Indices and Applications in High-Throughput Sequencing

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
vorgelegt von

David Weese

am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

Berlin 2012

Datum des Disputation: ***03.06.2013***

Gutachter:
Prof. Dr. Knut Reinert, *Freie Universität Berlin, Deutschland*
Prof. Dr. Laurent Mouchard, *Université de Rouen, Frankreich*

für Nadine, Helene und Johan

Short Abstract

During the last years, sequencing throughput increased dramatically with the introduction of so-called high-throughput sequencing. It allows the production of billions of base pairs (bp) per day in the form of reads of length 100 bp and more, and current developments promise the personal $1,000 genome in a couple of years. These advances in sequencing technology demand for novel approaches and efficient data structures specifically designed for the analysis of mass data. One such data structure is the substring index, that represents all substrings or substrings up to a certain length contained in a given text.

In this thesis, we present three different substring indices and their applications in the analysis of high-throughput sequencing data. Our contribution is threefold: first, we extend the indices which were originally designed to index a single sequence to be applicable to datasets consisting of millions of multiple strings. Further, we implement algorithms for the internal memory construction of each index and devise efficient external memory algorithms for indexing large datasets, e.g. multiple mammal genomes. To make all indices easy-to-use we provide a uniform framework for accessing the generalized suffix tree and use it to exemplarily implement three iterators for searching repeats. For the exact and approximate string matching problem we provide index based filtering algorithms and algorithms recursively descending suffix trees.

Second, we present RazerS, a read mapper that aligns millions of single or paired-end reads of arbitrary lengths to their potential genomic origin using either Hamming or edit distance. Our tool can work either lossless or with a user-defined loss rate at higher speeds. Given the loss rate, we present a novel approach that guarantees not to lose more reads than specified. This enables the user to adapt to the problem at hand and provides a seamless tradeoff between sensitivity and running time. We develop two index based filters and a banded variant of Myers' bit-vector algorithm to efficiently reduce the alignment search space and use OpenMP for shared-memory parallelism. We compare RazerS with other state-of-the-art read mappers and show that it has the highest sensitivity and a comparable performance on various real-world datasets.

Third, we propose a general approach for frequency based string mining, which has many applications, e.g. in contrast data mining. Our contribution is a novel and lightweight algorithm that is based on a deferred index data structure and is faster and uses less memory than the best available algorithms. We show its applicability for mining multiple databases with a variety of frequency constraints. As such, we use the notion of entropy from information theory to devise the *entropy substring mining problem* which is a multiple database generalization of the *emerging substring mining problem*. In addition we evaluate the algorithm rigorously using various string domains, e.g. natural language, DNA, or protein sequences. The experiments demonstrate the improvement of our algorithm compared to recent approaches.

All data structures, algorithms, and tools proposed in this thesis are part of SeqAn the generic C++ template library for sequence analysis, which is publicly available under `http://www.seqan.de/` and supports Linux, Mac OS X, and Windows.

Zusammenfassung

In den letzen Jahren konnte der Sequenzierdurchsatz mit Einführung sogenannter Hochdurchsatz-Sequenziertechnologien dramatisch gesteigert werden. Sie erzeugen mehrere Milliarden Basenpaare pro Tag in Form von Reads der Länge 100 bp und mehr und aktuelle Entwicklungen versprechen das persönliche 1000-Dollar-Genom in den kommenden Jahren. Diese technologischen Fortschritte erfordern neue Ansätze und effiziente Datenstrukturen, die speziell für die Analyse von Massendaten konzipiert sind. Eine solche Datenstruktur ist der Substring-Index, welcher alle Substrings oder Substrings bis zu einer festen Länge repräsentiert, die in einem Text vorkommen.

In dieser Arbeit präsentieren wir drei verschiedene Substring-Indizes und Anwendungen in der Analyse von Hochdurchsatz-Sequenzierdaten. Die ursprünglich auf eine einzelne Sequenz beschränkten Indizes werden zunächst für die Indizierung mehrerer Millionen Sequenzen erweitert. Weiter implementieren wir Algorithmen zum Aufbau von Indizes im Hauptspeicher und entwickeln effiziente Sekundärspeicheralgorithmen für die Indizierung großer Datensätze, bspw. mehrerer Säugetiergenome. Zur einfachen Benutzbarkeit stellen wir ein Framework zur Verfügung, das einen einheitlichen Zugriff auf den verallgemeinerten Suffixbaum erlaubt und benutzen jenes, um exemplarisch 3 Iteratoren zur Repeatsuche zu implementieren. Außerdem stellen wir Algorithmen bereit zur exakten und approximativen Stringsuche mittels indexbasierten Filteralgorithmen oder dem rekursiven Abstieg in Suffixbäumen.

