Enno Ohlebusch

Bioinformatics Algorithms

Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction

Enno Ohlebusch
Institute of Theoretical
Computer Science
Faculty of Engineering
and Computer Science
University of Ulm
89069 Ulm
Germany

ISBN 978-3-00-041316-2

Cover design: Volker Haese
Printed in Germany

Contents

Preface

The origins of this book go back to the 1990s, when members of the "Technische Fakultät" (joint Department of Computer Science and Biotechnology) at the University of Bielefeld, Germany, developed curricula for the diploma (equivalent to M.Sc.) in "Naturwissenschaftliche Informatik" (Computer Science in the Natural Sciences). Computer science students could choose either Biology, Chemistry or Physics as their second subject. In Bielefeld, my former supervisor Robert Giegerich was one of the driving forces behind the development of bioinformatics, the interdisciplinary field that combines molecular biology with computer science. At that time, I was a postdoc in his research group working on term rewriting systems. I inevitably became acquainted with bioinformatics, and it proved to be a stroke of luck. Robert prepared lecture notes for a course "Algorithms on Sequences," and his work was later extended by my former colleague Stefan Kurtz. Parts of Chapter 2 (exact string matching) and Section 8.1 (pairwise alignment) can be traced back to Stefan's manuscript [194]. It also contained a chapter on suffix trees and their bioinformatics applications. However, the linear-time construction of suffix trees is difficult to understand and teach. In 2003, it was independently shown by several authors that a direct linear-time construction of suffix arrays is possible. One of the algorithms used a rather simple and clever divide and conquer strategy, and this opened up new possibilities. Not only does this simplify teaching—because suffix trees can easily be built in linear time from suffix arrays—but, more importantly, suffix trees can be *completely replaced* by suffix arrays. Algorithms on suffix arrays are not only more space efficient than their counterparts on suffix trees, but they are also faster and easier to implement. I know from discussions with colleagues that a textbook on suffix arrays and their applications would be appreciated, so I wrote one.

One of the problems I encountered in writing this book was the vast literature on the subject. This is particularly true of phylogenetic reconstruction methods. Felsenstein [97] estimates that there are about 3,000 papers on methods for inferring phylogenies, and I have only read a small

fraction of those. Therefore, I apologize in advance to my colleagues if their important work has not been cited.

What the book is about

The primary reason for writing this book was to provide an overview of the state-of-the-art in string algorithms based on index structures. Dan Gusfield published his highly recommended textbook [139] over 15 years ago using the suffix tree as a central data structure. However, suffix trees cannot be used in large-scale applications. As already mentioned, it is possible to replace this index data structure by enhanced suffix arrays. There is an extensive literature, but no textbook, on suffix arrays. This book fills that gap. Additionally, it not only describes classic topics in the field of string algorithms from a new perspective, but also introduces important advanced techniques. For example, recent research focuses on compressed index structures that are based on the Burrows-Wheeler transform. Chapter 7 discusses new approaches in that field by focusing on wavelet trees and backward search, including the latest methods and algorithms. The book focuses on exact string problems (finding exact matches, exact repeats, etc.), rather than on the biologically more relevant inexact string problems (finding approximate matches, degenerate repeats, etc.). However, efficient methods that are robust under errors most often rely on methods that solve the corresponding exact string problem. In other words, one should study the latter in order to understand the former.

Apart from string algorithms, the book presents several other important topics in computer science and bioinformatics within a unified framework. It covers classic topics, such as exact string matching, alignments, and phylogenetic reconstruction as well as newer topics such as constant time range minimum queries and sorting by reversals. The reader should have a solid background in algorithms and data structures in order to fully understand the material covered here.

Emphasis is placed on concepts and methods used to resolve problems, but in contrast to many other texts, this book also puts emphasis on efficient implementations. To give an example, the vast majority of texts on phylogenetic reconstruction explain the UPGMA-algorithm and then state that it runs in quadratic time (if they are interested in the time complexity at all). The experienced reader will be able to find an $O(n^2 \log n)$ time solution, but usually not much more than that. This book provides enough detail to allow construction of an efficient implementation of the algorithm.

Given my background in theoretical computer science, all topics are thoroughly discussed, including proofs of correctness as well as worst-

case time and space complexity analyses. Many examples and figures make the material easier to access. The majority of the algorithms are presented in pseudo-code, so they should be easy to implement. There are exceptions though: To completely implement the space-efficient algorithms described in Chapters 6 and 7 is not an easy task. Fortunately, gifted programmers like my former Ph.D. student Simon Gog and others have developed libraries that can be used for this purpose.[1]

How to use this book

The chapters in this book can be read independently, except for Chapters 4–7, and one can use each of the independent chapters as course material. All of the chapters can be used in computer science courses, some for advanced undergraduate students and some for graduate students. (Most parts of the book have been used in courses taught at the University of Ulm, predominantly at the graduate level.) Chapters 8–10 are intended to serve as a text for computer science oriented courses on bioinformatics.

A course on string algorithms and their applications to bioinformatics can be taught by starting with the material in Chapter 4, and then picking topics from Chapter 5. These chapters use range minimum queries, but the reader who is not interested in the details of how to answer such queries in constant time can skip Chapter 3.

Chapter 6 demonstrates that it is possible to implement the algorithms of Chapters 4 and 5 space efficiently, but the material is quite advanced and only suitable for graduate students. It is worth noting that Chapter 6 does not depend on Chapter 5.

