases r_i and r_j is resembled by the derivation rules in (13.8) (see Figure 13.5 (iii)); and the splitting into two substructures is simulated in (13.9) (see Figure 13.5 (iv)). Finally, the production $S \to \lambda$ allows for eliminating useless non-terminals S.

For this grammar, one may now estimate the probability function ρ based on training data, transform it into CNF, and then determine the most probable derivation tree for a string r in the resulting grammar. From this we may then obtain the secondary structure of r. However, in this case one can also apply a recurrence that essentially resembles that of Nussinov's algorithm, and achieves the most probable derivation tree directly, thus avoiding the detour of transforming the grammar into CNF. The resulting algorithm is presented as Algorithm 13.5.

Finally, we again point out that modelling secondary structures of RNA by means of stochastic context free grammars has some advantages. It enables us to assign a probability to each secondary structure (each derivation tree) of an RNA r, and hence we can compute not only optimal, but also near optimal structures (of course according to our model only), as hypotheses for the real secondary structure of the molecule. It is worth noting that the stochastic context-free grammar used for the modelling process, is independent of the particular RNA sequence considered, i.e., it is generally applicable.

13.2 Structure-Based Comparison of Biomolecules

After having investigated some methods for deriving the secondary structure of an RNA from its primary structure in the previous section, we now want to present a possible application of the newly gained knowledge on the secondary structure. When we investigated similarities between RNA sequences before, it was based on their primary structures only; corresponding methods were considered in Chapter 5. Next, we would like to design a way to include this additional knowledge in the comparison. Let us therefore consider the secondary structure of an RNA as shown in Figure 13.10 (a). If we now represent the primary structure as a string maintaining in terms of arcs the connections between pairs of characters, that correspond to base pairs in the secondary structure, then we obtain the representation given in Figure 13.10 (b). We call this kind of representation *arc-annotated string*.

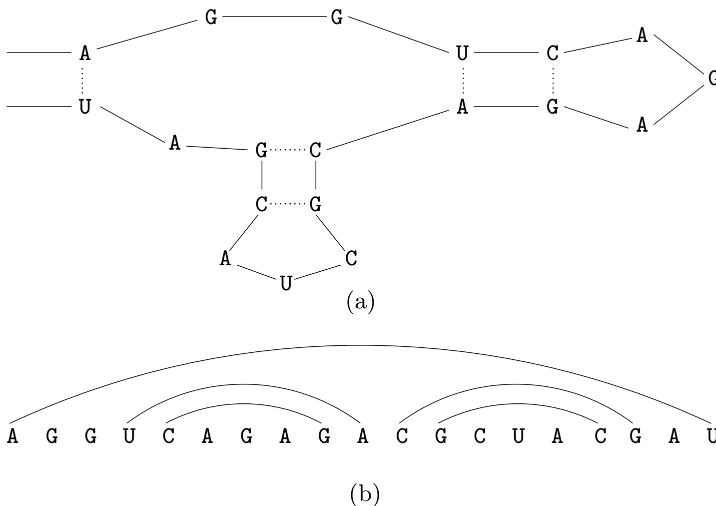


Fig. 13.10. (a) The secondary structure of an RNA; (b) the corresponding arc-annotated string

At this point we abstract from the particular meaning of an arc-annotated string as a representation of RNA secondary structure, and examine this notion in its whole generality.

Definition 13.9. *Let $s = s_1 s_2 \ldots s_n$ be a string over an alphabet Σ and let $P \subseteq \{(i,j) \mid 1 \leqslant i < j \leqslant n\}$ be an unordered set of position pairs in s. We call $S = (s, P)$ an* arc-annotated string[6] *with* string *s and* arc set *P. A pair (i, j) from the arc set P is called an* arc.

The arc-annotated string shown in Figure 13.10 (b) thus has the formal representation $S = (s, P)$, where

$$s = \texttt{AGGUCAGAGACGCUACGAU} \text{ and}$$
$$P = \{(1,19), (4,10), (5,9), (11,17), (12,16)\}$$

How can we now measure the similarity between two arc-annotated strings? Let us first recall the (global) alignment of two strings, as presented in Section 5.1.2. If we value each match 1, and each mismatch and gap 0, we get the particular problem of searching the *longest common subsequence.*

Definition 13.10. *The longest common subsequence problem, LCS for short, is the following optimization problem.*

Input: Two strings $s = s_1 s_2 \ldots s_n$ and $t = t_1 t_2 \ldots t_m$ over a common alphabet Σ.
Feasible solutions: Each common subsequence[7] $w \in \Sigma^*$ of s and t.
Cost: The length of subsequence w, $cost(w) = |w|$.
Optimization goal: Maximization.

Hence, given two strings s and t, we search for the longest string w that is a subsequence of s as well as of t that originates from deleting characters at certain positions in s and t, respectively.

Example 13.3. Let $s = abcdabcdabcd$ and $t = dcaaabaabdcd$. A longest common subsequence of s and t is $w = aababcd$, having length 7.

$$\begin{array}{cc|c|ccc|cc|cc|cc|c|cc|}
- & - & a & b & c & d & a & b & c & d & a & b & - & c & d \\
d & c & a & - & a & - & a & b & a & - & a & b & d & c & d
\end{array}$$

◊

On the other hand, one may also understand each subsequence as a mapping of positions in one string to positions in the other.

---

[6] In the literature, the term *arc-annotated sequence* is also used.

[7] Recall that a string w is called a subsequence of a string s if all symbols in w occur in the same order as in s. A subsequence is thus not necessarily a substring (see Definition 3.3).

Definition 13.11. *Let $s = s_1 s_2 \dots s_n$ and $t = t_1 t_2 \dots t_m$ be two strings and let $w = w_1 w_2 \dots w_k$ be a common subsequence of s and t. Then a bijective mapping φ from a subset M_s of $\{1, \dots, n\}$ onto a subset M_t of $\{1, \dots, m\}$ is called* consistent *with w if it satisfies the following properties.*

(i) Mapping φ preserves the order of symbols along the strings s and t, i.e., for all $i_1, i_2 \in M_s$,

$$i_1 < i_2 \iff \varphi(i_1) < \varphi(i_2).$$

(ii) The symbols on positions assigned by φ are equal, i.e., for all $i \in M_s$,

$$s_i = t_{\varphi(i)}.$$

A mapping consistent with a common subsequence w of two strings s and t is thus a feasible assignment of positions in s to positions in t, such that the symbols at the positions in s, t, and w are all the same. We may also describe this mapping by its position pairs, leading to the following representation:

$$\varphi = \{\langle x, y \rangle \mid \varphi(x) = y, x \in \{1, \dots, n\}, y \in \{1, \dots, m\}\}.$$

We use $\langle\rangle$ to better distinguish between elements from φ and the arc sets P of the arc-annotated strings.

For the common subsequence w from Example 13.3, a consistent mapping φ is given by

$$\varphi = \{\langle 1, 3\rangle, \langle 5, 5\rangle, \langle 6, 6\rangle, \langle 9, 8\rangle, \langle 10, 9\rangle, \langle 11, 11\rangle, \langle 12, 12\rangle\}.$$

The mapping consistent with a subsequence is not necessarily unique. So, for instance, also the mapping

$$\varphi' = \{\langle 1, 4\rangle, \langle 5, 5\rangle, \langle 6, 6\rangle, \langle 9, 8\rangle, \langle 10, 9\rangle, \langle 11, 11\rangle, \langle 12, 12\rangle\}$$

is consistent with the common subsequence w from Example 13.3. The mapping φ' thus corresponds to an alignment of string s and t in the following way:

$$\begin{array}{ccc|c|cccc|cc|cc|cc|c|cc|} - & - & - & a & b & c & d & a & b & c & d & a & b & - & c & d \\ d & c & a & a & - & - & - & a & b & a & - & a & b & d & c & d \end{array}$$

But independently from this, in general the length of a common subsequence w is identical to the number of position pairs in each of its consistent mappings φ.

Focussing on the length, we can, instead of considering the string representation of a common subsequence, deal with its consistent mapping. In the following, we consider in each case the type of description that appears to be most suitable to illustrate the issue.

After having considered the LCS in more detail, we extend this problem to measure the similarity of structures (instead of strings) which are provided in terms of arc-annotated strings. To do so, we first define what we want to understand as an *arc-preserving common subsequence*.

Definition 13.12. *Let $S = (s_1 s_2 \ldots s_n, P_s)$ and $T = (t_1 t_2 \ldots t_m, P_t)$ be two arc-annotated strings over an alphabet Σ. A string w is called an* arc-preserving common subsequence *of S and T if there exists a mapping φ consistent with w such that*

(i) $s_i = t_j$, for all elements $\langle i, j \rangle$ from φ, and
(ii) for all pairs of elements $(\langle i_1, j_1 \rangle, \langle i_2, j_2 \rangle)$ from φ

$$(i_1, i_2) \in P_s \iff (j_1, j_2) \in P_t.$$

An arc-preserving common subsequence is thus a string, which is a subsequence of both arc-annotated strings and which contains all existing arcs. In particular, property (ii) in the above definition implies that, if *both* ends of an arc in S are assigned to positions in T by φ, then between these assigned positions in T there has to be an arc as well.

Definition 13.13. *The longest arc-preserving common subsequence problem, LAPCS for short, is the following optimization problem.*

Input: Two arc-annotated strings $S = (s_1 \ldots s_n, P_s)$ and $T = (t_1 \ldots t_m, P_t)$ over an alphabet Σ.
Feasible solutions: Each arc-preserving common subsequence $w \in \Sigma^*$ of S and T.
Cost: The length of subsequence w, $cost(w) = |w|$, i.e., the number of position pairs in a mapping φ consistent to w, $cost(w) = cost(\varphi) = |\varphi|$.
Optimization goal: Maximization.

We can easily imagine that the structure of arcs within the arc-annotated strings influences the complexity of the LAPCS problem presented above. Yet, we did not put any structural requirements onto the set of arcs; but now we introduce a classification of arc-annotated strings according to certain criteria.

Let P denote the arc set of an arc-annotated string $S = (s_1 \ldots s_n, P)$. We distinguish the following four properties of P.

(R.1) No two arcs in P share a common endpoint: For any two arbitrary arcs $(i_1, j_1), (i_2, j_2) \in P$, we have

$$i_1 \neq j_2, j_1 \neq i_2 \quad \text{and} \quad i_1 = i_2 \iff j_1 = j_2.$$

(R.2) No two arcs in P cross each other: For any two arbitrary arcs (i_1, j_1), $(i_2, j_2) \in P$, we have

$$i_2 < i_1 < j_2 \iff i_2 < j_1 < j_2.$$

(R.3) No two arcs in P are nested: For any two arbitrary arcs (i_1, j_1), $(i_2, j_2) \in P$, we have

$$i_1 < i_2 \iff j_1 < j_2.$$

(R.4) The arc set P contains no arcs: $P = \emptyset$.

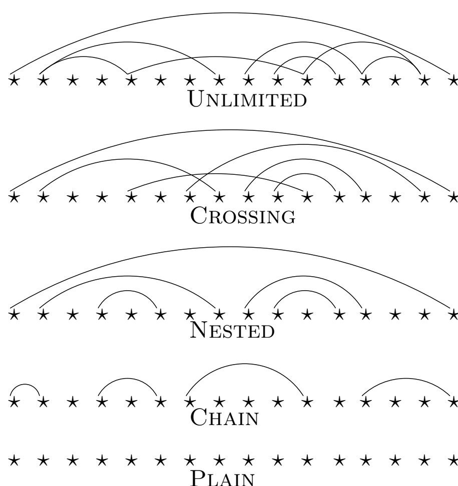


Fig. 13.11. Representatives from the different classes of arc-annotated strings. The $\star$-symbol is a wildcard for arbitrary characters

According to these restrictions we may now classify arc-annotated strings into the following classes.

Definition 13.14. *We specify the following classes of arc-annotated strings.*

- Unlimited $= \{S = (s, P) \mid S$ is an arc-annotated string$\}$
- Crossing $= \{S = (s, P) \mid S \in$ Unlimited and P satisfies (R.1)$\}$.
- Nested $= \{S = (s, P) \mid S \in$ Unlimited and P satisfies (R.1) and (R.2)$\}$.
- Chain $= \{S = (s, P) \mid S \in$ Unlimited and P satisfies (R.1) to (R.3)$\}$.
- Plain $= \{S = (s, P) \mid S \in$ Unlimited and P satisfies (R.4)$\}$.

With respect to this definition we thus obtain the following inclusions:

$$\text{Plain} \subsetneq \text{Chain} \subsetneq \text{Nested} \subsetneq \text{Crossing} \subsetneq \text{Unlimited}\,.$$

Characteristic examples for each of these classes are shown in Figure 13.11. Recalling our original motivation to study arc-annotated strings, we can relate the classes to corresponding RNA secondary structures as follows. The class Crossing includes all secondary structures of RNA as we considered them in Section 13.1. The only restriction is that for RNA secondary structures we allow a base only to pair with at most one other base. In particular, we can represent a pseudoknot in Crossing (see for instance Figure 13.12, where the pseudoknot from Figure 13.4 is shown in terms of an arc-annotated string). If we restrict ourselves to structures without pseudoknots, this yields structures belonging to the class Nested. With nested arcs we can describe all substructures of secondary structure as considered in Section 13.1, such as stems, hairpin loops, and bulges (see Figure 13.10). The class Chain only contains

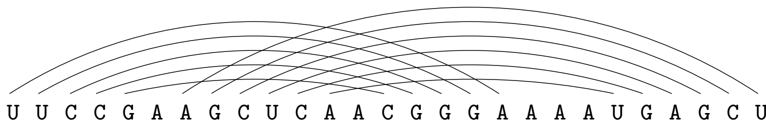


Fig. 13.12. Representation of the pseudoknot from Figure 13.4 as an arc-annotated string

strings with a highly restricted arc structure and thus can be considered an intermediate class of Nested and Plain that encloses strings without an arc annotation, that is, where we focus on similarities between primary structures of RNA only.

In the following, we study the LAPCS problem in more detail. It is helpful to refine the problem setting by regarding the classes of arc-annotated strings defined above.

Definition 13.15. *By* LAPCS($\text{Level}_1, \text{Level}_2$) *we denote the optimization problem LAPCS for arc-annotated strings* $S = (s_1 \dots s_n, P_s)$ *and* $T = (t_1 \dots t_m, P_t)$*, where* $S \in \text{Level}_1$ *and* $T \in \text{Level}_2$*.* Level_1 *and* Level_2 *refer to one of the classes described in Definition 13.14, i.e.,* $\text{Level}_1, \text{Level}_2 \in \{$Unlimited*,* Crossing*,* Nested*,* Chain*,* Plain$\}$*.*

Since the order of the arc-annotated strings S and T in the problem instance plays no role, we agree upon writing the classes of LAPCS introduced in Definition 13.15 in such a way that always $\text{Level}_2 \subseteq \text{Level}_1$, i.e., we write LAPCS(Crossing, Chain) instead of LAPCS(Chain, Crossing).

It is beyond the scope of this textbook to present solutions or complexity results for all the different classes of LAPCS from Definition 13.15. Hence, we focus on only some of the classes and refer the reader to the literature referenced in Section 13.5 for more comprehensive surveys.

The class LAPCS(Plain, Plain) corresponds to finding the longest common subsequence, since there are no arcs added to the strings. This problem, as mentioned at the beginning of this chapter, resembles the global alignment problem for two strings with a specific evaluation function, and is thus solvable in polynomial time.

If, on the other hand, we consider the class LAPCS(Crossing, Crossing), we can show this optimization problem to be NP-hard. To give the reduction, we first introduce the clique problem, known to be an NP-complete decision problem [79, 116].

Definition 13.16. *Let* $G = (V, E)$ *be an undirected graph. A subset* V' *of* V *is called a* clique*, if every two vertices* $v_i, v_j \in V'$*, where* $v_i \neq v_j$*, are connected by an edge in* G*, i.e.,* $\{v_i, v_j\} \in E$*.*

So, a clique is simply a complete (sub)graph.

Definition 13.17. *The clique problem is the following decision problem.*

Input: *An undirected graph* $G = (V, E)$ *and a positive integer* k.
Output: YES *if* G *contains a clique* V' *of size* k *(i.e., where* $|V'| = k$*).* NO *otherwise.*

Starting from the NP-completeness of the clique problem, we now prove that LAPCS(CROSSING, CROSSING) is an NP-hard optimization problem.

Theorem 13.1. LAPCS(CROSSING, CROSSING) *is an NP-hard optimization problem.*

Proof. For the proof of this claim we consider the decision version DECLAPCS (CROSSING, CROSSING) of LAPCS(CROSSING, CROSSING), which, in addition to the arc-annotated strings S and T from the class CROSSING, gets a natural number l as input, and gives the answer YES if there exists an arc-preserving common subsequence of S and T of length l, and the answer NO otherwise.

Let $G = (V, E)$, together with a number $k \in \mathbb{N}$, be an input for the clique problem, where $V = \{v_1, \ldots, v_n\}$. From this we construct an input (S, T, l) for DECLAPCS(CROSSING, CROSSING) as follows (see also Figure 13.13):

- The arc-annotated string $S = (s, P_s)$ is intended to encode the graph G, including its edges. Therefore we use a representation that reminds us of an adjacency matrix of a graph. Each vertex is transformed into a block of a's of length n framed by two b's, i.e., ba^nb. To encode an edge $\{v_i, v_j\}$, we now introduce an arc from the jth a in the block representing the ith vertex to the ith a in the block representing the jth vertex. Moreover, each block representing a vertex is covered by an arc linking the corresponding b's. Formally, we have
$$s = (ba^nb)^n,$$
$$P_s = \{((i-1)(n+2)+j+1, (j-1)(n+2)+i+1) \mid \{v_i, v_j\} \in E\} \cup \{((i-1)(n+2)+1, i(n+2)) \mid i \in \{1, \ldots, n\}\}.$$

- The second arc-annotated string $T = (t, P_t)$ should now serve to represent the clique of size k we are searching for. We proceed as in coding the graph G above. For each vertex, we construct a block of a's framed by a b at each end. Since we only want to encode a clique of size k, we accordingly need only k blocks containing only k symbols a each. Because a clique is a complete graph, we have to represent all possible edges in it of size k by introducing arcs in the same way as in S. Furthermore, each block is covered by an arc over the corresponding b's as well. Formally, this gives
$$t = (ba^kb)^k,$$
$$P_t = \{((i-1)(k+2)+j+1, (j-1)(k+2)+i+1) \mid i, j \in \{1, \ldots, k\}, i \neq j\} \cup \{((i-1)(k+2)+1, i(k+2)) \mid i \in \{1, \ldots, k\}\}.$$

- Now, since our goal is to check whether graph G contains a clique of size k, we have to ask whether the arc-annotated string T is a subsequence of S, because T represents a clique of size k and S represents the graph G. Accordingly, we ask for an arc-preserving common subsequence of S and T that has the same length as T itself. Our input parameter l for DecLAPCS(Crossing, Crossing) thus is

$$l = |T| = k \cdot (k+2).$$

In this way, we have constructed arc-annotated strings, that on the one hand have a quadratic length in the number of vertices in V and in k, and whose arcs are on the other hand directly derived from the edges of the original graph and the edges of a clique. Hence, the construction of (S, T, l) from (G, k) can be performed in polynomial time. This reduction from a particular input instance (G, k) of the clique problem into an input instance (S, T, l) for DecLAPCS(Crossing, Crossing) is illustrated in Figure 13.13.