Ferner, stellen wir RazerS vor — ein Programm, das einfache oder gepaarte Reads beliebiger Länge an ihren potentiellen genomischen Ursprung aligniert. RazerS kann entweder vollsensitiv oder mit einer spezifizierten Verlustrate und höherer Geschwindigkeit verwendet werden. Wir stellen einen neuen Ansatz vor, mit dem eine geforderte Mindestsensitivität garantiert werden kann. So wird dem Benutzer ein nahtloser Trade-off zwischen Sensitivität und Laufzeit ermöglicht. Um den Alignment-Suchraum zu verkleinern, haben wir zwei indexbasierte Filter und eine optimierte Variante von Myers' Bitvektoralgorithmus implementiert, und benutzen OpenMP zur Parallelisierung. Wir vergleichen RazerS mit aktuellen Read-Alignment Programmen und zeigen auf verschiedenen Realdatensätzen, dass es die höchste Sensitivität bei vergleichbarer Geschwindigkeit erzielt.

Zuletzt stellen wir einen generischen Ansatz für das frequenzbasierte String Mining vor mit Anwendungen bspw. im kontrastiven Data Mining. Unser Beitrag ist ein neuer Algorithmus, der einen dynamisch aufgebauten Suffixbaum verwendet und schneller und speichersparender ist als die besten verfügbaren Algorithmen. Wir zeigen die Anwendbarkeit für das String Mining mehrerer Datenbanken an Hand einer Reihe von Suchproblemen. Als ein solches führen wir das *entropiebasierte String Mining Problem* als Verallgemeinerung des *Emerging String Mining Problems* ein. Wir bewerten unseren Algorithmus auf verschiedenen Datenbasen, bspw. natürlichsprachlichen Texten, DNA- und Proteinsequenzen. Die Experimente demonstrieren die Verbesserung unseres Algorithmus gegenüber existierenden Ansätzen.

Alle Datenstrukturen, Algorithmen und Programme in dieser Arbeit sind Teil von SeqAn, der generischen C++ Template-Bibliothek für Sequenzanalyse, verfügbar unter `http://www.seqan.de/` und unterstützen Linux, Mac OS X und Windows.

Acknowledgments

Doing a PhD and writing this doctoral thesis would never have been possible without the help and support of many kind people around me. The first and foremost to mention is my advisor Knut Reinert, who taught me how to do research and provided a pleasant and stimulating working environment. Thank you for your guidance, inspirations, unshakable optimism, and many shared laughs. I really appreciate you being my doctoral father.

I want to express my gratitude to Laurent Mouchard who is willing to appraise this thesis and acknowledge the financial support of the Federal Ministry of Education and Research and the NaFöG-grant awarded by the Berlin City Council.

Thank you, Marcel and Hugues, for exciting joint multinational projects, of which I'm sure many will follow, and for the memorable trips to Stockholm and Cambridge. Marcel, don't give up the hope, next time we'll win the "Steps to Success" award. Many thanks to the current and former members of the Bioinformatics and Software Engineering groups at the Freie Universität Berlin. Danke, Markus and Ole, for your great Schmäh and the bad jokes. Our exceptional teamwork made it possible to win the pub quiz consolation prize.

Thank you, Andreas, Anne-Katrin, Birte, Enrico, Manuel, Sandro, and Tobi, for many joint SeqAn projects in the past and future; Sandro, Chris, and Stephan, for delightful evenings in Berlin's best schnitzel and burger restaurants; and all the others for a great time during and after work: Alex, Anja, Christopher, Clemens, Edna, Eva, Gesine, Isabella, Jochen, Julia, Kathrin, Martin, René, Uli, and Sally.

Last but not least, I thank my parents and my family for their continuous and unconditional support. I dedicate this work to Nadine, for backing me up and providing the time for writing, and to Helene and Johan for making me forget the tough days. Johan, don't mind the drowned Macbook, the new one is much better.