The most "modern" part of this book is Chapter 7. Only parts of it are dependent upon material found in Chapters 4 to 6, so most experienced students can immediately begin with Chapter 7. However, it is recommended to read Chapter 4 first. Thus, a course on string algorithms that focuses on large-scale bioinformatics applications can be taught in this way.

Acknowledgments

I am indebted to Stefan Kurtz for sharing his lecture notes [194] with me. I am also much obliged to Johannes Fischer who used the material of Section 5.3.6 in a lecture held at the University of Tübingen in 2008 and gave me feedback on it.

I would like to thank my present and former students Mohamed Abouelhoda, Michael Arnold, Martin Bader, Timo Beller, Katharina Berger, Axel

[1] Simon's library can be found at `https://github.com/simongog/sdsl`.

Fürstberger, Simon Gog, Adrian Kügel, Christoph Mehre, Julian Rüth, Thomas Schnattinger, Florian Schüle, Benjamin Weggenmann, and Maike Zwerger for their assistance in preparing figures, providing implementations, and other important work. It has been a pleasure to work with so many excellent students; many of them are co-authors on scientific publications.

Thanks to my brother-in-law Volker Haese for the cover-design and to my friends Roland Ehle, Michael Schmeisser, and Silvio Weißenborn in helping to bring this book before the public.

Chapter 1

Molecular Biology in a Nutshell

1.1 Nucleic acids and proteins

The three major macromolecules that are essential for all known forms of life are deoxyribonucleic acid (DNA), ribonucleic acid (RNA), and proteins.

DNA

Deoxyribonucleic acid is the carrier of genetic information of all known living organisms and many viruses. Most DNA molecules are double-stranded helices, consisting of two long polymers of simple units called nucleotides; see Figure 1.1. A nucleotide is composed of a phosphate group, a five-carbon sugar (2-deoxyribose), and a nucleobase attached to the sugar. There are four nucleotides in DNA that can be distinguished by their bases: adenine (A), cytosine (C), guanine (G), and thymine (T). A single strand of the DNA molecule has a backbone made of alternating sugars and phosphate groups: a phosphate group is linked to the $5'$-carbon of the sugar of its nucleotide and the $3'$-carbon of the sugar of another nucleotide (the numbers $1'$ to $5'$ refer to the carbon atom locations in the sugar structure). Figure 1.2 shows a schematic view of a single-stranded DNA.

Figure 1.1: Schematic view of the DNA double helix.

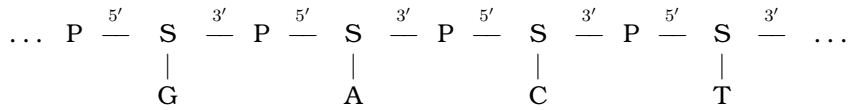

Figure 1.2: A schematic view of a DNA strand. P stands for a phosphate group and S for a sugar 2-deoxyribose.

A single strand of a DNA molecule—a chain of nucleotides—has a direction, conventionally noted as $5'$ to $3'$. If we omit the sugar-phosphate backbone in Figure 1.2, we obtain

$$5' \quad \ldots \quad G \quad A \quad C \quad T \quad \ldots \quad 3'$$

We often omit the $5'$ and $3'$ markers because the beginning of a DNA sequence is the $5'$ end unless otherwise stated.

A double-stranded DNA molecule consists of two complementary DNA sequences held together by base pairs. For example, the DNA sequence GACT and its complementary strand can be schematically viewed as

$$\begin{array}{cccccc} 5' & G & A & C & T & 3' \\ 3' & C & T & G & A & 5' \end{array}$$

The Watson-Crick base pairs G-C and A-T (guanine-cytosine and adenine-thymine) are formed by hydrogen bonds. G and C are called complementary bases, and so are A and T. The reverse complement of a DNA sequence is obtained by writing its complement with the $5'$ end on the left and the $3'$ end on the right. For example, the reverse complement of GACT is AGTC.

The complementary nature of the based-paired structure of a double-stranded DNA molecule provides the basis for replication. DNA replication begins at specific locations called origins. The unwinding of double-stranded DNA into two single strands and the simultaneous synthesis of new strands forms a replication fork; see Figure 1.3. Each of the two single strands serves as a template for the production of its complementary strand: an enzyme called DNA polymerase synthesizes the new DNA by adding nucleotides matched to the template strand. Synthesis always occurs in the $5'$ to $3'$ direction. So one strand, the leading strand, can be synthesized continuously while the other, the lagging strand, must be synthesized discontinuously in short fragments, which are later joined. To sum up, after replication there are two identical copies of the original double-stranded DNA molecule. Although replication errors may occur, cellular proofreading ensures that the error rate is kept very low.

DNA molecules may be circular or linear, and they can have an enormous length (the length is measured by the number of base pairs). A DNA

Chapter 2

Exact String Matching

Finding all occurrences of a pattern in a text is a problem that arises in many different contexts like text editing, information retrieval, and biological sequence analysis. For example, one of the typical functions of a text editor is to search for the occurrences of a particular string (the pattern) in a document (the text) and to replace them with another string, where both strings are supplied by the user. Another typical application is to search the World Wide Web for information using a web search engine. The user enters a query in form of one or several strings (the patterns) and the search engine returns a list of documents and web pages in which the patterns occur (so in this case, the text is in fact a collection of web pages). In bioinformatics, exact string matching is used to search for particular patterns in DNA sequences.

In this chapter, we present some well-known exact string matching algorithms that do *not* preprocess the text. In large-scale applications like searching for all occurrences of a particular pattern in a genome sequence, however, one usually computes an index of the text to accelerate the search. We will address this issue later.