It remains for us to show that the existence of a clique of size k in G implies the existence of an arc-preserving common subsequence of S and T of length l, and vice versa.

1. Let $\{v_{i_1}, \ldots, v_{i_k}\}$ be a clique of size k in G. Then, we are able to align the symbols from T with respect to S such that k blocks of T are assigned to the blocks $i_1, \ldots, i_k$ in S. In each block of T, the k symbols a are again assigned to the symbols a at positions $i_1, \ldots, i_k$ in the block of S. Since this assignment only maps a's to a's and b's to b's and moreover also preserves the order of characters in both strings s and t, it remains for us to show that the arcs are also mapped to arcs. Arcs spanning two b's framing a block are preserved because we always map complete blocks in T to complete blocks in S. Since $v_{i_1}, \ldots, v_{i_k}$ are vertices of a clique in G, the vertices are pairwise connected to each other and all corresponding a's in S are spanned by arcs. Since T represents a clique, i.e., it contains all possible edges, the arcs are preserved. Thus, the arc-preserving common subsequence corresponds to T and hence is of length l.
2. If we now consider the converse case where we have an assignment of length l, then we can argue as follows. Due to the arcs over the b's framing a block, only blocks can be mapped to blocks. Since the arc-annotated string T represents a clique of size k, and edges/arcs are constructed by the same rules as in S, only an assignment of the type described above is possible. Let $i_1, \ldots, i_k$ be the blocks in S that are assigned to blocks in T; then, $\{v_{i_1}, \ldots, v_{i_k}\}$ is a clique of size k in G. □

We have proved that LAPCS(Crossing, Crossing) is an NP-hard optimization problem. Moreover, we can also infer NP-hardness for all classes above (Crossing, Crossing). For instance, it is clear that LAPCS (Unlimited, Unlimited) is also NP-hard. The proof of Theorem 13.1 also yields that the decision problem to decide whether one arc-annotated string is an

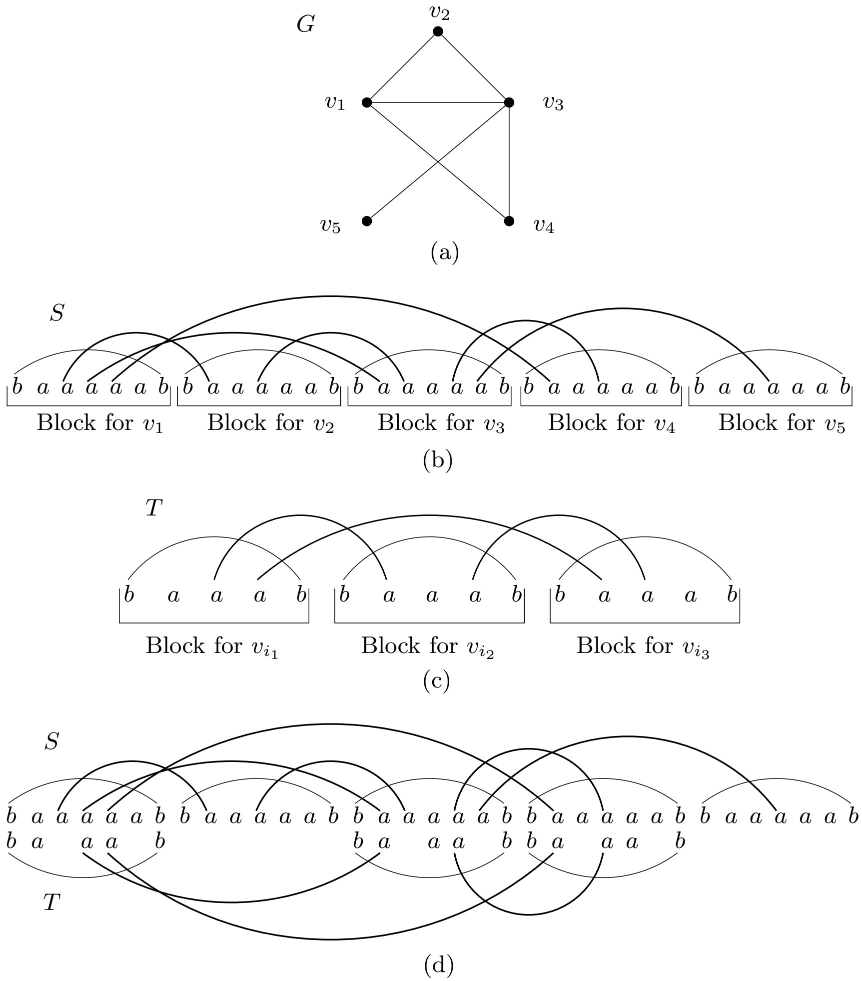


Fig. 13.13. An input (G, k) for the clique problem and the corresponding input instance of the DecLAPCS(Crossing, Crossing, l) problem. In (a) the graph G is shown for which we ask for a clique of size $k = 3$. The arc-annotated string S in (b) represents the graph G, where arcs corresponding to edges in the graph are drawn bold. In (c) the representation of a clique $\{v_{i_1}, v_{i_2}, v_{i_3}\}$ of size 3 of G is shown in terms of an arc-annotated string T. We are now looking for an arc-preserving subsequence of length 15. An assignment φ, corresponding to such a subsequence and representing the clique $\{v_1, v_3, v_4\}$ in G, is given in (d)

arc-preserving subsequence of another arc-annotated string is NP-hard for the class (CROSSING, CROSSING).

To come back to our biological motivation of comparing RNA secondary structures, Theorem 13.1 also says that it is in general hard to compare structures containing pseudoknots to each other, since they belong to the class CROSSING.

Because of this hardness result, we relax our requirement to compute the *longest* arc-preserving common subsequence, and instead search for good approximation algorithms for the LAPCS(CROSSING, CROSSING) problem. The algorithms do not necessarily yield the longest subsequence, but one for which we are able to prove that it is at most a constant fraction shorter than the optimal one. In the following, we present an algorithm that computes an arc-preserving subsequence that is at least half as long as the optimal one.

Let $S = (s, P_s)$ and $T = (t, P_t)$ be two arc-annotated strings. First, we compute the longest common subsequence w from s and t, this can be performed in polynomial time using the same method as that for determining an optimal alignment between two strings as presented in Chapter 5. That is, we completely ignore the arcs in the first step. Let us now consider a mapping φ consistent with w and construct the following graph $G_\varphi = (V, E)$ from it:

- $V = \{\langle i, j\rangle \mid \langle i, j\rangle \in \varphi\}$.
  Thus, position pairs in φ correspond to vertices in the graph.
- $E = \{\{\langle i_1, j_1\rangle, \langle i_2, j_2\rangle\} \mid \text{either } (i_1, i_2) \in P_s \text{ or } (j_1, j_2) \in P_t\}$.
  Two position pairs are connected by an edge if either P_s contains an arc between the respective positions while the corresponding arc does not exist in P_t, or vice versa. These edges shall thus represent which position pairs violate the arc-preserving property in the current subsequence w. If a particular arc occurs in both arc-sets P_s and P_t, then there exists *no* edge, since the arc was preserved, and there is hence no conflict with the arc-preserving property.

So, the graph G_φ describes which position pairs are not arc-preserving. To change the mapping φ to become arc-preserving, we now remove position pairs/vertices such that there are finally no edges left in the graph. Therefore, we first observe that each vertex in graph G_φ has at most degree 2 (at most one arc for the position in S and at most one for the position in T). If we now consider the connected components (recall Definition 12.15 for the definition of a connected component) of G_φ, these are either isolated vertices, paths, or cycles. In the case of isolated vertices there is nothing left to do. For a path or cycle, we may delete every second vertex, including its incident edges, and in this way obtain separated vertices only.

The graph resulting from this procedure describes position pairs of a mapping φ' that corresponds to an arc-preserving common subsequence of S and T. The steps of this algorithm are summarized in Algorithm 13.6.

We now investigate the running time and approximation ratio of the algorithm.

Algorithm 13.6 Approximation of LAPCS(Crossing, Crossing)

Input: Two arc-annotated strings $S = (s, P_s)$ and $T = (t, P_t)$ with $S, T \in$ Crossing, and $s = s_1 \dots s_n$ and $t = t_1 \dots t_m$.

1. Determine a longest common subsequence w of s and t. Let φ be a mapping consistent to w.
2. Construct the graph G_φ from φ as described above.
3. For each connected component in G_φ that consists of more than a single vertex, delete every second vertex, including its incident edges from G_φ.
4. Let $G_{\varphi'}$ be the resulting graph and φ' the set of position pairs that correspond to the vertices in $G_{\varphi'}$.
5. From φ', construct the corresponding string w'.

Output: The arc-preserving common subsequence w'.

Theorem 13.2. *Algorithm 13.6 computes a feasible solution for the* LAPCS *(*Crossing, Crossing) *problem and requires a running time in $O(n \cdot m)$, where n and m denote the lengths of the input strings.*

Proof. We first show that the string computed by Algorithm 13.6 is an arc-preserving common subsequence.

The string w' results from removing some symbols in w, namely, the symbols that correspond to a position pair deleted when transferring G_φ into $G_{\varphi'}$. Moreover, w is the longest common subsequence from s and t, which also implies that w' is a subsequence of s and t. Furthermore, the mapping φ' corresponding to string w' is arc-preserving. This follows from the fact that, in the graph G_φ, vertices/position pairs violate the arc-preserving property if and only if they are connected by an edge. In step 3 of the algorithm, a set of vertices/position pairs is deleted, such that the remaining graph $G_{\varphi'}$ contains no edges anymore, and thus the mapping φ' representing these position pairs does not violate the arc-preserving property.

The running time of Algorithm 13.6 can now be determined as follows. By using the method for computing a global alignment presented in Chapter 5, we can perform step 1 in time $O(n \cdot m)$. From this alignment, we obtain the position pairs included in the mapping φ by one run from left to right.

To construct the graph G_φ in step 2, and, in particular, to determine the edges, we have to check for each two position pairs $\langle i_1, j_1 \rangle, \langle i_2, j_2 \rangle$ in φ whether $(i_1, i_2) \in P_s$ and $(j_1, j_2) \in P_t$. Since φ contains at most $\min(n, m)$ position pairs, this step requires a running time in $O(\min(n, m)^2) \subseteq O(n \cdot m)$.

For traversing G_φ and performing the vertex removal in step 3, we investigate for each vertex v in G_φ, whether it is an isolated vertex, or whether v occurs in a path or cycle. Let us recall that each vertex in G_φ has at most degree 2, since $S, T \in$ Crossing, and we thus allow for at most one arc at each symbol in s and t. Hence, we can traverse the edges starting from v in at most two directions, and finally determine whether v lies on a path

or a cycle. Since each vertex has at most degree 2, G_φ contains at most $\frac{2\cdot\min(n,m)}{2} = \min(n,m)$ edges. Therefore, the above procedure requires time in $O(\min(n,m)\cdot\min(n,m)) \subseteq O(n\cdot m)$. If v is an isolated vertex, nothing has to be done. If v is part of a path, we can remove each second vertex of this path, starting with one of its end vertices. If v occurs in a cycle, we can start the deletion process with v itself, and remove every second vertex together with its incident edges from the graph. Using the same reasoning as above, this can be done in time $O(n \cdot m)$, indicating that the entire step 3 can be performed in time $O(n \cdot m)$.

From the remaining position pairs in $G_{\varphi'}$, we can now compute the string w' in time $O(\min(n,m))$. Summing up, Algorithm 13.6 requires a running time in $O(n \cdot m)$. □

Theorem 13.3. *Algorithm 13.6 is a 2-approximation algorithm for the* LAPCS(Crossing, Crossing) *problem.*

Proof. Let $S = (s, P_s)$ and $T = (t, P_t)$ be two arc-annotated strings. Let w_{opt} be a longest arc-preserving common subsequence of S and T, and let w' be the output of Algorithm 13.6. Then, each arc-preserving common subsequence of S and T is at most as long as a common subsequence of s and t. Let w be the longest common subsequence of s and t computed in step 1 of the algorithm. Then,

$$|w| \geqslant |w_{\text{opt}}|. \tag{13.10}$$

Since in step 3 of the algorithm we removed only every second vertex from each path or cycle in G_φ we deleted at most half the number of vertices. Thus, we have

$$2 \cdot |w'| \geqslant |w|. \tag{13.11}$$

Combining (13.10) and (13.11) we obtain the claimed result. □

We now consider, as in Section 13.1, structures without pseudoknots, i.e., we deal with the LAPCS(Nested, Nested) problem. For this problem the following result was shown [139].

Theorem 13.4. LAPCS(Nested, Nested) *is an NP-hard optimization problem.* □

In this case we can again apply Algorithm 13.6, which will provide a 2-approximation. In general, the proof of Theorem 13.3 is solely based on the restriction of the vertex degree in G_φ and the fact that the length of a longest common subsequence always establishes an upper bound on the length of a longest arc-preserving common subsequence. Therefore, Algorithm 13.6 is a 2-approximation algorithm for all problem classes contained in LAPCS(Crossing, Crossing), and, in particular, also for LAPCS(Nested, Nested).

The concept of structure-based comparison of molecules (or structure alignment) may clearly be applied also to other chain-like molecules besides

RNA. Arcs between characters may therefore refer to atoms that are only a small distance apart in the spatial formation of the molecule. The resulting arc-annotated string is often called a contact map. Moreover, this approach may serve as an intermediate step for actually predicting the spatial structures of molecules by comparing hypothetical structures to known structures stored in a database. We present an approach following this idea in the next section.

13.3 Protein Structure Prediction

Proteins are essential for a variety of important tasks in every organism; in particular, they catalytically enhance many vital processes as enzymes, serve as molecular building blocks of hairs and nails, take over signaling and defence functions, act as transport devices, e.g., hemoglobin is the carrier for oxygen in the blood, and more. To accomplish their tasks, their spatial structure is crucial, since it essentially determines their function. The knowledge of their spatial structure is therefore required to gain a deeper understanding of the complex processes in which proteins are involved and to be able to influence them, for instance, by means of drugs. One way to derive this structure is based on X-ray crystallography or nuclear magnetic resonance (NMR). However, the techniques are extremely time consuming and expensive. Consequently, computer-based approaches may be favored. Determining the primary structures of proteins, namely, their amino acid sequences, already constitutes a complex task. However, we do not further discuss this issue, but instead focus on the prediction of the spatial structures of proteins based on the knowledge of their primary structures. At a higher level, one may distinguish two different approaches.

De novo protein structure prediction: Given only the amino acid sequence of a protein, the goal is to derive its three-dimensional structure.

Knowledge based protein structure prediction: Here, the idea is to utilize the observation that, although the number of different proteins is very high, the number of different three-dimensional shapes seems to be relatively restricted. Given the known structures from a database, the goal is to match the primary structure of the studied protein to them, and to eventually derive insights on the protein's structural components.

Before we present two approaches to exemplarily discuss the two problems in more detail, we first give some basic facts on protein structure.

The formation of a spatial structure by proteins is referred to as *folding*. It is known that the folding behavior of proteins essentially depends on their primary structure. This is at least the case for a large and important class of proteins. However, there are also some enzymes (called chaperones) that enhance the folding process, but do not influence it. Changes in the spatial structure may also naturally occur due to chemical connections of the protein

with other molecules, or the surrounding media may play a certain role, but we will not discuss these issues here. For the computation of the spatial structure from the primary structure of a protein, the dependence of the folding on the primary structure clearly is crucial.

Let us now study the different structural levels of proteins in more detail. To do so, we first take a closer look at the composition of proteins.

Let us first recall the description of proteins given in Section 2.1. According to this, the basic building blocks of proteins are amino acids. By linking the carboxyl group and the amino group of the amino acids by peptide bonds, a polypeptide chain is formed. The bonds between the amino groups and carboxyl groups constitute the backbone of the chain. The side chains of the single amino acids are not involved here (see Figure 2.2). The number of amino acids along the polypeptide chain may vary from hundreds to thousands, and its sequence is denoted as the *primary structure* of the protein.

According to the side chains, the amino acids have different chemical properties. One distinguishes between hydrophobic, polar (sometimes also referred to as hydrophilic), acidic, and basic amino acids. We will not further study these characteristics here, but we mention that it has been observed that hydrophobic amino acids often occur inside the molecule and form hydrophobic cores there by hydrophobic interactions, while polar amino acids often appear at the border of the molecule where they take part in interactions with the surrounding polar medium, such as water.

Let us now discuss the structural hierachy of proteins. The higher-dimensional structures, resulting from interactions in the backbone of the polypeptide chain, are called *secondary structures.* At this level, interactions between side chains are ignored. One essentially distinguishes between two types of secondary structures, the *helices* and the *sheets*, often also referred to as *α-helices* and *β-sheets.* A helix consists of a screw-like twisting of the backbone, where the side chains are directed toward the outside of the helix. Bonds between the single atoms of the backbone in the windings provide stability to the structure (see Figure 13.14 (a)). On the other hand, β-sheets consist of parallel alignments of different regions of the polypeptide chain, whose backbone atoms interact as well. The side chains here point to the outside of the sheet (see Figure 13.14 (b)). In more detail, one distinguishes between parallel and antiparallel sheet structures, according to the direction of the particular parts of the chain. The direction is defined to be oriented from the end with the free amino group to the end with the free carboxyl group. Typically, sheet structures are depicted using broad arrows, indicating the direction of the chains, and helices are depicted by cylinders or helices. According to their chemical properties, amino acids may exhibit a disposition to occur in either helices or sheets. The regions of the backbone participating in neither a helix nor a sheet structure, called *loops.* Hence, loops are essentially links between helices and sheets in the secondary structure.