"More engineering work has to be done to improve the practical performance of these index structures (...) These implementations should be grouped under a common interface in libraries (...) One such library-project (...) is the SeqAn library."

Vyverman *et al.* [2012]

CONTENTS

Part I Introduction 1

1. Introduction 3
- 1.1 Preface 3
- 1.2 Sanger sequencing 4
- 1.3 High-throughput sequencing technologies 5
- 1.4 Applications of high-throughput sequencing 6
- 1.5 Overview 7
  - 1.5.1 Index data structures 8
  - 1.5.2 Read mapping 9
  - 1.5.3 Frequency string mining 10

2. Mathematical Preliminaries 13
- 2.1 Notations 13
- 2.2 Relations 14
- 2.3 Suffix tree 15
- 2.4 Transcripts and alignments 17
- 2.5 Approximate matching 19

Part II Index Data Structures 23

3. Enhanced Suffix Array 25
- 3.1 Definitions 25
  - 3.1.1 Suffix array 25
  - 3.1.2 LCP table 25
  - 3.1.3 Child table 26
- 3.2 Representation 29
- 3.3 Construction of the suffix array 30
  - 3.3.1 The linear-time algorithm by Kärkkäinen et al. 31
  - 3.3.2 Difference covers 33
  - 3.3.3 Our algorithms 35
  - 3.3.4 External memory variant 39
  - 3.3.5 Extension to multiple sequences 40
- 3.4 Construction of the lcp table 42
  - 3.4.1 The linear-time algorithm by Kasai et al. 43
  - 3.4.2 Space-saving variant 44
  - 3.4.3 Adaptation to external memory 45
  - 3.4.4 Extension to multiple sequences 46

3.5 Construction of the child table 47
3.5.1 Bottom-up suffix tree traversal 47
3.5.2 The linear-time algorithm by Abouelhoda et al. 47
3.5.3 Adaptation to external memory and multiple sequences 49
3.6 Applications 51
3.6.1 Searching the suffix array 51
3.6.2 Traversing the suffix tree 53
3.6.3 Accessing the suffix tree 57
3.6.4 Repeat search 58

4. Lazy Suffix Tree 65
4.1 The *wotd* algorithm 65
4.2 Lazy construction and representation 66
4.2.1 The original data structure 68
4.2.2 Our data structure 70
4.2.3 Extension to multiple sequences 73
4.3 Applications 73
4.3.1 Traversing and accessing the lazy suffix tree 74
4.3.2 Radix trees 76
4.3.3 Multiple exact pattern search 76
4.3.4 Approximate pattern search 78

5. q-gram Index 83
5.1 Definitions 83
5.2 The direct addressing q-gram index 84
5.3 Construction 84
5.3.1 Counting sort algorithm 85
5.3.2 Extension to multiple sequences 85
5.3.3 Adaptation to external memory 86
5.4 The open addressing q-gram index 86
5.5 Applications 88
5.5.1 q-gram counting filters for approximate matching 90

Part III Applications 95

6. Read Mapping 97
6.1 Related work 97
6.2 The RazerS algorithm 100
6.3 Definitions 101
6.4 Filtration 102
6.4.1 SWIFT filter 102
6.4.2 Pigeonhole filter 103
6.5 Lossy filtration and prediction of sensitivity 104
6.5.1 Sensitivity calculation of q-gram counting filters 105
6.5.2 Sensitivity calculation of pigeonhole filters 109
6.5.3 Choosing filtration parameters 110
6.6 Verification 111

6.6.1 Hamming distance verification 111
6.6.2 Edit distance verification 112
6.7 Paired-end mapping 117
6.8 Match processing 118
6.9 Parallelization 119
6.10 Experimental results 119
6.10.1 Comparing the SWIFT and pigeonhole filters 120
6.10.2 Analyzing the sensitivity estimation accuracy 121
6.10.3 Achieved speedup 125
6.10.4 Rabema benchmark results 125
6.10.5 Variant detection results 126
6.10.6 Performance comparison 127