2.1 Basic string definitions

Definition 2.1.1 An *alphabet* Σ is a finite set, whose elements are called *characters* (or letters). The size of Σ is denoted by $\sigma = |\Sigma|$.

For example, the basic modern Latin alphabet of lowercase letters is

$$\Sigma = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

and $\sigma = 26$. In most parts of this book, we assume a total order on the alphabet Σ. For example, the standard order of the Latin alphabet is:

$$a < b < c < \cdots < x < y < z$$

In bioinformatics, the DNA alphabet

$$\Sigma = \{\text{A,C,G,T}\}$$

plays a central role.

Definition 2.1.2 A *string* S is a finite sequence of characters (where the characters are drawn from some alphabet Σ). The length of a string S, denoted by $|S|$, is the number of characters composing the string. The empty string, denoted by ε, has length 0. Given a string S, its *reverse string* is obtained by reversing the order of the characters in S.

For example, TATAAACG is a string of length 8 on the DNA alphabet. If the alphabet is not specified, then we tacitly assume that it consists of all characters appearing in S.

The concatenation of two strings is the string formed by writing the first, followed by the second, with no intervening space. For example, the concatenation of TATA and AACG is TATAAACG. The juxtaposition is used as the concatenation operator. That is, if u and v are strings, then uv is the concatenation of these two strings. The empty string is the identity for the concatenation operator, i.e., $\varepsilon\omega = \omega = \omega\varepsilon$ for each string ω.

In the biological literature, the word "sequence" is often used instead of "string." For example, a string on the four-letter alphabet {A,C,G,T} is called a DNA sequence, and a string on the twenty-letter amino acid alphabet is called an amino acid sequence (or protein sequence if it represents the primary structure of a protein). Throughout this book, we use the word "sequence" instead of "string" solely in those cases in which the alphabet is either the DNA alphabet or the amino acid alphabet.

For a given alphabet Σ, Σ^n denotes the set of all strings on Σ having length n. Formally, $\Sigma^0 = \{\varepsilon\}$ and $\Sigma^n = \{a\omega \mid a \in \Sigma, \omega \in \Sigma^{n-1}\}$ for $n > 0$. The set of all strings on the alphabet Σ is

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$$

where $\mathbb{N}$ is the set of natural numbers (including zero). The set of all non-empty strings on the alphabet Σ is $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$.

Definition 2.1.3 Given a string S of length n, $S[i..j]$ denotes the *substring*[1] of S that begins at position i and ends at position j (by definition, $S[i..j]$ is

[1]The words "string" and "sequence" can be used synonymously but "substring" and "subsequence" can not. In mathematics, a subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements. By contrast, a substring is always a consecutive part of a string. This means that a substring of a string is a contiguous subsequence of the string, but a subsequence of a string is not necessarily a substring of the string.

Chapter 3

Answering Range Minimum Queries in Constant Time

This chapter deals with the problem of answering minimum range queries and lowest common ancestor queries in constant time, under the constraint that only linear time is spent in a preprocessing phase. In subsequent chapters we will make extensive use of this, but readers who are not interested in the algorithms that solve this problem may safely skip this chapter. However, they should acquaint themselves with the basic definitions.

3.1 Basic definitions

Definition 3.1.1 Given an array $A[1..n]$ of integers[1] and two indices i and j with $1 \leq i \leq j \leq n$, a *range minimum query* on the interval $[i..j]$ returns an index k so that $A[k] = \min\{A[l] \mid i \leq l \leq j\}$. Such a query will henceforth be denoted by $\mathsf{RMQ}_A(i, j)$.

We stress that our array indexing starts at 1.

By definition, an interval $[i..j]$ with $i > j$ is empty and $\mathsf{RMQ}_A(i, j)$ is undefined in this case. Moreover, if the minimum element in $A[i..j]$ occurs more than once in $A[i..j]$, then the index of the minimum element is not unique. In this case, we assume that $\mathsf{RMQ}_A(i, j)$ returns the first index of the minimum element in $A[i..j]$, unless stated otherwise.

As an example, consider the array $A = [8, 4, 6, 2, 12, 10, 14, 6]$. The range minimum query $\mathsf{RMQ}_A(5, 8)$, returns the index 8 because $A[8] = 6$ is the minimum element in $A[5..8] = [12, 10, 14, 6]$.

[1] Here, the array elements are integers, i.e., elements of the totally ordered set $\mathbb{Z}$. However, one can use any other totally ordered set instead of the integers.

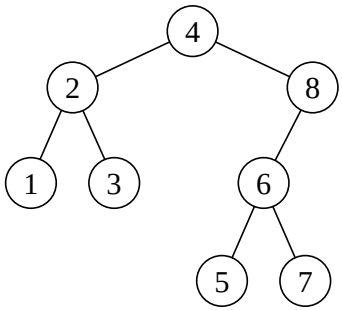

Figure 3.1: In this tree, $\mathsf{LCA}_T(5,7)$ returns the node 6, while $\mathsf{LCA}_T(1,7)$ returns the root node 4.

Definition 3.1.2 Given a rooted tree T and two nodes u and v in T, a *lowest common ancestor query* on u and v, denoted by $\mathsf{LCA}_T(u,v)$, returns the node farthest from the root that is an ancestor of both u and v.

Recall that in a rooted tree the ancestors of a node u are the nodes on the direct path from u to the root of the tree. Therefore, the lowest common ancestor of u and v is the node at which the path from u to the root meets the path from v to the root. As an example, consider the tree T in Figure 3.1. The lowest common ancestor query $\mathsf{LCA}_T(5,7)$ returns the node 6, while $\mathsf{LCA}_T(1,7)$ returns the root node 4.