The *tertiary structure* of a protein refers to the spatial arrangement of the single atoms in the polypeptide chain. Here, one typically groups together

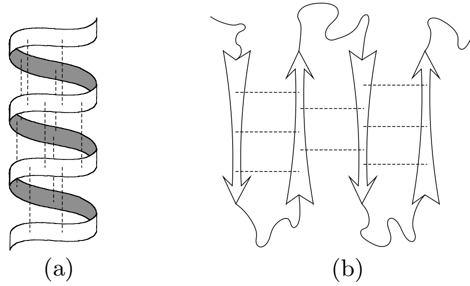


Fig. 13.14. Different types of protein secondary structures: (a) a helix; (b) a sheet. Stabilizing bonds between units in the backbone of the molecules are indicated by dashed lines

certain series of secondary structure elements into so-called *motifs*. A possible motif may be, for instance, the following series of secondary structures

sheet — loop — helix — loop — sheet — loop — helix — loop — sheet

as depicted in Figure 13.15. Larger units are called *domains*. They have a certain function and often form the "active center" of the protein, to which substrates may bind to or where other actions take place.

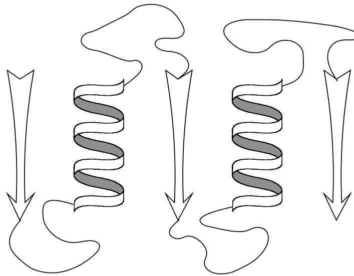


Fig. 13.15. A motif consisting of the following series of secondary structure elements: sheet — loop — helix — loop — sheet — loop — helix — loop — sheet

Eventually, some proteins consist of more than a single polypeptide chain, and may even include other molecular parts. With respect to this, the single polypeptide chains are called subunits of the protein molecule. The arrangement of the subunits with each other is referred to as the *quaternary structure* of the protein. A well-known example is the red blood cell, mainly consisting of the protein hemoglobin, whose main task is the transport of oxygen within the blood. This molecule consists of four subunits, each supplemented by an additional molecular part, called the hem (including an iron atom), which is responsible for the red color.

Summing up, we can describe the protein structure according to the following hierarchy.

Definition 13.18. *With regard to the spatial structure of proteins, four different levels are distinguished.*

Primary structure: The primary structure is the sequence of amino acids along the polypeptide chain.

Secondary structure: The secondary structure describes the interactions between atoms on the backbone of the polypeptide chain, forming substructures like α-helices, β-sheets, and loops.

Tertiary structure: The tertiary structure refers to the spatial arrangement of all atoms within the polypeptide chain. Secondary structure elements are grouped together as motifs and functional units called domains.

Quaternary structure: Finally, the quaternary structure describes the composition of the whole protein from polypeptide subunits and potentially other molecular parts.

As mentioned above, there are numerous approaches and models to determine the spatial structures of proteins from their primary structures. In general, one may again try to compute all possible structures and evaluate them according to their free energy, searching for the one with the lowest free energy. But this approach is not practical in this case for two reasons. First, it is impossible to handle all possible spatial structures because of their huge number, and second, there is yet no complete model of how to actually compute the free energy of a structure. Thus, we have to abstract from this general approach and restrict ourselves to simpler methods. In the following, we present one such method.

13.3.1 De Novo Structure Prediction — The HP Model

A three-dimensional model of a protein implies fixing in space the positions of the single atoms and, thus, also the windings, angles, and the lengths of bonds within the molecule. To evaluate such a spatial structure, we usually refer to its free energy. As stated above, a structure with minimal free energy is desired. So, our goal is to determine a three-dimensional structure with low free energy. The many unknown parameters constitute the main problem of such a model. For this purpose, we introduce several simplifications to achieve an abstract model of the problem.

First, we consider an abstraction of a protein structure in space. Instead of referring to single atoms of the protein, we restrict our focus to a higher level by considering amino acids as single units of identical size. We furthermore approximate the spatial structure by means of grid lattices of two or three dimensions, i.e., we consider a region in $\mathbb{Z}^d$, $d \in \{2, 3\}$.

Definition 13.19. *The* d-dimensional grid lattice *is the infinite graph* $\mathcal{L}_d = (V, E)$ *with vertex set* $V = \mathbb{Z}^d$ *and edge set* $E = \{\{x, x'\} \mid x, x' \in \mathbb{Z}^d, \|x - x'\|_2 = 1\}$, *where* $\|\cdot\|_2$ *denotes the Euclidean norm.*

It remains for us to define how a valid spatial folding of a protein in this setting may look.

Definition 13.20. *Let* $s = s_1 \dots s_n$ *be a string corresponding to a primary structure of a protein. A mapping* $\xi : \{1, \dots, n\} \to \mathcal{L}_d$ *is called an* embedding[8] *of* s *into the grid* $\mathcal{L}_d$ *if* ξ *satisfies the following properties:*

(i) All symbols that are neighbors in s *are also adjacent in the grid, i.e., for all* i, $1 \leqslant i < n$,

$$\|\xi(i) - \xi(i+1)\| = 1,$$

where $\|\cdot\|$ *denotes the Euclidean norm.*

(ii) No two positions in s *are assigned to the same vertex in the grid, i.e., for all* i, j, $i \neq j$,

$$\xi(i) \neq \xi(j).$$

Thus, amino acids can be placed on the vertices in the grid structure, and those in neighboring positions in the sequence must be placed onto adjacent vertices in the grid. As a result, angles between consecutive amino acids along the polypeptide chain are restricted to multiples of 90°, and, accordingly, the lengths of the bonds correspond to the edge lengths in the grid. Such an embedding of a protein into a grid can also be considered a *self-avoiding walk*.

Having described our simplified structural model, we can now propose an estimation of the free energy for it. As mentioned in the previous section, we can classify amino acids according to certain chemical properties of their side chains. In particular, we may distinguish between hydrophobic and polar (hydrophilic) amino acids (see Table 2.1). Experimental studies have led to the conjecture that hydrophobic interactions are the essential driving force in the folding process of the molecule, where hydrophobic amino acids form inner cores of the molecule, while hydrophilic amino acids shield cores from the surrounding medium. Hydrophobic interactions inside the hydrophobic core thus essentially contribute to the stability of the molecule. Therefore, we measure the free energy of our model in terms of hydrophobic-hydrophobic interactions. We assume that two hydrophobic amino acids are interacting if they are embedded into adjacent positions of the grid, but not in neighboring positions in the primary structure. We will call such an interaction a *contact edge* in our grid model. Instead of "minimizing" this free energy, we search for a structure maximizing the overall number of the contact edges.

Since we restrict ourselves to the consideration of hydrophobic interactions in this model, we can treat the primary structure as a string $s \in \{H, P\}^*$,

[8] In graph theory, embedding has a more general meaning, but we will not elaborate on this here.

where H represents hydrophobic and P represents polar amino acids.[9] For this reason, the model is also called *HP model*. We will further simplify our notion and replace characters H and P by 1 and 0, respectively. With respect to Definition 13.20, an embedding thus assigns 0s and 1s to certain vertices of a grid.

Usually, we use grids of dimension $d = 3$ or $d = 2$ in our structural model. An example of the embedding of the string $s = 0110101001000001001$ into a two-dimensional grid is illustrated in Figure 13.16.

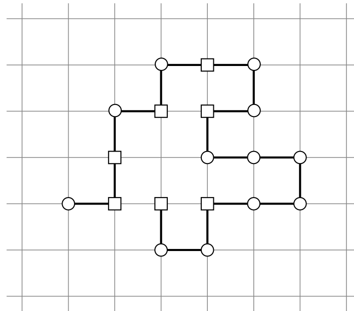


Fig. 13.16. Embedding of the string $s = 0110101001000001001$ into the two-dimensional grid $\mathcal{L}_2$. Grid vertices labeled 0 are drawn as circles and those labeled 1 are drawn as squares

For the symbols of a string, embedded into a grid, we distingish two types of adjacency.

Definition 13.21. *Let $s = s_1 \dots s_n$ be a string over the alphabet $\{0, 1\}$ and let ξ be an embedding of s into a grid $\mathcal{L}_d$. We denote two positions i and j, $1 \leqslant i, j \leqslant n$ as* connected neighbors *if $|i - j| = 1$, i.e., if they are neighbors in s.*

On the other hand, we denote two positions i and j, $1 \leqslant i, j \leqslant n$, as topological neighbors *if they are not connected neighbors but satisfy*

$$\|\xi(i) - \xi(j)\| = 1.$$

Hence, topological neighbors are the non-neighboring positions in the string that become adjacent when embedded into the grid according to ξ.

By $\mathsf{ones}(s) = \{i \mid s_i = 1\}$ we denote the set of 1-positions in s. Similarly, we denote by $\mathsf{zeros}(s) = \{i \mid s_i = 0\}$ the set of 0-positions in s.

A pair $\{i, j\}$ of 1-positions, $i, j \in \mathsf{ones}(s), i \neq j$, forms a contact edge *according to ξ if $\xi(i)$ and $\xi(j)$ are topological neighbors. For a single position i in s, we use the term* contact *to refer to its incident contact edges.*

[9] We also abstract from the fact that there are other characteristics of amino acids, and refer only to the degree of hydrophobicity.

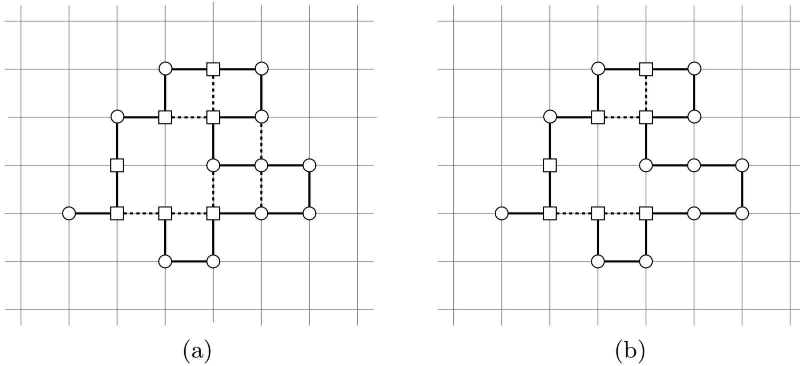


Fig. 13.17. (a) All pairs of topological neighbors for the embedding of s = 0110101001000001001 into a two-dimensional grid from Figure 13.16; (b) all contact edges

See Figure 13.17 for an illustration of the notions introduced in Definition 13.21. Note that the number of contacts for an embedding ξ is exactly twice the number of contact edges, since each contact edge contributes exactly one contact at each of its two endpoints.

As discussed above, the number of contact edges (or, equivalently, the number of contacts) will be used to measure the free energy of the spatial structure of the protein. This leads to the following optimization problem.

Definition 13.22. *Given a grid lattice $\mathcal{L}_d$ of dimension d, the* HP(d) problem *is defined as follows:*

Input: A string $s = s_1 \dots s_n$ over the alphabet $\{0, 1\}$.
Feasible solutions: Each embedding ξ of s into $\mathcal{L}_d$.
Cost: For each feasible solution ξ, the cost is the number of contact edges induced by ξ, i.e.,

$$cost(\xi) = |\{\{i, j\} \in E(\mathcal{L}_2) \mid \{i, j\} \text{ is a contact edge}\}|.$$

Optimization goal: Maximization.

For this problem, it is simply a question of convenience whether we refer to the number of contacts or the number of contact edges. We typically refer to contact edges, while sometimes counting the number of contacts, especially in the context of algorithms, appears to be favorable.

In what follows, we essentially consider the problem of predicting a protein structure for the two-dimensional case, i.e., for the HP(2) problem. Clearly, this is a strong restriction with respect to our original intention to predict the spatial structure of proteins, but it also has some advantages. First, it provides a good starting point to introduce some basic concepts for solving the problem, which may later be applied for the three-dimensional case as

well. So, it often appears to be a good heuristic[10] to stack solutions for the two-dimensional case on top of each other to finally obtain a solution for the three-dimensional case (see Figure 13.18).

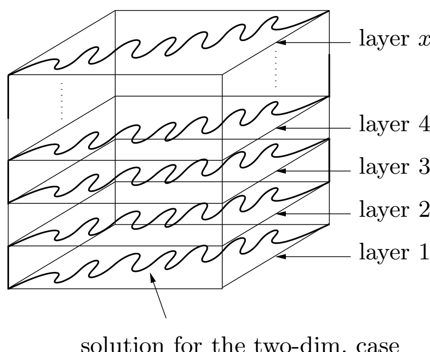


Fig. 13.18. Stacking feasible solutions for the HP(2) problem on top of each other yields a feasible solution for the HP(3) problem

Second, results for the two-dimensional case may reveal important insights about the molecule's structure that may be useful in pratice.

The following result concerning the complexity of the HP problem was shown by Berger and Leighton [27], and Crescenzi et al. [52].

Theorem 13.5. *The HP(d) problem is NP-hard for* $d = 2, 3$. □

We do not elaborate on this result here, but instead present an approximation algorithm for the problem.

Approximation Algorithm for the HP(2) Problem

Before we start with the description of the concrete algorithm, let us first remark on the general design of approximation algorithms useful in our context. In principle, an approximation algorithm may depend on any algorithmic concept or method, but to actually *prove* a good performance, i.e., to determine its approximation ratio, we have to compare the computed output to an optimal solution. Since we are usually unable to compute an optimal solution due to the hardness of the problem and hence have no idea of its cost, there is a need to bound the cost of the optimal solution. Moreover, this bound, or the structure this bound depends on, is often used in the approximation algorithm itself as a first step for the computation. As an illustration, let us consider the approximation algorithm for the metric TSP presented in Section 3.3. Here, the basic idea was that a minimum spanning tree is a lower bound

[10] At least with respect to the achieved approximation ratio.

on the optimal cost of our requested Hamiltonian tour. (If you delete one edge from such a tour, you directly obtain a spanning tree; hence, the minimum spanning tree is a bound on the cost of the Hamiltonian tour, since we only have positive edge weights.) Moreover, as also sketched above, the minimum spanning tree serves as the basic structure in our approximation algorithm from which we eventually derive a Hamiltonian tour with good performance, providing in this case an approximation ratio of 2.

Thus, our first goal is to find an upper bound on the cost of an optimal solution for our HP(2) problem. For this purpose, the following observations are helpful.

Lemma 13.1. *Let $s = s_1 \dots s_n$ be a string over the alphabet $\{0,1\}$ and let ξ be an embedding of s into $\mathcal{L}_2$. Then,*

(i) each position i, $2 \leqslant i \leqslant n-1$, may have at most two contacts, and positions $i = 1$ and $i = n$ may have at most three contacts in the embedding. Moreover,
(ii) for each contact edge $\{i, j\}$, the parities of i and j are different.

Proof. Claim (i) is clear from the fact that each vertex in a two-dimensional grid is adjacent to exactly four other vertices, and that exactly two of them are occupied by neighoring positions in the string (connected neighbors) for all inner positions i, $2 \leqslant i \leqslant n-1$, in the string. Thus, at most two connections remain free to establish contact edges. For the end positions $i = 1$ and $i = n$ of s the same argument holds, except that we have only one (instead of two) connected neighbors.

To prove claim (ii), we use the fact that $\mathcal{L}_2$ is bipartite.[11] To illustrate this, we refer to a coloring argument (different colors indicate the two partitions). If we color each vertex in the grid by exactly one of two colors, such that adjacent vertices have different colors, we obtain an assignment that resembles a chess board. Let us consider a contact edge $\{i, j\}$ and its corresponding vertices in the grid $\xi(i) = (x_1, y_1)$ and $\xi(j) = (x_2, y_2)$. Since adjacent vertices in the grid have different colors, along any path from (x_1, y_1) to (x_2, y_2) the colors alternate. Since (x_1, y_1) and (x_2, y_2) form a contact edge and are thus topological neighbors, they have different colors. As a consequence, each path from (x_1, y_1) to (x_2, y_2) must start with one color and end with the other color. Hence, each path from (x_1, y_1) to (x_2, y_2), in particular the one induced by embedding ξ, has an odd number of edges, which implies that i is even if and only if j is odd. □

Essentially, the same result holds for the three-dimensional case as well. There, the number of possible contacts for each position is bounded by 4 and the parity constraint in (ii) holds as well.

---

[11] A graph is bipartite if its vertex set can be partitioned into two sets V_1 and V_2, such that edges only exist between vertices from V_1 and V_2, but not inside V_1 or V_2.

Next, we utilize the parity constraint given by Lemma 13.1 (ii), together with the immediate observation in Lemma 13.1 (i), to establish an upper bound on the maximal number of contacts that can be achieved for an input to the HP(2) problem. We first introduce some additional notation.

Definition 13.23. *For a string $s = s_1 \dots s_n$ over the alphabet $\{0, 1\}$, we can partition the set* $\mathsf{ones}(s)$ *into the sets* $\mathsf{odds}(s)$ *and* $\mathsf{evens}(s)$*, referring to the set of odd and even 1-positions in s, respectively. (Note that we only partition the set of 1-positions according its parity and completely disregard the set of 0-positions.)*

For position sets $\alpha, \beta \in \{\mathsf{odds}(s), \mathsf{evens}(s)\}$, $\alpha \neq \beta$, we define $\alpha \preceq \beta$ if

- *either $|\alpha| < |\beta|$,*
- *or $|\alpha| = |\beta|$, and β contains at least as many end positions from s as α.*

Without loss of generality, we usually assume $\mathsf{odds}(s) \preceq \mathsf{evens}(s)$. We are now ready to prove the following result.

Theorem 13.6. *Let $s = s_1 \dots s_n$ be a string over the alphabet $\{0, 1\}$. Let* $\mathrm{Opt}(s)$ *be an optimal solution for the HP(2) problem, let $\mathsf{odds}(s) \preceq \mathsf{evens}(s)$, and let $t \in \{0, 1, 2\}$ denote the number of end positions in* $\mathsf{ones}(s)$*; then,*

$$cost(\mathrm{Opt}(s)) \leqslant 2 \cdot |\,\mathsf{odds}(s)| + t,$$

where $cost(\cdot)$ refers to the number of contact edges.

Proof. The theorem is a direct consequence of the observations given in Lemma 13.1. As $\mathsf{odds}(s) \preceq \mathsf{evens}(s)$, the number of possible contact edges is restricted to those established by positions in $\mathsf{odds}(s)$. Since each inner position may have at most two incident contact edges, and only end positions can contribute an additional third one, we obtain the desired result. □

We have now established an upper bound on the cost of an optimal solution for our problem. Next, we present an approximation algorithm for the HP(2) problem based on the computation of a *folding point*.

First, we can fix two adjacent columns of the grid and denote them by c_l and c_r. The idea is to determine an appropriate position *fp* in the input string s, and to fold the part of s on its left-hand side into the left-hand part of the grid bordered on the right by column c_l, and to fold the part of s on its right-hand side into the right-hand part of the grid bordered on the left by column c_r. The folding point *fp* thus indicates the position in s where the border between c_l and c_r is crossed. To establish as many contact edges as possible, 1s on the left-hand side of *fp* are arranged at column c_l if possible. Similarly, 1s on the right-hand side of *fp* are arranged at column c_r if possible. The idea is to align the 1s on c_l and c_r in such a way as to yield a high number of contact edges.