7. Frequency String Mining 131
7.1 Related work 131
7.2 Definitions 132
7.2.1 Predicates 133
7.2.2 Monotonicity 135
7.2.3 Conjunctive predicates 136
7.3 Monotonic hull 136
7.4 The linear-time algorithm by Fischer et al. 137
7.4.1 The original algorithm 137
7.4.2 Space efficient variants 138
7.5 A fast algorithm based on lazy suffix trees 139
7.5.1 The deferred frequency index 139
7.5.2 Algorithmic details 141
7.6 Experimental results 142
7.6.1 Two databases 144
7.6.2 Multiple databases 145
7.6.3 Detection of species specific protein domains 145

8. Conclusion and Future Work 149

A. Appendix 153
A.1 High-throughput sequencing technologies in detail 153
A.2 Proving sensitivity recursions 156
A.3 Read mapper parametrization 157
A.4 Extended variation detection tables 158
A.5 Extended performance comparison tables 159
A.6 Proving hull optimality 163

B. Curriculum Vitae 165

C. Declaration 169

Bibliography 170

Index 195

Part I

INTRODUCTION

Chapter 1 Introduction

1.1 Preface

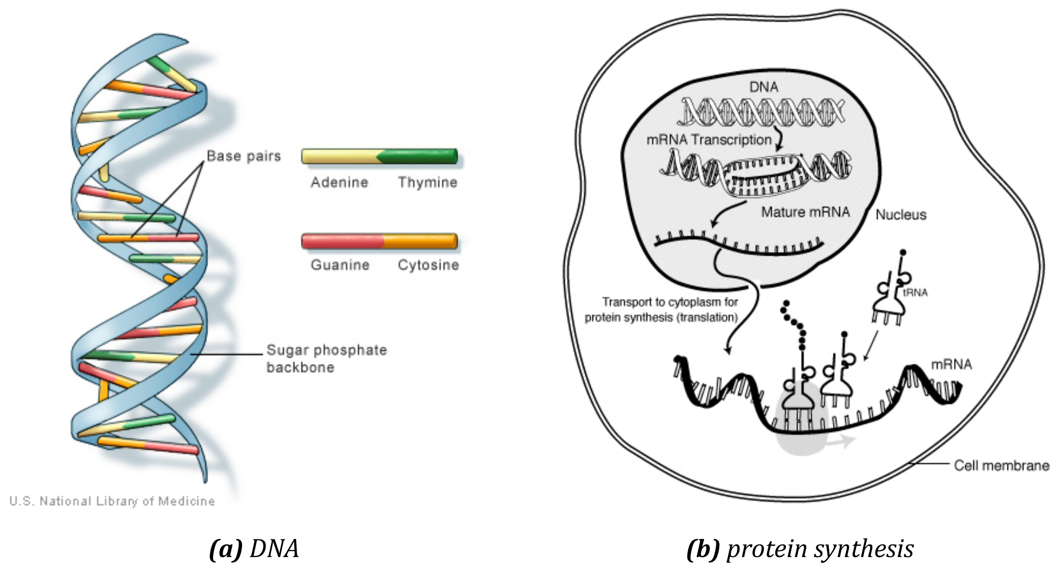
In February 2001, press releases announced that two groups, the publicly funded Human Genome Project [Lander *et al.*, 2001] and the private company Celera Genomics [Venter *et al.*, 2001], independently completed a mammoth project whose challenging aim was nothing less than decoding the entire human genome, i.e. the sequence of the 3,000,000,000 nucleotides contained in the nucleus of each of our cells, defining the shape of our body or our susceptibility to certain diseases.

Sequencing a DNA of that scale was only possible with improvements of the original Sanger sequencing technology, like the automated capillary electrophoresis. However, while it took 10 years and $300–$1000 million to sequence the human genome in those days, the same amount of data can nowadays be sequenced with commercially available high-throughput sequencers in a couple of days for less than $10,000, and current developments of sequencing technologies promise the personal $1,000 genome in a couple of years.

The sequence of the whole genome, the set of all DNA molecules in a cell, provides insights into the mechanisms of inheritance and evolutionary history of an organism. Comparison studies of human genomes allow to detect single nucleotide polymorphisms (SNPs) or large structural variations and to associate them with specific diseases. Those associations allow to improve diagnoses, to earlier detect genetic predispositions to common diseases, and to develop gene therapies or personalized medicine.