Early papers on finding lowest common ancestors in trees are [9] and [306]. Harel and Tarjan [146] showed that a fixed tree can be preprocessed in linear time and space so that LCA queries can be answered in constant time. Their algorithm was simplified by Schieber and Vishkin [283], but it cannot be called "simple" algorithm. Gabow et al. [117] showed that the problems of answering range minimum queries and lowest common ancestor queries are linearly equivalent. Berkman and Vishkin [39] observed that the reduction from LCA to RMQ via the Euler-Tour (see Section 3.2) is in fact a reduction to a special RMQ-problem, in which consecutive array elements differ by one. They solved this restricted RMQ-problem and obtained a new algorithm for finding lowest common ancestors in constant time. Bender and Farach-Colton [35] later provided a simplified presentation of their algorithm.

3.2 Range minimum vs. lowest common ancestor

In this section, it will be shown that range minimum queries can be answered in constant time (with linear time and space preprocessing) if and

Chapter 4

Enhanced Suffix Arrays

The meteoric increase of DNA sequences produced by next-generation sequencers demands new approaches in computer science for storing, analyzing, and mining the accumulating data. While the databases are growing rapidly, the data representing DNA sequences of the human chromosomes do not change much over time. As a consequence, index data structures such as suffix arrays or suffix trees can be used to efficiently solve a myriad of sequence analysis problems.

In this chapter, we will introduce the suffix array and the LCP-array of a string, and present linear-time algorithms to construct them. The combination of these (and possibly other) arrays is called an *enhanced suffix array*. Furthermore, we define the lcp-interval tree and provide tree traversal algorithms that solely rely on the enhanced suffix array. At the end of the chapter, we shall see that the lcp-interval tree of a string S coincides with the suffix tree of S. The main advantage of using an lcp-interval tree is that it can be traversed without building the tree itself.

Memory is not an issue here. As we shall see in Chapter 6, the memory usage of enhanced suffix arrays can be reduced significantly.

4.1 Suffix arrays

To construct the suffix array of a string S boils down to sorting all suffixes of S in lexicographic order (also known as alphabetical order, dictionary order, or lexical order). This order is induced by an order on the alphabet Σ. In this book, Σ is an ordered alphabet of constant size σ. It is sometimes convenient to regard Σ as an array of size σ so that the characters appear in ascending order in the array $\Sigma[1..\sigma]$, i.e., $\Sigma[1] < \Sigma[2] < \cdots < \Sigma[\sigma]$. Conversely, each character in Σ is mapped to a number in $\{1, \ldots, \sigma\}$. The smallest character is mapped to 1, the second smallest character is mapped to 2, and so on. In this way, we can use a character as an index for an array.

Definition 4.1.1 Let $<$ be a total order on the alphabet Σ. This order induces the *lexicographic order* on Σ^* (which we again denote by $<$) as follows: For $s, t \in \Sigma^*$, define $s < t$ if and only if either s is a proper prefix of t or there are strings $u, v, w \in \Sigma^*$ and characters $a, b \in \Sigma$ with $a < b$ so that $s = uav$ and $t = ubw$.

To determine the lexicographic order of two strings, their first characters are compared. If they differ, then the string whose first character comes earlier in the alphabet is the one which comes first in lexicographic order. If the first characters are the same, then the second characters are compared, and so on. If a position is reached where one string has no more characters to compare while the other does, then the shorter string comes first in lexicographic order.

In algorithms that need to determine the lexicographic order of two suffixes of the same string S, a cumbersome distinction between "has more characters" and "has no more characters" can be avoided by appending the special symbol \$ (called *sentinel character*) to S. In the following, we assume that \$ is smaller than all other elements of the alphabet Σ. If \$ occurs nowhere else in S and the lexicographic order of two suffixes of S is determined as described above, then it cannot happen that one suffix has no more characters to compare. As we shall see later, there are other situations in which it is convenient to append the special symbol \$ to a string.

Definition 4.1.2 Let S be a string of length n. For every i, $1 \leq i \leq n$, S_i denotes the i-th suffix $S[i..n]$ of S. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic order of the n suffixes of the string S. That is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$.

The *inverse suffix array* ISA is an array of size n so that for any k with $1 \leq k \leq n$ the equality $\mathsf{ISA}[\mathsf{SA}[k]] = k$ holds.

The inverse suffix array is sometimes also called $rank$-array because $\mathsf{ISA}[i]$ specifies the rank of the i-th suffix among the lexicographically ordered suffixes. More precisely, if $j = \mathsf{ISA}[i]$, then suffix S_i is the j-th lexicographically smallest suffix. Obviously, the inverse suffix array can be computed in linear time from the suffix array. Figure 4.1 shows the suffix array and the inverse suffix array of the string $S = ctaataatg$.

Suffixes of S that share a common prefix occur consecutively in the suffix array SA because SA contains the suffixes of S in lexicographically sorted order. In other words, they form an interval in the suffix array.