As only 1s at odd positions may pair with 1s at even positions, in our concrete scenario it seems to be useful to arrange 1s in $\mathsf{odds}(s)$ on column

c_l and 1s in $\mathsf{evens}(s)$ on column c_r. Clearly, as 1s at odd and even positions are potentially interweaved, we cannot hope to find a folding point that in fact separates them. The idea is to look for such a position that guarantees a considerable number of odd 1s in one and even 1s in the other column. We will formalize this idea in the following definition.

Definition 13.24. *Let $s = s_1 \dots s_n$ be a string over $\{0,1\}$ and let $\mathsf{odds}(s) \preceq \mathsf{evens}(s)$. For a position x, we set*

- $\mathsf{odds}_{|\leqslant x}(s) = \{i \in \mathsf{odds}(s) \mid i \leqslant x\}$ *and* $\mathsf{odds}_{|>x}(s) = \{i \in \mathsf{odds}(s) \mid i > x\}$,
- $\mathsf{evens}_{|\leqslant x}(s) = \{i \in \mathsf{evens}(s) \mid i \leqslant x\}$ *and* $\mathsf{evens}_{|>x}(s) = \{i \in \mathsf{evens}(s) \mid i > x\}$.

A position $fp \in \{1, \dots, n\}$ in s is called a folding point *of s if either*

(i) $|\mathsf{odds}_{|\leqslant fp}(s)| \geqslant \frac{|\mathsf{odds}(s)|}{2}$ *and* $|\mathsf{evens}_{|>fp}| \geqslant \frac{|\mathsf{odds}(s)|}{2}$, *or*
(ii) $|\mathsf{evens}_{|\leqslant fp}(s)| \geqslant \frac{|\mathsf{odds}(2)|}{2}$ *and* $|\mathsf{odds}_{|>x}(s)| \geqslant \frac{|\mathsf{odds}(s)|}{2}$.

Let us assume, without loss of generality, that Condition (i) holds in Definition 13.24. Then, we denote by $\overleftarrow{\mathsf{odds}}(s)$ an arbitrary subset of $\mathsf{odds}_{|\leqslant fp}(s)$ of size $\frac{|\mathsf{odds}(s)|}{2}$, and by $\overrightarrow{\mathsf{evens}}(s)$ an arbitrary subset of $\mathsf{evens}_{|>fp}$ of size $\frac{|\mathsf{odds}(s)|}{2}$. Positions in $\overleftarrow{\mathsf{odds}}(s)$ thus lie on the left-hand side of the folding point fp, while positions in $\overrightarrow{\mathsf{evens}}(s)$ lie on its right-hand side, and both contain exactly $\frac{|\mathsf{odds}(s)|}{2}$ elements. For the case where condition (ii) holds, we refer to the sets $\overleftarrow{\mathsf{evens}}(s)$ and $\overrightarrow{\mathsf{odds}}(s)$, with their obvious meanings.

Now, we arrange the positions in $\overleftarrow{\mathsf{odds}}(s)$ and $\overrightarrow{\mathsf{evens}}(s)$ in two adjacent columns c_l and c_r, such that each position in $\overleftarrow{\mathsf{odds}}(s)$ is aligned to a position in $\overrightarrow{\mathsf{evens}}(s)$. Since there is always an odd number of positions between two arbitrary positions in $\overleftarrow{\mathsf{odds}}(s)$ [$\overrightarrow{\mathsf{evens}}(s)$], particularly between consecutive positions, we can arrange the positions such that the elements are vertically spaced by exactly one grid vertex. If between consecutive positions in $\overleftarrow{\mathsf{odds}}(s)$ [$\overrightarrow{\mathsf{evens}}(s)$] more than one position in s exists, we can arrange them in horizontal U-shaped side arms.

Let us illustrate this procedure with an example.

Example 13.4. We consider the following string s of length 36:

001001001101101001100001110110010111.

There are eight 1-positions with odd index in s,

$$00\boxed{1}00100\boxed{1}101\boxed{1}0\boxed{1}001\boxed{1}00001\boxed{1}101\boxed{1}00101\boxed{1}1,$$

and ten 1-positions with even index,

$$00100\boxed{1}001\boxed{1}0\boxed{1}10100\boxed{1}10000\boxed{1}1\boxed{1}0\boxed{1}100\boxed{1}0\boxed{1}1\boxed{1}.$$

We thus obtain the sets

$$\mathsf{odds}(s) = \{3, 9, 13, 15, 19, 25, 29, 35\},$$

and

$$\mathsf{evens}(s) = \{6, 10, 12, 18, 24, 26, 28, 32, 34, 36\}.$$

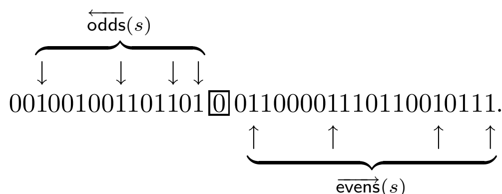
A potential folding point for s (according to Definition 13.24) is $fp = 16$, yielding the sets

$$\overleftarrow{\mathsf{odds}}(s) = \{3, 9, 13, 15\} = \mathsf{odds}_{|\leqslant fp}(s)$$

and

$$\overrightarrow{\mathsf{evens}}(s) = \{18, 24, 32, 36\} \subseteq \{18, 24, 26, 28, 32, 34, 36\} = \mathsf{evens}_{|>fp}(s),$$

i.e.,



We assign these positions in $\overleftarrow{\mathsf{odds}}(s)$ and $\overrightarrow{\mathsf{evens}}(s)$ to adjacent columns of the grid, namely, to columns c_l and c_r, pairwise facing each other (see Figure 13.19 (a)). In this way, we achieve four contact edges. The remaining substrings between these positions are folded into horizontal side arms (see Figure 13.19 (b)). ♢

This exemplary illustration of the proposed embedding should be sufficient and even more intuitive than an explicit and cumbersome formal index-based description of the folding. We therefore omit a precise description here and just summarize the basic steps of our procedure in Algorithm 13.7.

It remains for us to specify the computation of the folding point in step 1 of Algorithm 13.7. Let us again assume that $\mathsf{odds}(s) \preceq \mathsf{evens}(s)$ (the other case is analogous).

If the number of elements in $\mathsf{odds}(s)$ is even, then we determine position i in $\mathsf{odds}(s)$ where $|\,\mathsf{odds}_{|\leqslant i}(s)| = \frac{|\,\mathsf{odds}(s)|}{2}$. Consequently, position $fp = i + 1$ is a valid folding point, since on both sides of fp there are the same number of elements in $\mathsf{odds}(s)$, namely, $\frac{|\,\mathsf{odds}(s)|}{2}$, and on one of the two sides are at least $\frac{|\,\mathsf{odds}(s)|}{2}$ elements from $\mathsf{evens}(s)$, because $\mathsf{odds}(s) \preceq \mathsf{evens}(s)$.

If the number of elements in $|\,\mathsf{odds}(s)|$ is odd, then we choose the median position i in $\mathsf{odds}(s)$, i.e., i is chosen such that $|\,\mathsf{odds}_{|\leqslant i}(s)| = \frac{|\,\mathsf{odds}(s)|+1}{2}$. On the left-hand side of i, including i, there are thus at least $\frac{|\,\mathsf{odds}(s)|}{2}$ positions from $\mathsf{odds}(s)$. If there are at least $\frac{|\,\mathsf{odds}(s)|}{2}$ many positions from $\mathsf{evens}(s)$ on the

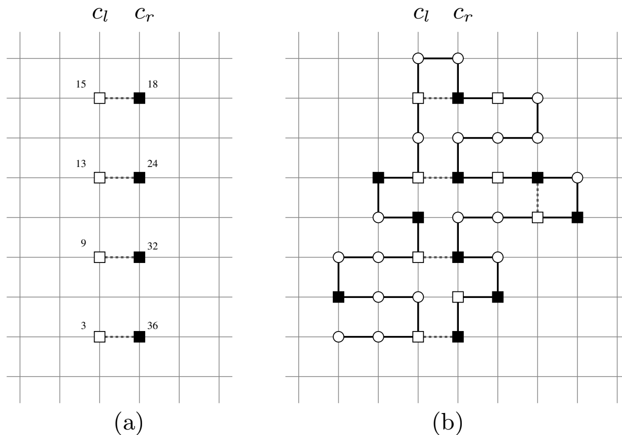


Fig. 13.19. An embedding of s = 001001001101101001100001110110010111 from Example 13.4 using $fp = 16$ as a folding point. Positions in $\mathsf{odds}(s)$ are shown as white boxes and positions in $\mathsf{evens}(s)$ are shown as black boxes

---

Algorithm 13.7 Approximating the HP(2) problem

Input: A string $s = s_1 \dots s_n$ over the alphabet $\{0, 1\}$. Let w.l.o.g. $\mathsf{odds}(s) \preceq \mathsf{evens}(s)$. (The other case is analogous.)

1. Compute a folding point fp of s.
2. Compute position sets γ und δ where
   - either $\gamma = \overleftarrow{\mathsf{odds}}(s)$ and $\delta = \overrightarrow{\mathsf{evens}}(s)$,
   - or $\gamma = \overleftarrow{\mathsf{evens}}(s)$ and $\delta = \overrightarrow{\mathsf{odds}}(s)$.
3. Arrange the positions in γ and δ in two adjacent columns of the grid, such that each position in γ forms a contact edge with a position in δ.
4. Connect the fixed positions in terms of horizontal U-shaped side arms.

Output: The embedding ξ of s into $\mathcal{L}_2$ induced by the above steps.

---

right-hand side of i, then $fp = i$ is a valid folding point. Otherwise, we choose $fp = i - 1$ as a folding point, since on the left-hand side of $i - 1$, including $i - 1$, there have to be at least $\frac{|\mathsf{odds}(s)|}{2}$ positions from $\mathsf{evens}(s)$, and on the right-hand side of $i - 1$ there are at least $\frac{|\mathsf{odds}(s)|}{2}$ positions from $\mathsf{odds}(s)$.

We next analyze the performance guarantee given by Algorithm 13.7.

Theorem 13.7. *Let $s = s_1 \dots s_n$ be an input for the HP(2) problem, and let $\alpha, \beta \in \{\mathsf{odds}(s), \mathsf{evens}(s)\}$, $\alpha \neq \beta$ and $\alpha \preceq \beta$. Then, Algorithm 13.7 is a $\left(\frac{4|\alpha|+2t_\alpha}{|\alpha|-2}\right)$-approximation algorithm for the HP(2) problem, where t_α denotes the number of end positions of s in α.*

Proof. Let $\mathrm{Opt}(s)$ be an optimal solution for the HP(2) problem with input s. Let ξ be the embedding of s into $\mathcal{L}_2$ computed by Algorithm 13.7. The idea of

Algorithm 13.7 is to align $\frac{|\alpha|}{2}$ positions in α to the same number of positions in β. However, in the worst case, the folding point *fp* might be chosen in such a way that *fp* belongs to $\overleftarrow{\alpha}$ and *fp* +1 belongs to $\overrightarrow{\beta}$, or vice versa. Then, the edge between positions *fp* and *fp* +1 does not establish a contact edge, since it connects neighbors. Therefore, we can only guarantee $\frac{|\alpha|-1}{2}$ contact edges for the solution computed by Algorithm 13.7.

Let Opt(s) be an optimal solution for the HP(2) problem with input s. By Theorem 13.6 we know that the number of contact edges in an Opt(s) is bounded by at most $2|\alpha| + t_\alpha$.

For the approximation ratio, this yields

$$\frac{cost(\mathrm{Opt}(s))}{cost(\xi)} \leqslant \frac{2|\alpha| + t_\alpha}{|\alpha|/2 - 1} = \frac{4|\alpha| + 2t_\alpha}{|\alpha| - 2}.$$

□

The above analysis also accounts for two borderline effects; the possibility of the occurrence of connected neighbors around the folding point and the fact that 1-positions at the ends of the input string might yield more contacts than inner 1-positions. Figure 13.20 illustrates the influence of 1-positions at the end of the string. Here, the set odds(s) consists of the end positions 1 and n of a string s that will form three contact edges with each position from evens(s) in an optimal solution (see Figure 13.20 (a)), while in the solution computed by Algorithm 13.7 only one contact edge is established (see Figure 13.20 (b)). We thus obtain only a 6-approximation in this case.

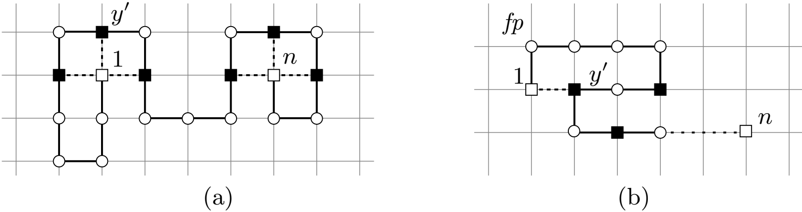


Fig. 13.20. An input indicating that Algorithm 13.7 obtains only a 6-approximation due to a borderline effect with respect to 1-positions at the ends of the input string. (a) A possible optimal solution; (b) the solution computed by the algorithm

For input instances with a large number of potential contact edges, the borderline effects are of limited consequence, and eventually tend to zero if the number of contact edges tends to infinity.

To analyze the approximability of Algorithm 13.7, ignoring the borderline effects, which clearly is meaningful in our context, we introduce the notion of *asymptotic approximation.*

Definition 13.25. *Let $\mathcal{U} = (\mathcal{I}, \mathcal{M}, cost, goal)$ be an optimization problem, let A be a consistent algorithm for $\mathcal{U}$, and let $R_A(I)$ denote the approximation ratio guaranteed by A for instance I (according to Definition 3.17). The* asymptotic approximation ratio *of A is defined by*

$$R_A^\infty = \inf\{\delta \geqslant 1 \mid \; R_A(I) \leqslant \delta \text{ for all } I \in \mathcal{I} \text{ satisfying } cost(\mathrm{Opt}(I)) \geqslant n \\ \text{for some } n \in \mathbb{N}\},$$

where $\mathrm{Opt}(I)$ *denotes an optimal solution for I.*

For every number $\delta > 1$, we say that A is an asymptotic δ-approximation algorithm *for $\mathcal{U}$ if*

$$R_A^\infty \leqslant \delta.$$

Following our above discussion, we obtain the next result.

Theorem 13.8. *Algorithm 13.7 is an asymptotic 4-approximation algorithm for the HP(2) problem.*

Proof. Algorithm 13.7 achieves an approximation ratio of

$$\frac{cost(\mathrm{Opt}(s))}{cost(\xi)} \leqslant \frac{2|\alpha| + t_\alpha}{|\alpha|/2 - 1} = \frac{4|\alpha| + 2t_\alpha}{|\alpha| - 2},$$

according to the proof of Theorem 13.7. With respect to asymptotic approximation, we may let $cost(\mathrm{Opt}(s))$ tend to infinity, and thus constants like $2t_\alpha$ and -2 can be ignored, eventually yielding our desired result. □

Additionally, we put much effort in the precise construction of the folding point in order to obtain the highest number of achievable contact edges for the folding strategy. On the other hand, a less elaborate procedure would imply a constant number of additional losses, which do not influence the asympotic approximation ratio.

Next, we determine the running time of Algorithm 13.7. We have shown how to compute the folding point using a simple method requiring time in $O(n)$. After determining the folding point, we again can construct the corresponding embedding ξ in linear time by simply placing the positions in the input onto the grid according to the "rules" described in Algorithm 13.7. We achieve the following result.

Theorem 13.9. *Algorithm 13.7 requires a running-time in $O(n)$, for a given string $s \in \{0,1\}^n$.* □

13.3.2 Protein Threading

Having discussed a model for protein prediction in the previous section, we next present another approach from the various ideas to solve this task. We

follow an inverse approach. Instead of computing a tertiary structure from a given primary structure, we try to derive the tertiary structure by aligning the primary structure with already known tertiary structures. This idea is based on the observation that there is a huge number of different primary structures of proteins, but there is only a considerably smaller number of three-dimensional shapes of the molecules discovered so far. In particular, certain regions in the sequence resemble each other in different molecules. To make this idea precise, we consider a primary structure s' of a protein (for which we want to determine its tertiary structure), and a known tertiary structure, together with its primary structure s (stored in a database, for instance). We then compute how well we can embed s' into the tertiary structure of s. This procedure is performed for a huge number of molecules (for instance, for all molecules stored in a particular database), and then the tertiary structure fitting s' best is used as a reasonable hypothesis for the tertiary structure of s'. This general concept is called *protein threading.*

Before we present the single steps of this approach in more detail, we have to specify the particular parameters. We can represent a primary structure by a string, as usual. Additionally, we have to agree on the representation of the known tertiary structure of a protein. Here, we focus on the secondary structure, i.e., on helices and sheets. We refer to these entities as *cores.* Since our model does not distinguish between helices and sheets, but uses the cores as basic entities, we can also interpret other structures such as motifs and domains as cores. Using the notion of cores, we next formally introduce a structural model for tertiary structures of proteins.

Definition 13.26. *Let s be a string (representing a protein). A* structure model *or* core model *of s is a 5-tuple $M = (m, c, \lambda, l_{\min}, l_{\max})$ with the following properties:*

- *The structure of s contains $m \in \mathbb{N}$* core regions *$C_1, \ldots, C_m$.*
- *The length of a core region C_i is given by c_i, and $c = (c_1, \ldots, c_m)$.*
- *Core regions C_i and C_{i+1} are connected via a loop of length λ_i, and $\lambda = (\lambda_0, \ldots, \lambda_m)$.*
- *The minimal length of a loop connecting C_i and C_{i+1} is given by $l_i^{\min}$, and $l_{\min} = (l_0^{\min}, \ldots, l_m^{\min})$.*
- *The maximal length of a loop connecting C_i and C_{i+1} is given by $l_i^{\max}$, and $l_{\max} = (l_0^{\max}, \ldots, l_m^{\max})$.*

Hence, the following must hold for a structure model:

- *$|s| = \lambda_0 + \sum_{i=1}^{m}(c_i + \lambda_i)$, and*
- *$l_i^{\min} \leqslant \lambda_i \leqslant l_i^{\max}$ for all i with $1 \leqslant i \leqslant m$.*

Next, we describe what an embedding of a string s' into a structure model looks like. If we assume that in this kind of alignment gaps can only occur inside loops and not within the actual cores, as this would "destroy" the structure of the core region, then we can describe an embedding by specifying

positions t_i in s' where t_i is assigned to the beginning of the core C_i. This leads to the following definition of this special kind of alignment, which we call *threading*.