DNA is a double-stranded polymer, two chains that entwine in the shape of a double helix. The chains are composed of four building blocks, the nucleotides adenine (`A`), cytosine (`C`), guanine (`G`), and thymine (`T`). The nucleotides in both strands are complementary and form base pairs (bp) via hydrogen bonds, i.e. `A` is linked to `T` and `C` is linked to `G`, see Figure 1.1a. Thus, one strand can be reconstructed from the other, which is done to replicate DNA during cell division. The strands have a direction and are antiparallel. Each is read from five prime (5’) to three prime (3’) end, referring to the fifth and third carbon atom in the sugar rings of the DNA backbone. In cells of human and other diploid organisms the DNA is organized in chromosome pairs (23 in human) consisting of recombinant maternal and paternal chromosomes.

The DNA is the carrier of all genetic information and consists of thousands of genes, the blueprints of protein molecules. Proteins play an important role in almost all cell functions, e.g. as enzymes that catalyze metabolic pathways, as signal transducers, or as



***(a)** DNA*

***(b)** protein synthesis*

Figure 1.1: *The DNA (a) consists of two anti-parallel strands of complementary nucleotides (base pairs). DNA is transcribed to mRNA and outside the cell nucleus translated into proteins (b).*

contractile proteins involved in muscle contraction. When a protein needs to be synthesized, the corresponding genes are transcribed into RNA, a single-stranded polymer consisting of the same building blocks as DNA except for thymine which is replaced by uracil (U). After non-coding parts are removed (splicing), the RNA is prepared and transported outside the cell nucleus, where it is then translated into a chain of amino acids, the building blocks of proteins. Besides its function as a messenger (mRNA) that encodes a protein, RNA is responsible for regulating the expression of genes or the transport of amino acids (tRNA) during protein synthesis, see Figure 1.1b.

High-throughput RNA sequencing (RNA-seq) reveals the sequence and relative quantity of RNA molecules present in a specific cell, tissue, or organ. It provides insights into the mechanisms of gene regulation and how these mechanisms are disturbed in tumor cells. Furthermore, RNA-seq enables identification of novel genes and different transcript isoforms, i.e. different variants how exons of a gene are spliced into mRNA and the different protein isoforms they encode. RNA molecules can be sequenced with any DNA sequencing technology after a prior synthesis of the missing complementary strand with the enzymes DNA polymerase and reverse transcriptase.

1.2 Sanger sequencing

The foundation for today's DNA sequencing methods was laid by Sanger *et al.* [1977]. Sanger and coworkers invented a sequencing method based on chain-terminators that stop the polymerase chain reaction (PCR). DNA polymerase is an enzyme involved in the replication of DNA during cell division. After the double-stranded DNA is unwound

technology	instrument	run time	yield [Mb/run]	read length [bp]	costs [$/Mb]	error rate [%]
Sanger	3730xl (capillary)	2 h	0.06	650	1500	0.1–1
Illumina	HiSeq 2000	8 d	200,000	2 × 100	0.10	≥ 0.1
SOLiD	SOLiD 4	12 d	71,400	50 + 35	0.11	> 0.06
Roche/454	FLX Titanium	10 h	500	400	12.4	1
SMRT™	PacBio RS	0.5–2 h	5–10	860–1100	11–180	16
HeliScope™	Helicos	N/A	28,000	35	N/A	N/A

Table 1.1: *Approximate run times, yields, read lengths, costs, and sequencing error rates of different high-throughput sequencing technologies by mid 2011 [Glenn, 2011].*

and dehybridized into two single strands, polymerase replicates the complementary of each strand by sequentially incorporating complementary nucleotides. For the first time, Sanger *et al.* utilized terminating nucleotides that immediately stop the replication after incorporation. The DNA template is first replicated multiple times (amplification) and then replicated in four solutions (`A`, `C`, `G`, and `T`) each of which contain all nucleotides and one terminating nucleotide in low concentration. After that, each solution contains entire copies and partial copies that end with the known terminating nucleotide, e.g. the solution with all four nucleotides and the terminating `C` contains prefixes of the complementary template strand that end with a `C`. A subsequent gel electrophoresis of the four sets is used to separate the prefixes by their lengths and directly reveals the template sequence. In the gel, a molecule migrates with a speed inversely proportional to its length. Thus the maximal read length that can be sequenced by the Sanger method is limited to 300–1000 nucleotides; for longer reads the relative length difference of one nucleotide is not accurately measurable.