Definition 4.1.3 Let SA be the suffix array of the string S. For every substring ω of S, the ω-interval in SA is the interval $[i..j]$ so that

- ω is not a prefix of $S_{\mathsf{SA}[i-1]}$,

Chapter 5

Applications of Enhanced Suffix Arrays

In this chapter, we discuss several applications of enhanced suffix arrays. The prime motivation for developing suffix trees and suffix arrays was fast exact string matching in situations in which the string S is kept fixed. An application of special importance in bioinformatics is sequence analysis, where S can be the DNA sequence of a chromosome. By concatenating the DNA sequences of all chromosomes (inserting a separator between each string), it is also possible to analyze a complete genome. This chapter is organized as follows. Section 5.1 discusses exact string searching algorithms. We shall see that the suffix array is an index data structure that significantly improves the speed of the search. Section 5.2 describes fast algorithms that compute the Lempel-Ziv factorization of a string. This factorization plays an important role in data compression and is also the basis for the detection of all tandem repeats. Section 5.3 presents several algorithms for finding various kinds of repeats. Repeat analysis is important because repetitive sequences are abundant in eukaryotic genomes (especially in mammalian and plant genomes). Then, we address the problem of comparing two or multiple strings. Sections 5.4 and 5.5 focus on the comparison of two strings. If one wishes to compare two eukaryotic genomes, then one usually starts with the computation of exact matches satisfying various criteria. A related problem is computing the matching statistics between two strings. Suffix-links in the (virtual) lcp-interval tree will prove to be the key in efficiently solving these and related problems. Section 5.6 deals with problems that involve multiple strings. Although it touches the field of document retrieval, the main emphasis is on the analysis of large collections of DNA sequences. Section 5.7 discusses string kernels. A kernel can be thought of as a similarity measure that is used to compare the data, and kernel methods are used e.g. for classification. Section 5.8 shows how to extract frequently occurring patterns from a set of string databases.

5.1 Exact string matching

In the following, P denotes a string of length m, called a *pattern*. It is assumed that m is small compared to the length n of the string S. We recall from Chapter 2 that the exact string matching problem (find all positions in S at which an occurrence of P begins) can be solved in $O(n+m)$ time.

In several applications, the string S is *static* (i.e., does not change) and many patterns $P^1, \ldots, P^k$ have to be matched against S. If all patterns are known in advance, then the Aho-Corasick algorithm (described in Section 2.5) finds all occurrences of $P^1, \ldots, P^k$ in S in $O(n+\sum_{i=1}^{k}|P^i|+z)$ time, where z denotes the overall number of occurrences. However, if yet another pattern P is input (which was not known in advance) then $O(n+m)$ time is further needed to find all positions in S at which an occurrence of P begins. In this situation, if the string S is static and the exact string matching problem has to be solved for an unknown number of patterns (which are successively input), it pays to build the enhanced suffix array of S in $O(n)$ time and to match each pattern against this index. We show how the enhanced suffix array of S allows us to answer

- *decision queries* of the type "Is P a substring of S?" in optimal $O(m)$ time (for a constant-size alphabet),
- *counting queries* of the type "How often does P occur in S?" in optimal $O(m)$ time (for a constant-size alphabet),
- *enumeration queries* of the type "Where are all z occurrences of P in S?" in optimal $O(m+z)$ time (for a constant-size alphabet),

totally independent of the size of S.

5.1.1 Forward search on suffix trees

Since the suffix tree ST for $S\$$ contains all substrings of $S\$$, it is easy to verify whether some pattern P (of course, we assume that $\$$ does not occur in P) is a substring of S: just follow the path from the root directed by the characters of P. If at some point one cannot proceed with the next character in P, then P does not occur in the suffix tree and hence it is not a substring of S. Otherwise, if P occurs in the suffix tree, then it is a substring of S. This string matching algorithm answers a decision query in $O(m)$ time, provided that for each node α in ST and each $a \in \Sigma$ one can determine the a-edge outgoing from α (if it exists) in constant time. Constant time access is possible if the outgoing edges are stored in an array of size σ, i.e., the array has an entry for each character of the alphabet Σ. This, however, increases the space consumption of the suffix tree to $O(n\sigma)$, which is not tolerable in larger applications. Thus,

Chapter 6

Making the Components of Enhanced Suffix Arrays Smaller

Up to this point, we did not trouble ourselves much about memory requirements. However, in some applications of enhanced suffix arrays we needed more than just one or two arrays. In large scale applications, the total memory usage of these arrays may be prohibitive. Fortunately, the various components of enhanced suffix arrays can be made smaller. We shall learn in this chapter how this can be done for the suffix array and the LCP-array. Moreover, we shall see that essentially one small data structure suffices to support the following tasks:

- find the parent interval of an lcp-interval,
- find all child intervals of an lcp-interval,
- find the suffix link interval of an lcp-interval,
- answer a range minimum query.

These results rely on the fact that $rank$ and $select$ queries can be answered in constant time. From now on, we state the alphabet size explicitly in complexity analyses.

6.1 Constant time $rank$ and $select$ queries

Virtually all methods to compress the components of enhanced suffix arrays take advantage of the following theorem.

Theorem 6.1.1 *A bit vector $B[1..n]$ can be preprocessed in linear time so that the following operations are supported in constant time:*

- $rank_b(B, i)$*: returns the number of occurrences of bit b in $B[1..i]$.*

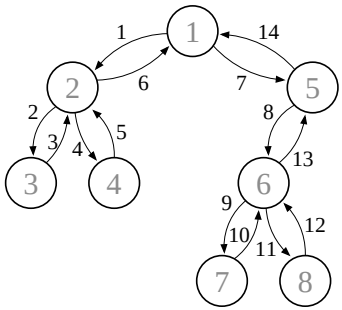

Figure 6.1: A preorder traversal of the tree follows the consecutively numbered arrows. Each node is labeled with its preorder index i, i.e., node i is the i-th node visited in the preorder traversal.