Definition 13.27. *Let s be a string, and let $M = (m, c, \lambda, l_{\min}, l_{\max})$ be a structure model of s. Let s' be a string. A* threading *T of s' into M is an m-tuple*

$$T = (t_1, \ldots, t_m),$$

where

- $1 + l_0^{\min} \leqslant t_1 \leqslant 1 + l_0^{\max}$,
- *for all i, $1 \leqslant i < m$: $t_i + c_i + l_i^{\min} \leqslant t_{i+1} \leqslant t_i + c_i + l_i^{\max}$, and*
- $t_m + c_m + l_m^{\min} \leqslant |s'| + 1 \leqslant t_m + c_m + l_m^{\max}$.

To illustrate this definition, a schematic view of a threading is shown in Figure 13.21.

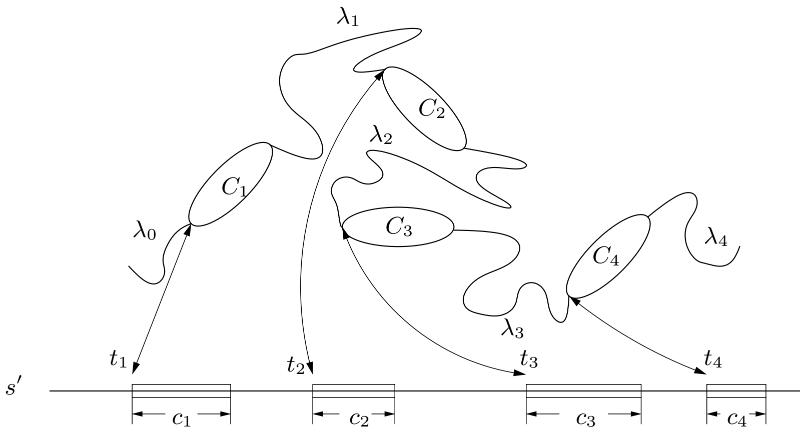


Fig. 13.21. Schematic view of protein threading

The requirements for a threading given in Definition 13.27 mirror the meaning intended by the structure model, namely, that a new core region C_{i+1} can only start if at least $l_i^{\min}$ and at most $l_i^{\max}$ positions in s' are assigned to a loop between the two cores C_i and C_{i+1}. These requirements are also called *ordering constraints*. On the other hand, we may instead require that positions t_i of a threading, for all $1 \leqslant i \leqslant m$, are contained in the following interval

$$1 + \sum_{j<i} (c_j + l_j^{\min}) \leqslant t_i \leqslant |s'| + 1 - \sum_{j \geqslant i} (c_j + l_j^{\min}).$$

We may refer to these inequalities as *spacing constraints*. They represent a weaker requirement than the ordering constrains from which they can be deduced.

How shall we now evaluate such a threading? First of all, it should be clear that we want to evaluate the quality of the assignment of a substring of s' to a certain core C_i given by a threading $(t_1, \ldots, t_m)$. To do so, we use a function $g_1(i, t_i)$ that evaluates the assignment of the substring $s'[t_i, t_i + c_i - 1]$ to the core C_i. Moreover, there might be interactions between several of the core regions, which should be considered as well. Interactions between at most r core regions may be modeled using a function $g_r(i_1, i_2, \ldots, i_r, t_{i_1}, t_{i_2}, \ldots, t_{i_r})$. Clearly, these functions also rely on the particular sequences; nevertheless, we do not refer to these in the notation explicitly, for the sake of convenience. The function values themselves are usually derived using experimental data. Here, we restrict ourselves to interactions between at most two core regions, and thus we define the protein threading problem as follows.

Definition 13.28. *The* protein threading problem *is the following optimization problem.*

Input: A structure model $M = (m, c, \lambda, l_{\min}, l_{\max})$ *of a string* s, *a string* s', *and two functions* $g_1 : \{1, \ldots, m\} \times \{1, \ldots, |s'|\} \to \mathbb{Q}$ *and* $g_2 : \{1, \ldots, m\}^2 \times \{1, \ldots, |s'|\}^2 \to \mathbb{Q}$.
Feasible solutions: All threadings $T = (t_1, \ldots, t_m)$ *of* s' *into* M.
Cost: For all feasible solutions $T = (t_1, \ldots, t_m)$, *the costs are given by*

$$cost(T) = \sum_{i=1}^{m} g_1(i, t_i) + \sum_{i=1}^{m} \sum_{j=i+1}^{m} g_2(i, j, t_i, t_j).$$

Optimization goal: Minimization.

Although this version of the problem is restricted to interactions between at most two cores, it is NP-hard. We prove this result by presenting a reduction from the DecMaxCut problem to the decision version of the protein threading problem.

Definition 13.29. *The* DecMaxCut problem *is the following decision problem.*

Input: A graph $G = (V, E)$ *and a positive integer* k.
Output: Yes *if there exists a partition of the vertex set* V *into two sets* X *and* $X - V$ *such that at least* k *edges are in the cut, i.e.,* $|E \cap \{\{x, y\} \mid x \in X, y \in V - X\}| \geqslant k$. No *otherwise.*

This problem was shown to be NP-hard by Garey et al. [80].

Theorem 13.10. *The* DecMaxCut *problem is NP-hard.* □

Next, we show that the protein threading problem is also NP-hard.

Theorem 13.11. *The protein threading problem is NP-hard.*

Proof. To prove this theorem, we present a polynomial reduction from the DecMaxCut problem to the decision version of the protein threading problem, which, in addition to an input instance (M, s', g_1, g_2) for the protein threading problem, gets an integer h as input and yields Yes if and only if there exists a threading of cost at most h.

Let (G, k) be an input for the DecMaxCut problem, where $G = (V, E)$. From this, we construct a structure model M with $|V| = n$ core elements each of length 1 that are connected to each other by loops of length 0. On the other hand, the possible length of the loops is arbitrary. We thus obtain $M = (n, c, \lambda, l_{\min}, l_{\max})$, where

- $c = (1, \ldots, 1)$,
- $\lambda = (0, \ldots, 0)$,
- $l_{\min} = (0, \ldots, 0)$, and
- $l_{\max} = (\infty, \ldots, \infty)$.

The structure model M hence mirrors the vertices in the graph G. Compared with this, a threading should represent a cut, i.e., a partition of the vertex set; we therefore choose s' as

$$s' = (01)^n.$$

In this way, each core is assigned to either a 0 or a 1 in s' by the threading. It remains for us to transform the size of the implied cut to the evaluation of the threading. To do so, we use the function g_2 scoring the interaction between two core regions, since the size of a cut is measured in the number of edges connecting different parts of the partition. We thus consider the relation between the core regions only, and hence set $g_1(i, t_i) = 0$ for all i and t_i. On the other hand, we score the situation -1 when a pair of cores is assigned to different values, 0 and 1, by the threading and there exists an edge between the corresponding vertices in G. Formally, we set g_2 to

$$g_2(i, j, t_i, t_j) = \begin{cases} -1 \text{ if } i < j,\ \{v_i, v_j\} \in E, \text{ and } s'_{t_i} \neq s'_{t_j} \\ \ \ 0 \text{ otherwise.} \end{cases}$$

An illustration of this reduction is given in Figure 13.22. Hence, a threading of s' into M implies a partition of the vertices in G, and vice versa. The number of edges between the parts of the partition, i.e., the size of the cut, corresponds to the absolute value of the cost of the threading, since each such edge is scored -1 by g_2. Finally, there exists a cut of size k in G if and only if there exists a threading of cost $-k$ for (M, s', g_1, g_2). □

We have used very specific evaluation functions in the proof and have referred to the worst-case complexity only. In practice, the functions g_1 and g_2, as determined experimentally, will not show the property used for the reduction in the proof.

Let us now consider an algorithm that will solve the protein threading problem in an exact way, but that may require exponential running time

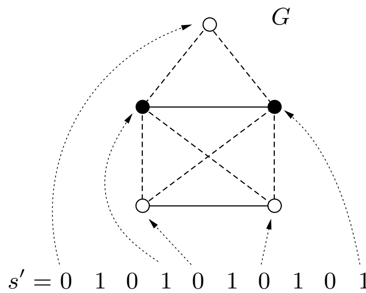


Fig. 13.22. Reduction from DECMAXCUT to the decision version of protein threading. The vertices from G are partitioned into the sets X and $V - X$; vertices in X are drawn using unfilled circles, and vertices in $V - X$ using filled circles. A threading of s' assigns to each vertex in G (core element in M) either the value 0 or the value 1 and induces in this way a partition into X and $V - X$. Edges in the cut are drawn as dashed lines

for some unfavorable inputs. However, it is more intelligent than applying a simple brute force method. The algorithm depends on the *branch and bound* method. This idea is often used for solving NP-hard problems exactly, since it may require exponential running time in the worst case, since in principle it searches the whole set of possible solutions; but, on the other hand, it is able to prune solutions that can be foreseen to be unpromising at an early stage. The general setting is quite similar to the backtracking method we applied to the partial digest problem in Section 7.1.2.

The general procedure is as follows. First, we find a way to stepwise specify a solution with partial solutions, which can be successively developed further, to eventually obtain a complete feasible solution. We thus obtain a partition of the solution space in terms of partial specifications. A good illustration is in terms of a rooted tree. The root corresponds to an empty specification, leaves represent feasible solutions, and inner vertices correspond to partial specifications inside a subtree. Starting from an inner vertex, or the root itself, a further specification of the partial solution corresponds to the transition from a parent vertex to its children, and is called *branching step*. Next, consider the situation where we have already found one feasible solution and thus also know its cost α; in this case, we do not want to branch further into partial solutions which we know in advance will yield worse (in our case, higher) costs. If we now know for an inner vertex that all solutions derived from this specific partial solution, i.e., all solutions inside the subtree rooted at this vertex, will have at least cost $\beta > \alpha$, we may ignore the subtree in the continued search for an optimal solution. To apply this idea of disregarding unfavorable branches of the search tree, we need to estimate the cost of feasible solutions in a subtree in an intelligent way. This is called the *bounding step*.

Let us consider this procedure for our protein threading problem. Let $M = (m, c, \lambda\ , l_{\min}, l_{\max})$ be a structure model. According to the ordering

constraint

$$t_i + c_i + l_i^{\min} \leqslant t_{i+1} \leqslant t_i + c_i + l_i^{\max}$$

for a threading, we can represent the set of feasible threadings $\mathcal{T}$ in terms of an m-tuple of intervals $([b_1, e_1], [b_2, e_2], \ldots, [b_m, e_m])$, where $b_i \leqslant t_i \leqslant e_i$ holds for all $1 \leqslant i \leqslant m$. A branching step may start with choosing one of the intervals, for instance, $[b_j, e_j]$. This interval may then be divided into three subintervals $[b_j, d_j - 1]$,$[d_j, d_j]$, and $[d_j + 1, e_j]$. Here, d_j is an arbitrary value inside the original interval $[b_j, e_j]$. This means that in the second interval we enforce the choice $t_j = d_j$ for the set of consistent threadings. We denote the sets of threadings resulting from such a partition of an original set $\mathcal{T}$ by $\mathcal{T}[b_j, d_j - 1]$, $\mathcal{T}[d_j, d_j]$, and $\mathcal{T}[d_j + 1, e_j]$.

Let us now consider the bounding step of the algorithm, namely, the specification of a lower bound on the cost of all threading represented by an m-tuple of intervals. A simple lower bound is, for instance, the sum of all minimal values of the functions g_1 and g_2 in the valid intervals:

$$\mathit{lowerbound}(([b_1, e_1], \ldots, [b_m, e_m])) =$$
$$\sum_{i=1}^{m} \left(\min_{b_i \leqslant x \leqslant e_i} g_1(i, x) + \sum_{j=i+1}^{m} \min_{\substack{b_i \leqslant y \leqslant e_i \\ b_j \leqslant z \leqslant e_j}} g_2(i, j, y, z) \right).$$

Having specified the branching as well as the bounding step, we recapitulate the procedure in Algorithm 13.8. To traverse the search tree as efficiently as possible, we first investigate those branches that have the lowest bound using a priority queue. The commands Enqueue and Dequeue insert a new element such that the queue remains sorted and return the element with the lowest value, respectively. Here, the sorting criterion used is clearly the lower bound for the set of threadings that can be derived from the partial solution given by the element.

In concluding, we note that the better the lower bound, the larger the number of branches of the solution space that can be ignored, which results in better running times. We refer to such improved lower bound functions in the bibliographic notes in Section 13.5. Moreover, it should be clear that, if we know a good feasible solution in advance, for instance, one computed by a suitable heuristic, we can again prune large parts of the solution space. Therefore, preprocessing to compute such a good solution for use in the algorithm is meaningful in any case.

13.4 Summary

13.4.1 RNA Secondary Structure Prediction

The spatial structure of RNA molecules is partitioned into three structural levels: the primary structure, i.e., the sequence of nucleotides along the RNA

Algorithm 13.8 Protein threading by branch and bound

Input: A structure model $M = (m, c, \lambda, l_{\min}, l_{\max})$ of a string s, a string s', two functions g_1 and g_2, and a lower bound function *lowerbound*.

1. Initialization:

   $opt := \infty$ {Cost of the currently best threading}
   $thr :=$ not defined {Currently best threading}
   $\mathcal{T} := ([0,\infty], \ldots, [0,\infty])$ {Set of all threadings}
   $lb := lowerbound(\mathcal{T})$
   $Q :=$ Enqueue$(Q, (\mathcal{T}, lb))$

2. Branch and Bound:

   while $Q \neq$ NIL **do**
     $(\mathcal{T}_i, lb_i) :=$ Dequeue(Q)
     if $lb_i < opt$ **then**
       if $|\mathcal{T}_i| = 1$ **then**
         let t be the only valid threading in $\mathcal{T}_i$
         if $cost(t) < opt$ **then**
           $opt := cost(t)$
           $thr := t$
       else
         Chose j and d_j with $1 \leqslant j \leqslant m$ and $b_j < d_j < e_j$ for $\mathcal{T}_i$.
         $Q :=$ Enqueue$(Q, (\mathcal{T}_i[b_j, d_j - 1], lowerbound(\mathcal{T}_i[b_j, d_j - 1])))$
         $Q :=$ Enqueue$(Q, (\mathcal{T}_i[d_j, d_j], lowerbound(\mathcal{T}_i[d_j, d_j])))$
         $Q :=$ Enqueue$(Q, (\mathcal{T}_i[d_j + 1, e_j], lowerbound(\mathcal{T}_i[d_j + 1, e_j])))$

Output: An optimal threading *thr* with cost *opt*.

---

strand; the secondary structure, i.e., the set of base pairs established between the nucleotides of the strand; and the tertiary structure, i.e., the actual position of molecules in three-dimensional space. The function of RNA depends on its spatial structure. Consequently, knowledge about this structure is desired. Here, the secondary structure represents a kind of intermediate structure between the primary and tertiary structures.

Bonds between bases may lead to specific types of substructures that are, according to their shape, called stem, hairpin loop, bulge, interior loop, or multiple loop. A special substructure is the pseudoknot, where crossings of base pairs along the RNA strand yield some kind of knot.

One approach to deriving the secondary structure from the primary structure is based on the idea of minimizing the free energy of the corresponding conformation — the lower this energy, the more stable the molecular structure of the molecule. Nussinov's algorithm (Algorithm 13.1) tries to approximate the free energy in terms of maximizing the number of base pairs in the structure. A refined estimation of free energy tries to incorporate the specific substructures in the conformation and their special contribution to the free energy. For this purpose, the algorithm by Zuker and Stiegler (Algorithm

13.2) accounts not only for the number of base pairs but also for the induced substructures. Both algorithms use the method of dynamic programming.

A different approach underlies the modeling of the secondary structure in terms of stochastic context-free grammars. Here, we deal with context-free grammars, where each production is assigned a specific probability. The application of a production $S \rightarrow \texttt{C}B\texttt{G}$ in a derivation of the string representation (primary structure) of an RNA is intended to indicate a base pair $(\texttt{C}, \texttt{G})$. Using the probabilities of the productions, we can assign probabilities also to each derivation, and, in particular, also to each derivation tree. Given such a stochastic context-free grammar and a primary structure of RNA, we can compute the most probable derivation tree for the primary structure using a stochastic variant of the CYK algorithm (Algorithm 13.4); and by the productions used, we gain a hypothesis for the secondary structure of the considered RNA.

13.4.2 Structure-Based Comparison of Biomolecules

RNA secondary structures can be represented in terms of arc-annotated strings, which consist of an underlying string and a set of position pairs in it whose corresponding symbols are connected by an arc. According to the structure implied by these arcs we can distinguish between different classes of arc-annotated strings (Unlimited, Crossing, Nested, Chain, Plain).

To compare molecules with respect to more than their primary structure, we include higher-dimensional structures using arc-annotated strings. This leads to the problem of computing the *longest common subsequence* and its extension to arc-annotated strings, the problem of determining the *longest arc-preserving common subsequence*, or LAPCS for short.

According to the classification of arc-annotated strings, we can consider this problem for different classes. The problem LAPCS(Plain, Plain) here corresponds to the longest common subsequence problem and can be solved using a simple alignment algorithm in polynomial time. By contrast, LAPCS(Crossing,Crossing) was shown to be an NP-hard optimization problem using a reduction from the clique problem. However, there exists a 2-approximation algorithm, which is clearly also applicable to all subproblems, in particular, for the biologically interesting case of LAPCS(Nested,Nested), which can be used to model RNA structures without pseudoknots.

13.4.3 Protein Structure Prediction

The spatial structure of proteins is, similarly to that of RNA, differentiated in a hierarchical system. We distingiush between the primary structure, i.e., the sequence of amino acids, the secondary structure, i.e., helices and sheets, the tertiary structure, i.e., the interplay of several secondary structures, and the quaternary structure, i.e., the assembly of proteins from several molecular subunits.

It is a tedious and expensive task to determine the spatial structure of proteins by experiments. As a consequence, one searches for an approach that allows its computation of from the primary structure of the protein only, which in principle is possible, since the primary structure completely determines the folding of the whole molecule (at least for a large class of proteins).

To infer the spatial structure from the primary one, we require an abstract model of the folding space. One example is a three- or two-dimensional grid lattice, where we try to embed the primary structure in terms of a self-avoiding walk. The free energy of such an embedding can be measured by the number of hydrophobic interactions in the core of the molecule. In the proposed grid model, this corresponds to the number of hydrophobic amino acids assigned to adjacent vertices in the grid. Since each vertex in a two-dimensional grid has at most four neighbors and, moreover, only the vertices connected by a path of odd length can be adjacent, we can easily establish an upper bound on the cost of an optimal solution. Based on the same idea, we can design an approximation algorithm for this problem that asymptotically achieves a ratio of 4.

Another approach to determine the spatial structure of a protein is based on the observation that many structures have similar core regions and only differ in the linkage of these cores. The term core may, for instance, refer to helices or sheet structures. The idea of protein threading is to align a primary structure whose spatial structure is desired to a three-dimensional structure that was resolved before, and that can now be stored in a database. For this, the known three-dimensional structures are represented in terms of a structure model and the quality of an assignment of the primary structure to this structure model is evaluated. The resulting optimization problem was shown to be NP-hard in its general form. We proposed a suitable branch and bound approach to compute an exact solution for the problem.