- $select_b(B, i)$*: returns the position of the* i*-th occurrences of bit* b *in* $B[1..n]$*.*

The bit vector $B[1..n]$ *uses* n *bits and the supporting data structures use only* $o(n)$ *bits.*

For the bit vector $B = 1110100111010000$, we have for example $rank_1(B, 6) = 4$ and $select_0(B, 4) = 11$. If it is clear from the context which bit vector B is meant, we write $rank_b(i)$ and $select_b(i)$ instead of $rank_b(B, i)$ and $select_b(B, i)$.

It is beyond the scope of this book to provide a proof of the theorem. The reader is referred to the seminal work of Jacobson [162, 163] who showed that attaching a dictionary of size $o(n)$ bits to the bit vector $B[1..n]$ is enough to answer $rank$ queries in constant time. His solution for the $select$ operation was not optimal, but later Clark [58] proved that $o(n)$ bits are enough to answer $select$ queries in constant time; see also [228].

Jacobson [163] was interested in the $rank$ and $select$ operations because they allowed him to simulate tree traversals on bit vectors. There are several possibilities to represent a (static) tree by a bit vector. Here we will review just one of them [230]: the tree T is represented by a balanced sequence of parentheses that is obtained by a preorder (depth-first) traversal of T; cf. Figure 6.1. To be precise, start with the root of T and do the following:

1. Write an opening parenthesis.
2. For each child node (from left to right), write its balanced parentheses sequence.
3. Write a closing parenthesis.

Chapter 7

Compressed Full-Text Indexes

Until now, we have used the suffix array as an index data structure, and exact string matching was done in forward direction. This chapter is dedicated to index data structures and applications in which a forward search is replaced with a backward search. It is organized as follows. First, we review the Burrows-Wheeler transform [48]—a well-known technique employed in lossless data compression—on which backward search is based. Second, we describe the search algorithm discovered by Ferragina and Manzini [100]. Third, we introduce the wavelet tree invented by Grossi et al. [134]. This data structure supports backward search and has many other virtues. In subsequent sections, we shed new light on solutions to problems faced in Chapter 5, such as developing new algorithms that address space efficiency issues, as well as problems related to bidirectional searches and approximate string matching.

7.1 The components of a compressed full-text index

Many variations of *compressed suffix trees* (CSTs) have been proposed in the literature (see e.g. [273]), and these do not all have the same functionality. Because we focus on backward search, which is normally not supported by a CST, we prefer the term "compressed full-text index".

Definition 7.1.1 A *compressed full-text index* of a string S is a space-efficient data structure that supports (at least) the following operations:

1. backward search,
2. access to the suffix array of S,
3. access to the LCP-array of S,
4. navigation on the (virtual) suffix tree of S.

The compressed full-text index of the string S that is used throughout this book consists of the following four components:

1. the wavelet tree of the Burrows-Wheeler transformed string of S,
2. the sparse suffix array of S from Section 6.2.1,
3. the compressed LCP-array as explained in Section 6.2.2,
4. the balanced parentheses sequence BPS of the LCP-array that was introduced in Section 6.3.

We emphasize that each of the four components can be replaced with another component that has the same functionality. For example, the wavelet tree can be substituted by the compressed suffix array [135, 270] sketched in Section 6.2.1 because backward search can be done with the ψ-function; see Exercise 7.3.3. Further alternatives are described in [238]. However, the wavelet tree has many sophisticated properties that make it most suitable for many applications. Alternative compressed representations of the LCP-array are discussed in [126], among which is a representation that is based on the array LCP′ from Section 6.3.6. There are also alternatives to the BPS of the LCP-array, most notably the BPS_{pre} introduced in Section 6.1; cf. [231, 273]. We refer to [124] for an in-depth experimental study of the various incarnations of compressed full-text indexes.

7.2 The Burrows-Wheeler transform

The Burrows-Wheeler transform was introduced in a technical report written by David Wheeler and Michael Burrows [48]; see the historical notes in Adjeroh et al. [6]. In practice, the Burrows-Wheeler transformed string tends to be easier to compress than the original string; see e.g. [48, Section 3] and [215] for reasons why the transformed string compresses well.

Here we assume that the string S of length n is terminated by the sentinel character \$. Although this is not necessary for the Burrows-Wheeler transform to work correctly (cf. [48]), in virtually all practical cases the file to be compressed is terminated by a special symbol, the EOF (end of file) character. Moreover, it allows us to use a fast suffix sorting algorithm to compute the transformed string.

7.2.1 Encoding

The Burrows-Wheeler transform transforms a string S in three steps:

1. Form a conceptual matrix M' whose rows are the cyclic shifts of the string S.

Chapter 8

Sequence Alignment

In this chapter, we discuss methods to measure how similar biological sequences are. In modern molecular biology, this is important because of the, as Dan Gusfield [139] calls it, "first fact of biological sequence analysis:"

> In biomolecular sequences (DNA, RNA, or amino acid sequences), high sequence similarity usually implies significant functional or structural similarity.

To understand why this is so, we need to know the basic principles of evolution. Evolution is broadly described as the theory that all life on earth is descended from a single common ancestor. Genes in two species are *homologous* if the same gene was present in their last common ancestor. The term *homolog* may apply to the relationship between genes separated (a) by the event of speciation or (b) by the event of duplication. In case (a), the genes are *orthologs*. Normally, orthologs retain the same function in the course of evolution. The identification of orthologs is critical for a reliable prediction of gene function in newly sequenced genomes. In case (b), the genes are *paralogs*. Paralogs evolve new functions, usually related to the original one. This is possible because after duplication there are two copies of the same gene, and mutations altering the product of one copy are not harmful to the organism as long as the other copy functions properly (this paradigm of protein evolution is often called "duplication with modification").

Sequence alignment allows us to measure sequence similarity. David Mount [227] introduces it as follows:

> Sequence alignment is the procedure of comparing two (pairwise alignment) or more (multiple alignment) sequences by searching for a series of individual characters or character patterns that are in the same order in the sequences. Two sequences are aligned by writing them across a page in rows.

> Identical or similar characters are placed in the same column, and nonidentical characters can either be placed in the same column as a mismatch or opposite a gap in the other sequence. In an optimal alignment, nonidentical characters and gaps are placed to bring as many identical or similar characters as possible into vertical register. Sequences that can readily be aligned in this manner are said to be similar.

Sequence alignment is central in bioinformatics. As mentioned above, one distinguishes between pairwise and multiple alignment. The former will be discussed in Section 8.1 and the latter is dealt with in Section 8.2. Moreover. there are two types of sequence alignment: global and local. Global alignments form the basis of phylogenetic inference (see Chapter 10) and comparative genomics, whereas similarity detected in a local alignment is generally interpreted as structural/functional closeness. In this book, we will focus on global alignment methods.

8.1 Pairwise alignment

In the following, let S^1 and S^2 be strings on the alphabet Σ of length n_1 and n_2, respectively. The formal definition of an alignment between S^1 and S^2 reads as follows.

Definition 8.1.1 A *global alignment* between S^1 and S^2 is a $(2 \times n)$ matrix A so that:

1. $A(i,j) \in \Sigma \cup \{-\}$, where $-$ is a special gap symbol not occurring in Σ.
2. After removal of all gap symbols the first row of A equals S^1 and the second row of A equals S^2.
3. No column of A consists solely of gap symbols.

Note that condition (2) implies that $n \geq \max\{n_1, n_2\}$, while condition (3) has $n \leq n_1 + n_2$ as a consequence. In fact, for $n_1 \geq n_2$ a shortest possible alignment between S^1 and S^2 is the alignment

$$\begin{pmatrix} S^1[1]\, S^1[2] \ldots S^1[n_2]\, S^1[n_2+1] \ldots S^1[n_1] \\ S^2[1]\, S^2[2] \ldots S^2[n_2] \quad - \quad \ldots \quad - \end{pmatrix}$$

of length $n = n_1 = \max\{n_1, n_2\}$, while a longest possible alignment between S^1 and S^2 is the following alignment of length $n = n_1 + n_2$:

$$\begin{pmatrix} S^1[1]\, S^1[2] \ldots S^1[n_1] \quad - \quad - \quad \ldots \quad - \\ - \quad - \quad \ldots \quad - \quad S^2[1]\, S^2[2] \ldots S^2[n_2] \end{pmatrix}$$

Chapter 9

Sorting by Reversals

9.1 Introduction

During evolution, genomes are subject to both small-scale and large-scale mutations. Small-scale mutations (point mutations) consist of the substitution, insertion or deletion of single nucleotides, while large-scale mutations (genome rearrangements) alter the order and orientation (strandedness) of genes on the chromosomes. In the single chromosome case (e.g. bacterial or mitochondrial DNA), the most common rearrangements are *inversions* (where a section of the genome is excised, reversed in orientation, and reinserted) and *transpositions* (where a section of the genome is excised and reinserted at a new position in the genome; this may or may not also involve an inversion). As is usually done in bioinformatics, we will use the term "reversal" as synonym for "inversion." Further large-scale mutations are duplications, deletions (gene loss), and insertions (horizontal gene transfer). In genomes with multiple chromosomes, important genome rearrangements are reciprocal translocations (where two non-homologous chromosomes break and exchange fragments), but also fusions (where two chromosomes fuse) and fissions (where a chromosome breaks into two parts) occur. Genome rearrangements are rare compared to point mutations, and they can give us valuable information about ancient events in the evolutionary history of organisms. For this reason, one is interested in the most plausible genome rearrangement scenario between species. Work on genome rearrangements dates back to the 1930's, when Dobzhansky and Sturtevant studied the geographical and temporal variation of chromosomal arrangements in *Drosophila pseudoobscura* and its relatives; see e.g. [81]. The comparison of genomes based on their gene arrangements was pioneered by Sankoff; see e.g. [278].

As an example, we consider mitochondrial DNA (mtDNA). A eukaryotic cell contains several thousand mitochondria. Mitochondria descended from free-living bacteria that became symbiotic with eukaryotic cells. In

Gene	Gene product
COI, COII, COIII	Cytochrome c oxidase subunits I, II, and III
Cytb	Cytochrome b apoenzyme
ND1-6, 4L	NADH dehydrogenase subunits 1 to 6 and 4L
ATP6, ATP8	ATP synthase subunits 6 and 8
srRNA	small ribosomal subunit RNA (12S rRNA)
lrRNA	large ribosomal subunit RNA (16S rRNA)
L1 and L2	two leucine tRNAs
S1 and S2	two serine tRNAs
tRNAs	18 amino acid-specific transfer RNAs

Figure 9.1: Mitochondrial DNA contains 37 genes: 13 code for proteins, two for rRNA, and 22 code for tRNA (identified by the one-letter amino acid code; cf. Figure 1.6 on page 6). The following synonyms are often used: cox1, cox2, cox3 for COI, COII, COIII; cob for Cytb; nad1-6 and nad4L for ND1-6 and ND4L; rns for srRNA and rnl for lrRNA.