13.5 Bibliographic Notes

13.5.1 RNA Secondary Structure Prediction

Besides chapters in the monographs by Clote and Backofen [49], Durbin et al. [62], Setubal and Meidanis [180], and Waterman [201], Wang and Zhang [200] give a very nice overview on the methods utilized in the framework of RNA secondary structure prediction.

Figure 13.1 originates from a similar picture in the textbook by Karlson et al. [115] (in German).

Algorithms 13.1 and 13.2 have been proposed in works by Nussinov et al. [151] and Zuker and Stiegler [212], respectively. The running time of Algorithm 13.2 can be improved using a more clever implementation [202]; in particular, one can achieve a running time in $O(n^2)$ under the assumption that the free energy for loops can be modeled in terms of linear functions.

Related approaches are also presented in [201]. Values used in practice for the free energy of the single substructures can be found in [49].

Approaches to include pseudoknots into structure prediction for RNA are studied by Akutsu [5], Lyngsø and Pedersen [141], and Rivas and Eddy [166]. Zuker [211] considered methods to compute good suboptimal structures besides structures with minimum free energy. They may be used as good hypotheses for the secondary structure and undergo further investigation and evaluation.

The description of the modelling of RNA secondary structure prediction by stochastic context-free grammars follows the comprehensive presentation in [62]. An introduction to this concept can also be found in the book by Baldi and Brunak [21]. The construction of a stochastic context-free grammar on the basis of training data can be performed using the inside-outside algorithm [127]. [127]Another approach for determining such a grammar relies on the usage of multiple alignments of RNA primary structures [64].

The CYK algorithm for conventional context-free grammars was independently proposed by Cocke, Younger [209], and Kasami, and can also be found in the book by Hopcroft et al. [103].

Another approach for determining RNA secondary structures is based on considering RNA sequences in closely related organisms. Mutations, in particular base substitutions, often have no effect on the secondary structure since they affect both bases forming one base pair in the secondary structure (therefore, the three-dimensinal shape and, thus, the function of the molecule are conserved). Hence, one considers pairs of bases in the closely related RNA sequences that are changed by the transition of one RNA to another and conjectures a corresponding pair in the secondary structure. Further observations on this approach are found in [200].

13.5.2 Structure-Based Comparison of Biomolecules

The concept of arc-annotated strings was considered by Evans in her PhD thesis in 1999 [67]. In particular, the idea for the NP-hardness proof for LAPCS(Crossing, Crossing) by a reduction from the Clique problem goes back to this publication. The NP-hardness of the class LAPCS(Nested, Nested) has been proven in a work by Lin et al. [139], the 2-approximation algorithm for the LAPCS(Crossing, Crossing) problem was proposed by Jiang et al. [110]. These articles moreover contain many more results concerning the complexity and algorithmic aspects of the problem.

A related approach to describing the structural similarity of proteins was studied by Goldman et al. [83].

In the context of arc-annotated strings, often the concept of *parameterized complexity* is considered. The idea of parameterized complexity is to extract a parameter responsible for the inherent exponential running time of the considered problem (under the assumption P≠NP), and to eventually propose

an exact algorithm with running time that is arbitrary (i.e., possibly exponential) in the extracted parameter but only polynomial in the size of the remaining input. This concept may lead to practical solutions if the parameter for real input instances is small, so that an exponential (in this small parameter) running time does not hurt too much. From this point of view, one may also try to develop parameterized algorithms for certain parameters found to be small in practice or at least meaningful in the specific problem setting. In our case, one such parameter may be the depth of nested arcs or the number of crossings of arcs in arc-annotated strings. Approaches following this line of research can be found in the works by Evans [67], and by Alber et al. [7]. A general introduction to the concept of parameterized complexity can be found in the books by Downey and Fellows [63] and Niedermeier [150], who comprehensively study this concept, and also in the book by Hromkovič [105].

13.5.3 Protein Structure Prediction

Section 13.3 is based in part on descriptions in the books by Clote and Backofen [49] and Setubal and Meidanis [180]. A more comprehensive overview, together with a multitude of further references, is given in the PhD thesis by Pedersen [156].

The evidence that the spatial structure of the protein is completely determined by its primary structure was given by Anfinsen et al. [12].

The grid lattice as a structural model in the context of protein structure prediction, in particular the HP model, was proposed by Dill [59]. Dill et al. [60] also claim that most properties of the three-dimensional model are mirrored by similar properties in the two-dimensional setting. There are many results known for this model. The NP-hardness of the resulting optimization problems was shown by Berger and Leighton [27] and Crescenzi et al. [52]. Algorithm 13.7 for the HP(2) problem was originally presented by Hart and Istrail [101]. They gave a thorough description of the algorithm and also presented an extension of the underlying ideas to the three-dimensional case, where they achieve an $\frac{8}{3}$-approximation algorithm. Newman [148] and Newman and Ruhl [149] improved these results by proposing a 3-approximation algorithm and an $\left(\frac{8}{3} - \varepsilon\right)$-approximation algorithm for the two- and the three-dimensional cases, respectively. With regard to the modeling of the HP problem as a mathematical program, we refer to the survey by Greenberg et al. [86]. Moreover, other types of lattices have been considered in the literature; we examplarily refer the reader to the triangular grids studied by Agarwala et al. [2] and grids with plane diagonals studied by Böckenhauer and Bongartz [34]. Additionally, various further extensions of the original HP model have been proposed; we refer the reader to the survey by Chandru et al. [42] for an overview.

The protein threading problem was proposed by Jones et al. [111]. The approach was further investigated by Lathrop and Smith. Lathrop [128] showed

the NP-hardness of the problem. Together with Smith [129], he proposed the discussed branch and bound approach for solving the problem. The efficient computation of the lower bound function as well as improved versions of it may be found in [129] and [49]. Further results concerning the algorithmic complexity of the problem have been presented by Akutsu and Miyano [6]. The NP-hardness proof presented in this books relies on their work, though Akutsu and Miyano proved an even harder result, namely, that the protein threading problem cannot be approximated arbitrarily well, unless P = NP. Furthermore, they studied approximation algorithms to solve this problem.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: Replacing suffix arrays with enhanced suffix arrays. *Journal of Discrete Algorithms* 2, 2004, pp. 53–86.
2. R. Agarwala, S. Batzoglou, V. Dančík, S. E. Decatur, S. Hannenhalli, M. Farach, S. Muthukrishnan, and S. Skiena: Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the HP model. *Journal of Computational Biology* 4(2), 1997, pp. 275–296.
3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
4. A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *Data Structures and Algorithms*. Addison-Wesley, 1983.
5. T. Akutsu: Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics* 104, 2000, pp. 45–62.
6. T. Akutsu and S. Miyano: On the approximation of protein threading. *Theoretical Computer Science*, 210, 1999, pp. 261–275.
7. J. Alber, J. Gramm, J. Guo, and R. Niedermeier: Towards Optimally Solving the Longest Common Subsequence Problem for Sequences with Nested Arc Annotations in Linear Time. *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, Springer LNCS 2373, 2002, pp. 99–114.
8. F. Alizadeh, R. M. Karp, L. A. Newberg, and D. K. Weisser: Physical mapping of chromosomes: A combinatorial problem in computational biology. *Algorithmica* 13(1/2), 1995, pp. 52–76.
9. F. Alizadeh, R. M. Karp, D. K. Weisser, and G. Zweig: Physical mapping of chromosomes using unique probes. *Journal of Computational Biology* 2(2), 1995, pp. 159–184.
10. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman: A basic local alignment search tool. *Journal of Molecular Biology* 215, 1990, pp. 403–410.
11. S. F. Altschul, T. L. Madden, A. Zhang, Z. Zhang, W. Miller, and D. J. Lipman: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research* 25, 1997, 3389–3402.

12. C. B. Anfinsen, E. Haber, and F. H. White: The kinetics of the formation of native ribonuclease during oxidation of the reduced polypeptide domain. *Proceedings of the National Academy of Science of the USA*, 47, 1961, pp. 1309–1314.
13. A. Apostolico: The myriad virtues of subword trees. In: A. Apostolico, and Z. Galil (eds.): *Combinatorics on Words*. Springer, NATO ASI series vol. 112, 1985, pp. 85–96.
14. A. Apostolico, M. E. Bock, S. Lonardi, and X. Xu: Efficient detection of unusual words. *Journal of Computational Biology* 7(1/2), 2000, pp. 71–94.
15. A. Apostolico and R. Giancarlo: The Boyer-Moore-Galil string searching strategies revisited. *SIAM Journal on Computing* 15, 1986, pp. 98–105.
16. C. Armen and C. Stein: Improved length bounds for the shortest superstring problem (extended abstract). *Proceedings of the Fourth International Workshop on Algorithms and Data Structures (WADS'95)*, Springer LNCS 955, 1995, pp. 494–505.
17. C. Armen and C. Stein: A $2\frac{2}{3}$ superstring approximation algorithm. *Discrete Applied Mathematics* 88, 1998, pp. 29–57.
18. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi: *Complexity and Approximation — Combinatorial Optimization Problems and Their Approximability Properties*, Springer, 1999.
19. V. Bafna, D. Gusfield, G. Lancia, and S. Yooseph: Haplotyping as perfect phylogeny: A direct approach. *Journal of Computational Biology* 10(3–4), 2003, pp. 323–340.
20. V. Bafna and P. A. Pevzner: Sorting by transpositions. *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'96)*, 1996, pp. 614–623. Full version in *SIAM Journal on Discrete Mathematics* 11, 1998, pp. 224–240.
21. P. Baldi and S. Brunak: *Bioinformatics — The Machine Learning Approach*, 2nd edition. MIT Press, 2001.
22. J. P. Barthelemy and A. Guenoche: *Trees and Proximity Representations*. Wiley, 1991.
23. L. E. Baum and T. Petrie: Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Mathematical Statistics* 37, 1966, pp. 1554–1563.
24. M. A. Bender, D. Ge, S. He, H. Hu, R. Y. Pinter, S. Skiena, and F. Swidan: Improved bounds on sorting with length-weighted reversals (extended abstract). *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, 2004, pp. 919–928.
25. A. Ben-Dor, I. Pe'er, R. Shamir, and R. Sharan: On the Complexity of Positional Sequencing by Hybridization. *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, Springer LNCS 1645, 1999, pp. 88–100.
26. G. Benson: Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Research* 27(2), 1999, pp. 573–580.
27. B. Berger and T. Leighton: Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete. *Journal of Computational Biology*, 5, 1998, pp. 27–40.

28. A. Bergeron: A very elementary presentation of the Hannenhalli-Pevzner theory. *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, Springer LNCS 2089, 2001, pp. 106–117.
29. A. Bergeron, J. Mixtacki, and J. Stoye: Reversal distance without hurdles and fortresses. *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM'04)*, Springer LNCS 3104, 2004, pp. 388–399.
30. P. Berman and S. Hannenhalli: Fast sorting by reversal. *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, Springer LNCS 1075, 1996, pp. 168–185.
31. P. Berman, S. Hannenhalli, and M. Karpinski: 1.375-approximation algorithm for sorting by reversals. *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*, Springer LNCS 2461, 2002, pp. 200–210.
32. V. Berry, D. Bryant, T. Jiang, P. Kearney, M. Li, T. Wareham, and H. Zhang: A practical algorithm for recovering the best supported edges of an evolutionary tree. *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, 2000, pp. 287–296.
33. A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis: Linear Approximation of Shortest Superstrings. *Journal of the ACM* 41 (4), 1994, pp. 630–647.
34. H.-J. Böckenhauer and D. Bongartz: Protein folding in the HP model on grid lattices with diagonals (extended abstract). *Proceedings of the 29th International Symposium on Mathematical Foundations of Computer Science (MFCS 2004)*, Springer LNCS 3153, 2004, pp. 227–238.
35. P. Bonizzoni, G. Della Vedova, R. Dondi, and J. Li: The haplotyping problem: An overview of computational models and solutions. *Journal of Computer Science and Technology* 18(6), 2003, pp. 675–688.
36. K. S. Booth and G. S. Lueker: Testing for the consecutive ones property, interval graphs, and graph planarity using *PQ*-tree algorithms. *Journal of Computer and System Sciences* 13, 1976, pp. 335–379.
37. R. S. Boyer and J. S. Moore: A fast string-searching algorithm. *Communications of the ACM* 20(10), 1977, pp. 762–772.
38. P. Buneman: The recovery of trees from measures of dissimilarity. In: F. R. Hodson, D. G. Kendall, and P. Tautu (eds.): *Mathematics in the Archaeological and Historical Sciences*, Edinburgh University Press, 1971, pp. 387–395.
39. N. A. Campbell and J. B. Reece: *Biology.* Pearson Benjamin Cummings, 2005.
40. A. Caprara: Sorting by reversals is difficult. *Proceedings of the First Annual International Conference on Computational Molecular Biology (RECOMB 97)*, 1997, pp. 75–83.
41. H. Carillo and D.J. Lipman: The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics* 48, 1988, pp. 1073–1082.
42. V. Chandru, A. DattaSharma, and V. S. A. Kumar: The algorithmics of folding proteins on lattices. *Discrete Applied Mathematics* 127(1), 2003, pp. 145–161.
43. N. Christofides: *Worst-case analysis of a new heuristic for the travelling salesman problem.* Technical Report, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

44. G. Churchill: Stochastic models for heterogeneous DNA sequences. *Bulletin of Mathematical Biology* 51, 1989, pp. 79–94.
45. M. Cieliebak: *Algorithms and Hardness Results for DNA Physical Mapping, Protein Identification, and Related Combinatorial Problems.* PhD thesis, ETH Zurich, 2003.
46. M. Cieliebak, S. Eidenbenz, and G. J. Woeginger: *Double Digest Revisited: Complexity and Approximability in the Presence of Noisy Data.* Technical Report of the ETH Zurich 382, 2002.
47. M. Cieliebak, S. Eidenbenz, and P. Penna: *Noisy Data Make the Partial Digest Problem NP-hard.* Technical Report of the ETH Zurich 381, 2002.
48. A. Clark: Inference of haplotypes from PCR-amplified samples of diploid populations. *Molecular Biology and Evolution* 7, 1990, pp. 111–122.
49. P. Clote and R. Backofen: *Computational Molecular Biology — An Introduction.* Wiley, 2000.
50. T. H. Cormen, C. E. Leiserson, and R. L. Rivest: *Introduction to Algorithms.* McGraw-Hill, 1990.
51. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: *Introduction to Algorithms*, 2nd edition. McGraw-Hill, 2001.
52. P. Crescenzi, D. Goldman, C. Papadimitriou, A. Piccolboni, and M. Yannakakis: On the complexity of protein folding. *Journal of Computational Biology*, 5, 1998, pp. 423–465.
53. M. Crochemore: An optimal algorithm for computing the repetitions in a word. *Information Processing Letters* 12(5), 1981, pp. 244–250.
54. M. Crochemore and W. Rytter: *Text Algorithms.* Oxford University Press, 1994.
55. B. DasGupta, T. Jiang, S. Kannan, M. Li, and E. Sweedyk: On the complexity and approximation of syntenic distance. *Proceedings of the First Annual International Conference on Computational Molecular Biology (RECOMB 97)*, 1997, pp. 99–108.
56. K. J. Danna, G. H. Sack, and D. Nathans: Studies of simian virus 40 DNA. VII. a cleavage map of the SV40 genome. *Journal of Molecular Biology* 78, 1973, pp. 263–276.
57. M. Dayhoff, R. M. Schwartz, and B. C. Orcutt: A model of evolutionary change in proteins. In: M. Dayhoff (ed.): *Atlas of Protein Sequence and Structure* Vol. 5, National Biomedical Research Foundation, 1978, pp. 345–352.
58. R. Diestel: *Graph Theory* (Graduate Texts in Mathematics; 173), 3rd edition. Springer, 2005.
59. K. A. Dill: Theory for the folding and stability of globular proteins. *Biochemistry*, 24, 1985, p. 1501.
60. K. A. Dill, S. Bromberg, K. Yue, K. M. Fiebig, D. P. Yee, P. D. Thomas, and H. S. Chan: Principles of protein folding — a perspective from simple exact models. *Protein Science*, 4, 1995, pp. 561–602.
61. Z. Ding, V. Filkov, D. Gusfield: A linear-time algorithm for the perfect phylogeny haplotyping (PPH) problem. *Proceedings of the 9th Annual International Conference on Research in Computational Molecular Biology (RECOMB'05)*, Springer LNCS 3500, pp. 585–600.
62. R. Durbin, S. Eddy, A. Krogh, and G. Mitchinson: *Biological Sequence Analysis — Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press, 1998.

63. R. G. Downey and M. R. Fellows: *Parameterized Complexity.* Springer, 1999.
64. S. Eddy and R. Durbin: RNA sequence analysis using covariance models. *Nucleic Acid Research* 22(11), 1994, pp. 2079–2088.
65. J. Ehrlich, D. Sankoff, and J. H. Nadeau: Synteny conservation and chromosome rearrangements during mammalian evolution. *Genetics* 147, 1997, pp. 289–296.
66. E. Eskin, E. Halperin, and R. M. Karp: Efficient reconstruction of haplotype structure via perfect phylogeny. *Journal of Bioinformatics and Computational Biology* 1(1), 2003, pp. 1–20.
67. P. A. Evans: *Algorithms and Complexity for Annotated Sequence Analysis.* Dissertation, University of Victoria, Canada, 1999.
68. W. J. Ewens and G. R. Grant: *Statistical Methods in Bioinformatics — An Introduction.* Springer, 2002.
69. M. Farach, S. Kannan, and T. Warnow: A robust model for finding optimal evolutionary trees. *Algorithmica* 13, 1995, pp. 155–179.
70. J. Felsenstein: Evolutionary trees from DNA sequences: A maximum likelihood approach. *Journal of Molecular Evolution* 17, 1981, pp. 368–378.
71. J. Felsenstein: Statistical inference of phylogenies. *Journal of the Royal Statistical Society A* 146(3), pp. 246–272.
72. D. Feng and R. Doolittle: Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution* 25, 1987, pp. 351–360.
73. V. Ferretti, J. N. Nadeau, and D. Sankoff: Original Synteny. *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, Springer LNCS 1075, 1996, pp. 159–167.
74. W. M. Fitch: Toward defining the course of evolution: minimum change for a specified tree topology. *Systematic Zoology* 20, 1971, 406–416.
75. W. M. Fitch and E. Margoliash: The construction of phylogenetic trees. *Science* 155, 1967, pp. 279–284.
76. A. M. Frieze, F. P. Preparata, and E. Upfal: Optimal Reconstruction of a Sequence from its Probes. *Journal of Computational Biology* 6 (3/4), 1999, pp. 361–368.
77. D. R. Fulkerson and O. A. Gross: Incidence matrices and interval graphs. *Pacific Journal of Mathematics* 15(3), 1965, pp. 835–855.
78. J. Gallant, D. Maier, and J. A. Storer: On finding minimal length superstrings. *Journal of Computer and System Sciences* 20, 1980, pp. 50–58.
79. M. R. Garey and D. S. Johnson: *Computers and Intractability — A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.
80. M. R. Garey, D. S. Johnson, and L. Stockmeyer: Some simplified NP-complete graph problems. *Theoretical Computer Science* 1, 1976, pp. 237–267.
81. A. Gibbons: *Algorithmic Graph Theory.* Cambridge University Press 1985.
82. J. F. Gentleman and R. C. Mullin: The distribution of the frequency of occurrence of nucleotide subsequences, based on their overlap capability. *Biometrics* 45, 1989, pp. 35–52.
83. D. Goldman, S. Istrail, and C. H. Papadimitriou: Algorithmic aspects of protein structure similarity (extended abstract). *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)*, 1999, pp. 512–521.