1	2	3	4	5	6	7	8	9	10
COI	-S2	D	COII	K	ATP8	ATP6	COIII	G	ND3
11	12	13	14	15	16	17	18	19	20
R	ND4L	ND4	H	S1	L1	ND5	-ND6	-E	Cytb
21	22	23	24	25	26	27	28	29	30
T	-P	F	srRNA	V	lrRNA	L2	ND1	I	-Q
31	32	33	34	35	36	37			
M	ND2	W	-A	-N	-C	-Y			

Figure 9.2: Mitochondrial gene arrangement of *Homo sapiens*; see [14,40]. A minus sign (-) indicates that the gene lies on the light strand of the mtDNA.

other words, the mtDNA is derived from the circular genomes of the bacteria that were engulfed by the early ancestors of today's eukaryotic cells. The circular chromosome of mtDNA is quite small: the human mtDNA, for example, contains about 16568 bp. The mitochondrial DNA molecule contains 37 genes; see Figure 9.1. In humans, one strand (called the heavy strand because it is heavier than the other strand) carries 28 genes and the other (the light strand) carries only 9 genes; see Figure 9.2.

Let us consider the gene arrangement in the circular mitochondrial chromosome of two different species: the fruit fly *Drosophila melanogaster* and the mosquito *Anopheles quadrimaculatus*; see Figures 9.3 and 9.4. The mitochondrial gene arrangements are the same except for three tRNA differences: R and A have switched positions and S1 was inverted. It is

Chapter 10

Phylogenetic Reconstruction

10.1 Introduction

There is strong evidence that all life on earth is descended from a single common ancestor. Over a period of at least 3.8 billion years that life form has split repeatedly into new and independent lineages. The evolutionary relationships among these species is referred to as their *phylogeny* and phylogenetic reconstruction is concerned with inferring the phylogeny of groups of organisms. These groups are called *taxa* (singular: taxon). Figure 10.1 shows a phylogenetic tree of the great apes. This tree has been constructed based on genetic and fossil evidence but the actual phylogeny is unknown because ancestral species have become extinct.

The splitting of lineages is called *speciation*. The most common reason for a speciation event is that one population becomes split into two sub-populations that can eventually no longer interbreed with each other. There are several ways this can happen, but the easiest one to visualize is geographical isolation. For example, the formation of the Congo River separated two chimpanzee populations because chimpanzees are not proficient swimmers, and the sea separates Sumatran from Bornean orang-utans; cf. Figure 10.1. Once that happens, each of the two populations evolve independently of each other. They undergo different random mutations and are exposed to different selection pressures. After many generations, the changes accumulate to the extent that the two populations evolve into separate species.

The tree of life is a phylogeny of species, but methods of phylogenetic reconstruction can also be applied in other biological contexts, such as inferring the phylogeny of different populations within a species. These methods can even be applied in non-biological settings: e.g. Barbrook et al. [27] described the phylogeny of different fifteenth-century manuscripts of "The Wife of Bath's Prologue" from The Canterbury Tales.

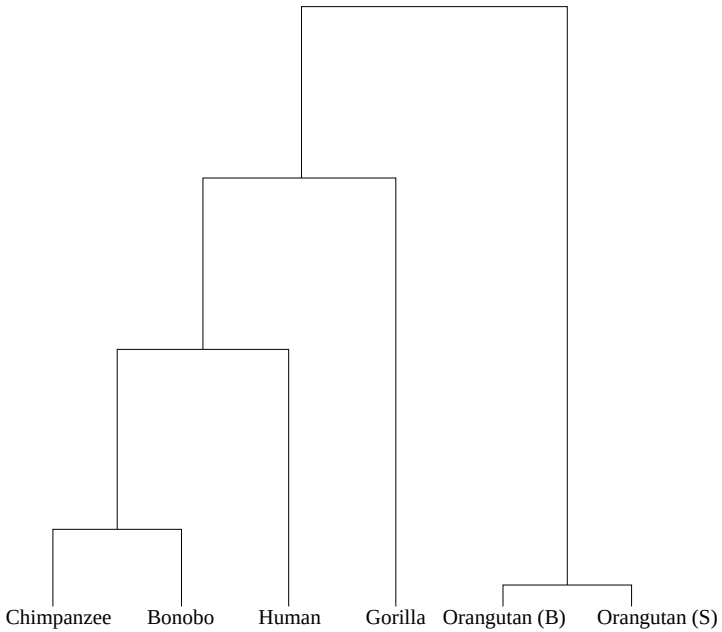

Figure 10.1: Phylogeny of the great apes (Hominidae), including six extant species: chimpanzees (*Pan troglodytes*), bonobos (*Pan paniscus*), humans (*Homo sapiens*), gorillas (*Gorilla gorilla*), Bornean orangutans (*Pongo pygmaeus*), and Sumatran orangutans (*Pongo abelii*). The length of a vertical line (branch length) represents the amount of time that lies between two speciation events. Chimpanzee and bonobo split with the formation of the Congo River, around 2 million years (myr) ago [261]. Scally et al. [282] placed the human-chimpanzee and human-chimpanzee-gorilla speciation events at approximately 6 and 10 myr ago. However, the estimates differ from study to study: according to Ingman et al. [160], the last common ancestor (LCA) of chimpanzees and human lived about 5 myr ago, while [57] estimate that it lived 5-6 myr ago. According to [207], the orangutans diverged from the Hominidae family about 12-16 myr ago. They estimate that the two orangutan species diverged about 400,000 years ago [207].