84. L. Goldstein and M.S. Waterman: Mapping DNA by stochastic relaxation. *Advances in Applied Mathematics* 8, 1987, pp. 194–207.
85. M. C. Golumbic: *Algorithmic Graph Theory and Perfect Graphs.* Academic Press 1980.
86. H. J. Greenberg, W. E. Hart, and G. Lancia: Opportunities for Combinatorial Optimization in Computational Biology. *INFORMS Journal on Computing* 16(3), 2004, pp. 211–231.
87. D. Greenberg and S. Istrail: Physical mapping by STS hybridization: Algorithmic strategies and the challenge of software evaluation. *Journal of Computational Biology* 2(2), 1995, pp. 219–274.
88. L. J. Guibas and A. M. Odlyzko: String overlaps, pattern matching and nontransitive games. *Journal of Combinatorial Theory, Series A* 30, 1981, pp. 183–208.
89. D. Gusfield: Efficient algorithms for inferring evolutionary trees. *Networks* 21, 1991, pp. 19–28.
90. D. Gusfield: Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology* 55, 1993, pp. 141–154.
91. D. Gusfield: *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
92. D. Gusfield: Inference of haplotypes from samples of diploid populations: Complexity and algorithms. *Journal of Computational Biology* 8(3), 2001, pp. 305–323.
93. D. Gusfield: Haplotyping as perfect phylogeny: Conceptual framework and efficient solutions. *Proceedings of the 6th Annual International Conference on Computational Biology (RECOMB'02)*, ACM 2002, pp. 166-175.
94. D. Gusfield: An overview of combinatorial methods for haplotype inference. In: S. Istrail, M. Waterman, and A. Clark (Eds.): *Computational Methods for SNPs and Haplotype Inference*, Springer LNBI 2983, 2004, pp. 9–25.
95. B. V. Halldorsson, V. Bafna, N. Edwards, R. Lippert, S. Yooseph, and S. Istrail: Combinatorial problems arising in SNP and haplotype analysis. *Proceedings of the 4th International Conference on Discrete Mathematics and Theoretical Computer Science (DMTCS'03)*, Springer LNCS 2731, 2003, pp. 26–47.
96. S. Hannenhalli and P. A. Pevzner: Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). *Proceedings of the 27th ACM Symposium on the Theory of Computing (STOC'95)*, 1995, pp. 178–189. Full version in *Journal of the ACM* 46, 1999, pp. 1–27.
97. S. Hannenhalli and P. A. Pevzner: Transforming men into mice (polynomial algorithm for genomic distance problem). *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science (FOCS'95)*, 1995, pp. 581–592.
98. A. Hansen: *Bioinformatik — Ein Leitfaden für Naturwissenschaftler.* Birkhäuser, 2001.
99. F. Harary: *Graph Theory.* Addison-Wesley, 1972.
100. D. Harel and R. E. Tarjan: Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing* 13, 1984, pp. 338–355.

101. W. E. Hart and S. C. Istrail: Fast protein folding in the hydrophobic-hydrophilic model within three-eighths of optimal. *Journal of Computational Biology*, 3 (1), 1996, pp. 53–96.
102. S. Henikoff and J. G. Henikoff: Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the U.S.A.* 89, 1992, pp. 10915–10919.
103. J. E. Hopcroft, R. Motwani, and J D. Ullman: *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley 2001, 2nd edition.
104. J. Hromkovič: *Theoretical Computer Science.* Springer, 2004.
105. J. Hromkovič: *Algorithmics for Hard Problems*, 2nd edition. Springer, 2003.
106. International Human Genome Sequencing Consortium: Initial sequencing and analysis of the human genome. *Nature* 409, Feb. 2001, pp. 860–921.
107. S. Istrail, M. Waterman, and A. Clark (Eds.): *Computational Methods for SNPs and Haplotype Inference*, Springer LNBI 2983, 2004.
108. T. A. Jenkyns: The greedy travelling salesman's problem. *Networks* 9, 1979, pp. 363–373.
109. T. Jiang, P. Kearney, and M. Li: Orchestrating quartets: approximation and data correction. *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science (FOCS'98)*, 1998, pp. 416–425.
110. T. Jiang, G.-H. Lin, B. Ma, and K. Zhang: The longest common subsequence problem for arc-annotated sequences. *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, Springer LNCS 1848, 2000, pp. 154–165.
111. D.T. Jones, W.R. Taylor, and J.M. Thornton: A new approach to protein fold recognition. *Nature*, 358, 1992, pp. 86–89.
112. J. Kärkkäinen and P. Sanders: Simple linear work suffix array construction. *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, Springer LNCS 2719, 2003, pp. 943–955.
113. H. Kaplan, R. Shamir, and R. E. Tarjan: Faster and simpler algorithm for sorting signed permutations by reversals. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, 1997, pp. 344–351.
114. S. Karlin, C. Burge, and A. M. Campbell: Statistical analyses of counts and distributions of restriction sites in DNA sequences. *Nucleic Acids Research* 20(6), 1992, pp. 1363–1370.
115. P. Karlson, D. Doenecke, and J. Koolman: *Kurzes Lehrbuch der Biochemie für Mediziner und Naturwissenschaftler.* Thieme, 14th edition 1994 (in German).
116. R. M. Karp: Reducibility among combinatorial problems. In: R. E. Miller and J. W. Thatcher (eds.): *Complexity of Computer Computations*, Plenum Press 1972, pp. 85–103.
117. R. M. Karp and R. Shamir: Algorithms for Optical Mapping. *Journal of Computational Biology* 7(1/2), 2000, pp. 303–316.
118. P. Kearney: Phylogenetics and the quartet method. In: T. Jiang, Y. Xu, and M. Q. Zhang (eds.): *Current Topics in Computational Molecular Biology*, MIT Press, 2002, pp. 111–133.

119. J. D. Kececioglu: *Exact and approximation algorithms for DNA sequence reconstruction.* PhD Thesis 1991, University of Arizona, Tucson, Technical Report 91-26.
120. J. D. Kececioglu and E. W. Myers: Combinatorial Algorithms for DNA Sequence Assembly. *Algorithmica* 13, 1995, pp. 7–51.
121. J. Kececioglu and D. Sankoff: Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica* 13, 1995, pp. 180–210.
122. J. Kleinberg and D. Liben-Nowell: The syntenic diameter of the space of N-chromosome genomes. In: D. Sankoff and J. H. Nadeau (eds.): *Comparative Genomics*, Kluwer Academic Press, 2000, pp.185–197.
123. D. E. Knuth, J. H. Morris, and V. R. Pratt: Fast pattern matching in strings. *SIAM Journal on Computing* 6(2), 1977, pp. 323-350.
124. R. Ladner, N. Lynch, and A. Selman: A comparison of polynomial time reducibilities. *Theoretical Computer Science* 1, 1975, pp. 103–123.
125. G. Lancia, V. Bafna, S. Istrail, R. Lippert, and R. Schwartz: SNPs problems, complexity, and algorithms. *Proceedings of the 9th Annual European Symposium on Algorithms (ESA'01)*, Springer LNCS 2161, 2001, pp. 182–193.
126. G. M. Landau and J. P. Schmidt: An algorithm for approximate tandem repeats. *Proceedings of the Fourth Annual Symposium on Combinatorial Pattern Matching (CPM'93)*, Springer LNCS 684, 1993, pp. 120–133.
127. K. Lari and S. Young: The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language* 4, 1990, pp. 35–56.
128. R. H. Lathrop: The protein threading problem with sequence amino acid interaction preferences is NP-complete. *Protein Engineering* 7 (9), 1994, pp. 1059–1068.
129. R. H. Lathrop and T. F. Smith: Global optimum protein threading with gapped alignment and empirical pair score functions. *Journal of Molecular Biology* 255, 1996, pp. 641-665.
130. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys (eds.): *The Traveling Salesman Problem.* Wiley, 1985.
131. M. Y. Leung, G. M. Marsh, and T. P. Speed: Over and underrepresentation of short DNA words in herpes virus genomes. *Journal of Computational Biology* 3, 1996, pp. 345–360.
132. V. I. Levenshtein: Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 6, 1966, pp. 707–710.
133. B. Lewin: *Genes VIII.* Pearson Prentice Hall, 2004.
134. M. Li, B. Ma, and L. Wang: Finding similar regions in many strings. *Proceedings of the 31st ACM Annual Symposium on the Theory of Computing (STOC'99)*, 1999, pp. 473–482.
135. M. Li, B. Ma, and L. Wang: Finding similar regions in many sequences. *Journal of Computer and System Sciences* 65(1), 2002, pp. 73–96.
136. D. Liben-Nowell: On the structure of syntenic distance. *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, Springer LNCS 1645, 1999, pp. 50–65.
137. D. Liben-Nowell: Gossip is synteny: Incomplete gossip and the syntenic distance between genomes. *Journal of Algorithms* 43, 2002, pp. 264–283.

138. D. Liben-Nowell and J. Kleinberg: Structural properties and tractability results for linear synteny. *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, Springer LNCS 1848, 2000, pp. 248–263.
139. G-H. Lin, Z.-Z. Chen, T. Jiang, and J. Wen: The longest common subsequence problem for sequences with nested arc annotations (extended abstract). *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*, Springer LNCS 2076, 2001, pp. 444–455.
140. D. J. Lipman and W. R. Pearson: Rapid and sensitive protein similarity searches. *Science* 227, 1985, pp. 1435–1441.
141. R. B. Lyngsø and C. N. S. Pedersen: Pseudoknots in RNA secondary structures. *Proceedings of the Fourth Annual International Conference on Computational Molecular Biology (RECOMB 2000)*, 2000, pp. 201–209.
142. M. G. Main and R. J. Lorentz: An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms* 5, 1984, pp. 422–432.
143. U. Manber and E. W. Myers: Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing* 22(5), 1993, pp. 935–948.
144. E. M. McCreight: A space-economical suffix tree construction algorithm. *Journal of the ACM* 23, 1976, pp. 262–272.
145. M. Middendorf: More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science* 125(2), 1994, pp. 205–228.
146. E. W. Myers: Whole-genome DNA sequencing. *IEEE Computing in Science and Engineering*, 1999, pp. 33–43.
147. S. B. Needleman, and C. D. Wunsch: A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 1970, pp. 443–453.
148. A. Newman: A New Algorithm for Protein Folding in the HP Model. *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, 2002, pp. 876–884.
149. A. Newman and M. Ruhl: Combinatorial Problems on Strings with Applications to Protein Folding. *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN 2004)*, Springer LNCS 2976, 2004, pp. 369–378.
150. R. Niedermeier: *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006.
151. R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman: Algorithms for loop matchings. *SIAM Journal of Applied Mathematics* 35, 1978, pp. 68–82.
152. A. Panconesi and M. Sozio: Fast Hare: A fast heuristic for single individual SNP haplotype reconstruction. *Proceedings of the 4th International Workshop on Algorithms in Bioinformatics (WABI'04)*, Springer LNCS 3240, 2004, pp. 266–277.
153. C. Papadimitriou and K. Steiglitz: *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.
154. W. R. Pearson: Flexible sequence similarity searching with the FASTA3 program package. *Methods in Molecular Biology: Bioinformatics Methods and Protocols* 132, 2000, pp. 185–219.

155. W. R. Pearson and D. J. Lipman: Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the U.S.A.* 85, 1988, pp. 2444–2448.
156. C. N. S. Pedersen: *Algorithms in Computational Biology.* Dissertation, BRICS, 2000.
157. P. A. Pevzner: L-tuple DNA sequencing: computer analysis. *Journal of Biomolecular Structure Dynamics* 7, 1989, pp. 63–73.
158. P. A. Pevzner: DNA physical mapping and alternating Eulerian cycles in colored graphs. *Algorithmica* 13, 1995, pp. 77–105.
159. P. A. Pevzner: *Computational Molecular Biology — An Algorithmic Approach.* MIT Press, 2000.
160. P. A. Pevzner, M. Y. Borodovsky, and A. A. Mironov: Linguistics of nucleotide sequences. I: The significance of deviations from mean statistical characteristics and prediction of the frequencies of occurence of words. *Journal of Biomolecular Structure and Dynamics* 6, 1989, pp. 1013–1026.
161. R. Y. Pinter and S. Skiena: Sorting with length-weighted reversals. *Proceedings of the 13th International Conference on Genome Informatics (GIW'02)*, 2002, pp. 103–111.
162. N. Pisanti and M.-F. Sagot: Further thoughts on the syntenic distance between genomes. *Algorithmica* 34, 2002, pp. 157–180.
163. L. R. Rabiner: A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE* 77(2), 1989, pp. 257–286.
164. R. Rauhut: *Bioinformatik: Sequenz — Struktur — Funktion.* Wiley-VCH, 2001.
165. G. Reinert, S. Schbath, and M. S. Waterman: Probabilistic and statistical properties of words: An overview. *Journal of Computational Biology* 7(1/2), 2000, pp. 1–46.
166. E. Rivas and S. R. Eddy: A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology* 285, 1999, pp. 2053–2068.
167. R. Rizzi, V. Bafna, S. Istrail, and G. Lancia: Practical algorithms and fixed-parameter tractability for the single individual SNP haplotyping problem. *Proceedings of the Second International Workshop on Algorithms in Bioinformatics (WABI'02)*, Springer LNCS 2452, 2002, pp. 29–43.
168. J. Rosenblatt and P. D. Seymour: The structure of homometric sets. *SIAM Journal on Algebraic and Discrete Methods* 3(3), 1982, pp. 343–350.
169. S. M. Ross: *Introduction to Probability Models*, 8th edition. Academic Press 2002.
170. W. Rytter: A correct preprocessing algorithm for Boyer-Moore string searching. *SIAM Journal on Computing* 9, 1980, pp. 509–512.
171. R. K. Saiki, S. Scharf, R. Faloona, K. B. Mullis, G. T. Horn, H. A. Erlich, and N. Arnheim: Enzymatic amplification of beta-globin genomic sequences and restriction site analysis for diagnosis of sickle cell anemia. *Science* 230, 1985, pp. 1350–1354.
172. F. Sanger, S. Nicklen, and A. R. Coulson: DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences of the U.S.A.* 74, 1977, pp. 5463–5467.

173. D. Sankoff and N. El-Mabrouk: Genome Rearrangement. In: T. Jiang, Y. Xu, and M. Q. Zhang (eds.): *Current Topics in Computational Molecular Biology*. MIT Press, 2002.
174. D. Sankoff and J. H. Nadeau: Conserved synteny as a measure of genomic distance. *Discrete Applied Mathematics* 71, 1996, pp. 247–257.
175. B. Schieber and U. Vishkin: On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing* 17, 1988, pp. 1253–1262.
176. W. Schmitt and M. S. Waterman: Multiple solutions of DNA restriction mapping problem. *Advances in Applied Mathematics* 12, 1991, pp. 412–427.
177. G. Schnitger: *Algorithmen der Bioinformatik*. Vorlesungsskript, Johann Wolfgang Goethe-Universität, Frankfurt am Main, 2001.
178. E. Schröder: Vier combinatorische Probleme. *Zeitschrift für Mathematik und Physik* 15, 1870, pp. 361–376.
179. C. Semple and M. Steel: *Phylogenetics*. Oxford University Press, 2003.
180. J. Setubal and J. Meidanis: *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.
181. M. Sipser: *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
182. S. S. Skiena, W. D. Smith, P. Lemke: Reconstructing sets from interpoint distances (extended abstract). *Proceedings of the 6th Annual Symposium on Computational Geometry*, ACM 1990, pp. 332–339.
183. S. S. Skiena G. Sundaram: A partial digest approach to restriction site mapping. *Bulletin of Mathematical Biology* 56(2), 1994, pp. 275–294.
184. T. F. Smith and M. S. Waterman: Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1981, pp. 195–197.
185. R. R. Sokal and C. D. Michener: A statistical method for evaluating systematic relationships. *University of Kansas Scientific Bulletin* 28, 1958, pp. 1409–1438.
186. K. Strimmer and A. von Haeseler: Quartet puzzling: a quartet maximum-likelihood method for reconstructing tree topologies. *Molecular Biology and Evolution* 13(7), 1996, pp. 964–969.
187. Z. Sweedyk: A $2\frac{1}{2}$-Approximation Algorithm for Shortest Superstring. *SIAM Journal on Computing* 29 (3), 1999, pp. 954–986.
188. F. Swidan, M. A. Bender, D. Ge, S. He, H. Hu, and R. Y. Pinter: Sorting by Length-Weighted Reversals: Dealing with Signs and Circularity. *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM'04)*, Springer LNCS 3109, 2004, pp. 32–46.
189. D. L. Swofford and G. J. Olsen: Phylogeny reconstruction. In: D. M. Hillis and C. Moritz (eds.): *Molecular Systematics*, Sinauer Associates, 1990, pp. 411–501.
190. J. Tarhio and E. Ukkonen: A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science* 57, 1988, pp. 131–145.
191. J. S. Turner: Approximation Algorithms for the Shortest Common Superstring Problem. *Information and Computation* 83, 1989, pp. 1–20.
192. E. Ukkonen: A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica* 5, 1990, pp. 313–323.

193. E. Ukkonen: On-line construction of suffix-trees. *Algorithmica* 14, 1995, pp. 249–260.
194. J. van Helden, B. André, and J. Collado-Vides: Extracting regulatory sites from the upstream region of yeast genes by computational analysis of oligonucleotide frequencies. *Journal of Molecular Biology* 281, 1998, pp. 827–842.
195. V. Vassilevska: Explicit Inapproximability Bounds for the Shortest Superstring Problem *Proceedings of the 30th International Symposium on Mathematical Foundations of Computer Science (MFCS 2005)*, Springer LNCS 3618, 2005, pp. 793–800.
196. V. V. Vazirani: *Approximation Algorithms.* Springer, 2001.
197. J. C. Venter et al.: The sequence of the human genome. *Science* 291, Feb. 2001, pp. 1304–1351.
198. A. J. Viterbi: Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. *IEEE Transactions on Information Theory* IT-13, 1967, pp. 260–269.
199. L. Wang and T. Jiang: On the complexity of multiple sequence alignment. *Journal of Computational Biology* 1(4), 1994, pp. 337–348.
200. Z. Wang and K. Zhang: RNA secondary structure prediction. In: T. Jiang, Y. Xu, and M. Q. Zhang (eds.): *Current Topics in Computational Molecular Biology.* MIT Press, 2002.
201. M. S. Waterman: *Introduction to Computational Biology — Maps, Sequences and Genomes.* Chapman & Hall/CRC, 1995.
202. M. S. Waterman and T. F. Smith: Rapid dynamic programming algorithms for RNA secondary structure. *Advances in Applied Mathematics* 7, 1986, pp. 455–464.
203. J. D. Watson: *The Double Helix: A Personal Account of the Discovery of the Structure of DNA.* Athenaeum, 1968.
204. J. D. Watson: *DNA — The Secret of Life.* Arrow Books, 2004.
205. J. D. Watson and F. H. C. Crick: Genetical implications of the structure of deoxyribonucleic acid. *Nature* 171, 1953, pp. 964–967.
206. G. A. Watterson, W. J. Ewens, T. E. Hall, and A. Morgan: The chromosome inversion problem. *Journal of Theoretical Biology* 99, 1982, pp. 1–7.
207. P. Weiner: Linear pattern matching algorithms. *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, 1973, pp. 1–11.
208. M. Yannakakis: Node- and edge-deletion NP-complete problems. *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC'78)*, 1978, pp. 265–274.
209. D. H. Younger: Recognition and parsing of context-free languages in time n^3. *Information and Control* 10(2), 1967, pp. 189–209.
210. Z. Zhang: An exponential example for a partial digest mapping algorithm. *Journal of Computational Biology* 1(3), 1994, pp. 235–239.
211. M. Zuker: On finding all suboptimal foldings of an RNA molecule. *Science* 244, 1989, pp. 48–52.
212. M. Zuker and P. Stiegler: Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. Nucleic Acid Research 9, 1981, pp. 133-148.

Index

$\Omega(f(n))$ 30
$\Theta(f(n))$ 30
λ *see* string, empty
2-factor 209

accepted mutation 95
adenine 10
adjacent 25
algorithm 28
  consistent 32
alignment 81ff.
  compatible 109f.
  global 84ff.
  local 84, 90
  multiple 101ff.
  of two strings 82ff.
    computation 88
  optimal local 90
  score 83
  semiglobal 91
alignment tree 110
alphabet 23
amino acid 7
  hydrophilic 353
  hydrophobic 353
  polar 353
approximation algorithm 32f.
approximation ratio 32
arc 338
arc set 338
asymptotic approximation 362
autocorrelation 224
  polynomial 225
automaton 39

backtracking 135
bad character rule 44
  preprocessing 45
base pair
  reachable 322
  valid 320
Bernoulli string 224
binary tree 28
  directed 28
  undirected 28
binding site 213
bit score 100
BLAST 99ff.
BLOCKS database 97
BLOSUM matrix 97
Boyer–Moore algorithm 49
branch and bound 368
breakpoint 241
bulge 322

C1F *see* consecutive ones form
C1P *see* consecutive ones property
Celera Genomics 120
CG-islands 228, 233
Chain 341
chain termination method 18
character *see* symbol
character matrix 269
  binary 269
child 27
chimeric clone 16, 158, 174
Chomsky normal form 332
chord 26

chromosome 12
clique problem 342
clone 144
clone library 144
cloning 15
CNF *see* Chomsky normal form
column distance graph 161
CompactAddTree problem 266
companion *see* companion column
companion column 293
companion graph 298
companion pair
  equated 293
  negated 293
companion row 293
complete pair matrix 290
component graph 303
compression 177
computing problem 28
concatenation 24
connected component 299, 346
consecutive ones form 146
consecutive ones property 146
  testing for 156
consensus 102, 173
  ε-consensus 199
  distance to 102
consensus string approximation 215
consensus string problem 214
contact 354
  edge 353f.
contig 198
cost function 29
cost measure 54
  logarithmic 54
  uniform 54, 69
coverage 172
  incomplete 174
Crossing 341
cycle 26
  simple 26
cycle cover 190
cycle cover superstring algorithm 191
CYK algorithm 335, 373
  stochastic 335
cytosine 10

database search 97ff.
DDP 127
de novo protein structure prediction 352ff.
Dec(0,1)ShortestSuperseq problem 105
DecDDP 128
decision problem 28
  corresponding 32
DecLAPCS 343
DecMaxCut problem 366
DecMultSPAlign problem 105
DecSCS 182
deletion 19, 83
denaturation 15
deoxyribonucleic acid *see* DNA
derivation 329
  rule *see* production
  tree 333
diagonal run 98
digest 124
diploid 287
disjoint DDP 131
distance 179
distance graph
  for SCS 179
  for the CompactAddTree problem 266
distance set 127
divide and conquer 219
DNA 10
DNA array *see* DNA chip
DNA chip 19
DNA contamination problem 63
DNA fingerprinting 217
DNA sequencing 119ff.
domain 351
double digest approach 124ff.
double digest problem *see* DDP
dynamic programming 84, 104, 231, 314, 324

edge 25
  directed 27
edge labeling 28
edit distance 84
edit graph 88
embedding 353
emission probability 228
Eulerian cycle 26, 202
  computing an 205

Eulerian path 26, 202
ExPerfPhyl problem 270

false negative 20, 145
false positive 20, 145
FASTA 98f.
feasible solution 29
fingerprint 143
fingerprinting 143
fission 249f.
Fitch algorithm 277f.
folding point 359
forcing pattern 294
forcing component 299
forcing graph 298
forcing pair 294
fragment assembly problem 172
fragment conflict 306
fragment conflict graph 306
frequency analysis 224
full digest 125
function
  polynomially bounded 30
function computing problem 29
fusion 249f.

gap 82
gap scoring 94
gap symbol 82
gel electrophoresis 16
gene 12
genome 13
genome rearrangements 237ff.
genotype 287f.
genotype matrix 289
  regular 293
genotypical characters 268
global alignment problem 84
GMinP 159
good suffix rule 44
  preprocessing 49
grammar
  context-free 329
  stochastic context-free 329, 331
graph 25ff.
  bipartite 26
  complete 25
  connected 26
  directed 27
  edge-weighted 28
  Eulerian 203
greedy method 182
greedy superstring algorithm 183
grid lattice 353
guanine 11

hairpin loop 322
Hamiltonian cycle 26
Hamiltonian cycle problem 28, 31
Hamming distance 162, 214, 277
haplotype 287f.
haplotype matrix 289
haplotyping 287ff.
helix 350
heterozygous 288
hidden Markov model 228ff.
high scoring pair 100
hit 100
HMM *see* hidden Markov model
HMM decoding problem 231
homologous genes 239
homology 13
homomorphism 25
homozygous 288
host 15
hot spot 98
HP model 352ff.
HP(d) problem 355
HSP *see* high scoring pair
Human Genome Project 120
hybridization 11, 15
hybridization data 143
hybridization matrix 144

incident 25
indicator function 296
  feasible 296
insert 15
insertion 19, 83
inside-outside algorithm 373
interchromosomal transformations 238
interior loop 323
interval graph 169
intrachromosomal transformations 238
inversion *see* reversal

Landau symbols 30
LAPCS 340

lattice 353
layout 173
lce 217
LCS 338
leaf 27
left-diverse 67
left-symbol 67
letter *see* symbol
Levenshtein distance *see* edit distance
lexicographical order 69
link 199
  weakest 199
local alignment problem 90
local search 281
longest arc-preserving common subsequence problem *see* LAPCS
longest common extension 217
longest common extension problem 217
longest common subsequence problem *see* LCS
longest common substring problem 61
loop 323, 350
lowest common ancestor 218

magic word 214
majority voting 103, 174
mapping
  by hybridization 143ff.
  optical 168
  restriction site- 141, 143
  with non-unique probes 165f.
  with unique probes 146ff.
marker 120, 123
match 83
mate pair 120
maximum compression common superstring problem *see* MCCS
MaxQuartettConsist problem 282
MbH *see* mapping, by hybridization
MCCS 177
median string 214
merge 25
metric 251, 258
MFR problem 309
minimum fragment removal problem *see* MFR problem
minimum SNP removal problem *see* MSR problem
minimum vertex bipartizer problem 310
MinOSR problem 248
MinParTop problem 280
MinSR problem 240
  approximation 245
MinSynteny problem 251
  approximation 254
mismatch 83
motif 351
MSR problem 309
MultConsensusAlign problem 103
multicontig model 198
multicontig problem 199
multiple loop 323
MultSPAlign problem 104

neighbors
  connected 354
  topological 354
Nested 341
non-terminal 329
nondeterminism 31
NP 31
NP-complete 31
NP-hard 31f.
nucleic acid 9ff.
nucleotide 10
Nussinov algorithm 324f.

$O(f(n))$ 30
optimization goal 83
optimization problem 29
ordering constraints 365
orientation
  unknown 174
ov 25
overlap 25, 173
  approximate 92f.
    computation 93
  computation 29, 64
  empty 25
  generalized 41
overlap graph 179
overlap-layout-consensus scheme 172

P 31
PAM distance 95
PAM matrix 95

PAM unit 95
parameterized complexity 373
parent 27
parsimony principle 275ff.
parsimony problem 276
parsimony score 277
partial digest 131
partial digest approach 131ff.
  ideal data 131
partial digest problem *see* PDP
  atomic distances 134
  backtracking algorithm 136
  level 134
path 26
  directed 27
  in a Hidden Markov Model 228
  simple 26
pathlabel 54
pattern 37
PDP 133
perfect phylogeny haplotyping problem
    see PPH problem
PerfPhyl problem 270
permutation 239
  directed 239, 247
  extended representation 241
  signed 239
  undirected 239
phenotypical characters 268
phylogenetic tree 257ff.
  perfect 269, 289
    construction 272
    existence 275
  undirected perfect 290
  unrooted 276
phylogeny *see* phylogenetic tree
physical map 120, 123
physical mapping 123ff.
Plain 341
point set 133
polynomial-time approximation scheme
    see PTAS
polynomial-time reduction 32
polypeptide chain 9
Pos 54
position set 126
PPH problem 290
PQ-tree 147
  consistent permutation 149
  empty 149
  empty vertex 151
  equivalent 149
  front 148
  full vertex 151
  legal operation 148
  partial vertex 151
  pattern 150
  pertinent subtree 151
  pertinent vertex 151
  reduction 149
  substitute 150
  universal 149
Pref 25
pref 25
prefix 24
preprocessing 39
primary structure
  of proteins 350
  of RNA 320
primer 16
probe 20, 144
production 329
protein 7ff.
  core model 364
  de novo structure prediction 352ff.
  folding 349
  HP model 352ff.
  primary structure 9, 350
  quaternary structure 351
  secondary structure 350
  structure model 364
  structure prediction 349ff.
  tertiary structure 350
protein threading 363ff.
  branch and bound 370
protein threading problem 366
pseudoknot 323
PTAS 217

quartet 281
  consistent 281
  optimal 281
quartet method 281ff.
quartet puzzling 282f.
quaternary structure
  of proteins 351

read 18

reciprocal translocation 238
reconstruction model 197
reconstruction problem 197
relation computing problem 29
repeat 66, 174
  approximate 66
  exact 66
  maximal exact 66
    compact representation 68
repeat search problem 66
resolving pair 288
restriction enzymes 15, 123
restriction site 15, 123
restriction site mapping 123ff.
reversal 238
  for directed permutation 248
  for undirected permutation 239
reverse complement 11
ribonucleic acid *see* RNA
RNA 10
  primary structure 320
  secondary structure 320
  substructure 322
  tertiary structure 321
RNA substructure
  size of 323
root 27

SbH *see* sequencing, by hybridization
SbH reconstruction problem 202
scoring function 83
  good 112
scoring matrix 94f.
SCS 176
secondary structure
  of proteins 350
  of RNA 320ff.
self-avoiding walk 353
sequence 23
sequencing *see* DNA sequencing
  by hybridization 171, 201ff.
  shotgun 171ff.
sequencing errors 18
set partition problem 128
sheet 350
shortest common superstring problem 176
  decision version 182
  with compression measure 178
  with length measure 178
shortest covering string problem 165
shotgun method 120
shotgun sequencing *see* sequencing, shotgun
signals 213ff.
similarity 83
  computation 87
similarity matrix 84
simulated annealing 281
single nucleotide polymorphism *see* SNP
skew algorithm 72f.
SNP 288ff.
SNP conflict 311
SNP matrix 306
  gapless 311
  reduced 311
  weight 313
SNP site 288
sorting
  counting sort 71
  radix sort 72
SP-score 103
spacing constraints 365
spanning tree algorithm 33
spectrum 201
spectrum graph 202
stack 135
stacked pair 322
star alignment 110f.
state 40
  accepting 40
  initial 40
stem 322
string 23ff.
  arc-annotated 337f.
  compatible 201
  empty 23
  length 23
  reverse complementary 11
  simply compatible 201
string depth 54
string homomorphism *see* homomorphism
string matching 37ff.
  approximate 91
  Boyer–Moore algorithm 44ff.
  naive approach 38

with automata 39ff.
with suffix trees 57
string matching automaton 40f.
construction 42
string matching problem 38
naive approach 38
strip 242
ascending 242
descending 242
STS probes 146
subgraph 26
induced 26
subsequence 24
arc-preserving common 339
consistent mapping 339
longest arc-preserving common 340
longest common 338
substitution 19, 83
substring 24
frequent 223
infrequent 223
longest common 61
substring edit distance 197
substring free 177
substring problem 58
Suff 25
suff 25
suffix 24
suffix array 68ff.
inverse 73
suffix similarity 46
suffix tree 50ff., 218
compact 53
construction 55
generalized 59
simple 50
construction 51
sum of pairs *see* SP-score
supersequence 105
shortest common 105
superstring
induced 181
shortest common 176
trivial 177
symbol 23
syntenic distance 249f.
syntenic operations 250
synteny graph 253
synteny problem 251

T-contig 199
tandem repeat 217
approximate 217
computation 222
tandem repeat problem 219
tandem repeats 217ff.
taxa 257
taxon *see* taxa
terminal 329
tertiary structure
of proteins 350
of RNA 321
text 37
threading 365
three point condition 259
threshold problem 32
thymine 11
Time 30
transition function 40
transition probability 228
translocation 248ff.
transposition 238
traveling salesman problem *see* TSP
tree 27
additive 265ff.
compact additive 266
computation 267
directed 27
ordered 27
rooted 27
spanning 27
ultrametric 259
distance 259
tree compatible 271
triangle inequality 33, 112, 164, 259
trie *see* suffix tree, simple
tRNA 321
TSP 29, 164
metric 33
turnpike reconstruction problem 168

ultrametric 259
UNLIMITED 341
UPGMA algorithm 261, 263
uracil 11

variance 224
vertex 25

balanced 204
degree 25
indegree 27
inner 27
outdegree 27
semi-balanced 204
vertex labeling 28
vertical run 313
Viterbi algorithm 232

Watson–Crick complement 11
whole genome shotgun approach 120

Zuker algorithm 326f.

Natural Computing Series

L. Kallel, B. Naudts, A. Rogers (Eds.): **Theoretical Aspects of Evolutionary Computing.** X, 497 pages. 2001

G. Păun: **Membrane Computing. An Introduction.** XI, 429 pages, 37 figs., 5 tables. 2002

A.A. Freitas: **Data Mining and Knowledge Discovery with Evolutionary Algorithms.** XIV, 264 pages, 74 figs., 10 tables. 2002

H.-P. Schwefel, I. Wegener, K. Weinert (Eds.): **Advances in Computational Intelligence. Theory and Practice.** VIII, 325 pages. 2003

A. Ghosh, S. Tsutsui (Eds.): **Advances in Evolutionary Computing. Theory and Applications.** XVI, 1006 pages. 2003

L.F. Landweber, E. Winfree (Eds.): **Evolution as Computation.** DIMACS Workshop, Princeton, January 1999. XV, 332 pages. 2002

M. Hirvensalo: **Quantum Computing.** 2nd ed., XI, 214 pages. 2004 (first edition published in the series)

A.E. Eiben, J.E. Smith: **Introduction to Evolutionary Computing.** XV, 299 pages. 2003

A. Ehrenfeucht, T. Harju, I. Petre, D.M. Prescott, G. Rozenberg: **Computation in Living Cells. Gene Assembly in Ciliates.** XIV, 202 pages. 2004

L. Sekanina: **Evolvable Components. From Theory to Hardware Implementations.** XVI, 194 pages. 2004

G. Ciobanu, G. Rozenberg (Eds.): **Modelling in Molecular Biology.** X, 310 pages. 2004

R.W. Morrison: **Designing Evolutionary Algorithms for Dynamic Environments.** XII, 148 pages, 78 figs. 2004

R. Paton†, H. Bolouri, M. Holcombe, J.H. Parish, R. Tateson (Eds.): **Computation in Cells and Tissues. Perspectives and Tools of Thought.** XIV, 358 pages, 134 figs. 2004

M. Amos: **Theoretical and Experimental DNA Computation.** XIV, 170 pages, 78 figs. 2005

M. Tomassini: **Spatially Structured Evolutionary Algorithms.** XIV, 192 pages, 91 figs., 21 tables. 2005

G. Ciobanu, G. Păun, M.J. Pérez-Jiménez (Eds.): **Applications of Membrane Computing.** X, 441 pages, 99 figs., 24 tables. 2006

K.V. Price, R.M. Storn, J.A. Lampinen: **Differential Evolution.** XX, 538 pages, 292 figs., 48 tables and CD-ROM. 2006

J. Chen, N. Jonoska, G. Rozenberg: **Nanotechnology: Science and Computation.** XII, 385 pages, 126 figs., 10 tables. 2006

A. Brabazon, M. O'Neill: **Biologically Inspired Algorithms for Financial Modelling.** XVI, 275 pages, 92 figs., 39 tables. 2006

T. Bartz-Beielstein: **Experimental Research in Evolutionary Computation.** XIV, 214 pages, 66 figs., 36 tables. 2006

S. Bandyopadhyay, S.K. Pal: **Classification and Learning Using Genetic Algorithms.** XVI, 314 pages, 87 figs., 43 tables. 2007

H.-J. Böckenhauer, D. Bongartz: **Algorithmic Aspects of Bioinformatics.** X, 396 pages, 118 figs., 9 tables. 2007