In DNA sequencing by capillary electrophoresis, as used for sequencing the human genome, the four terminating nucleotides are marked with four fluorescent dyes such that only one instead of four separate reactions are necessary per template. Capillaries filled with gel replace the slab gel used in Sanger sequencing and enable sequencing many templates in parallel. At the end of each capillary, a laser and detector determines the terminating nucleotide of the molecules that leave the capillary in the order of their size.

1.3 High-throughput sequencing technologies

During the last years, sequencing throughput increased dramatically with the introduction of so-called *high-throughput sequencing* (HTS), also known as deep sequencing or next generation sequencing (NGS). It allows the production of billions of base pairs (bp) per day in the form of reads of length 100 bp and more. Since 2004, when 454 Life Sciences released the first commercially available HTS sequencing machine, throughput continues to increase and new technologies provide longer reads than currently available. Moreover, the sequencing costs decrease more rapidly than the costs for hard disk storage or Moore's law for computing costs [Stein, 2010].

Currently available high-throughput sequencing platforms are ABI SOLiD (Life Tech-

nologies Corp.), HeliScope™ (Helicos Biosciences Corp.), Illumina (Illumina Inc.), SMRT™ (Pacific Biosciences Inc.), and Roche/454 (Roche Diagnostics Corp.). See Table 1.1 for a comparison of their throughputs, run times, and expendable costs. Compared to sequencing by gel electrophoresis, a key improvement is to cycle and image the incorporation of nucleotides or to detect the incorporation in real time. Replacing the gel tremendously reduced the sequencing costs and has made it possible to miniaturize and parallelize sequencing. Common to all technologies is that the DNA is first fractionated into smaller double-stranded fragments, which are optionally amplified, and then sequenced in parallel from one end. Additionally, most of the technologies provide a so-called *paired-end sequencing* protocol in which the fragments are sequenced from both ends. More details about the technologies mentioned above can be found in Appendix A.1.

One of the largest publicly available databases for nucleotide-sequence data is the European Nucleotide Archive [Leinonen *et al.*, 2011]. It contains the Sequence Read Archive [Kodama *et al.*, 2012] which was established in 2009 for the purpose of providing access to raw data from high-throughput sequencing platforms for the wider research community. In 2011 the amount of publicly available sequencing data exceeded 100 trillion base pairs of which 84 % account for the Illumina platform, whereas ABI/SOLiD and Roche/454 comprised 12 % and 2 % respectively.

1.4 Applications of high-throughput sequencing

High-throughput whole-genome sequencing has become an invaluable technology for a multitude of applications, e.g. the detection of SNPs [Hillier *et al.*, 2008; Bentley *et al.*, 2008; Ley *et al.*, 2008; Wang *et al.*, 2008] and large structural genome variations [Chen *et al.*, 2008], or for reference guided [Wang *et al.*, 2008] and de novo genome assembly [Li *et al.*, 2010; Simpson and Durbin, 2012]. Sequencing environmental samples makes it possible to detect the contained organisms in metagenomic assays [Huson *et al.*, 2007; Rodrigue *et al.*, 2010]. The RNA-seq protocol, in which RNA is reverse transcribed into cDNA and sequenced, enables the identification of genes and alternative splicing events either annotation based [Richard *et al.*, 2010; Roberts *et al.*, 2011] or de-novo [Robertson *et al.*, 2010; Adamidi *et al.*, 2011; Schulz *et al.*, 2012], and to quantify their abundance and analyze gene expression levels [Mortazavi *et al.*, 2008; Montgomery *et al.*, 2010; Trapnell *et al.*, 2010]. Chromatin-immunoprecipitation of DNA followed by high-throughput sequencing (ChIP-seq) provides information on interactions between proteins and DNA, e.g. to identify transcription factor binding sites [Schmidt *et al.*, 2010], histone modification patterns [Barski *et al.*, 2007], or methylation patterns [Meissner *et al.*, 2008].

A common challenge in all these applications is to efficiently compare large amounts of sequences against each other, be it to search for the genomic origin of sequenced reads in a reference genome (read mapping), to find overlaps between sequenced reads (sequence assembly, read error correction), or to find sequence motifs that either have been conserved in different organisms (genome alignment) or are specific for a certain organism, disease, or transcription factor (frequency string mining). Depending on the type of sequences, the comparison can either be exact or needs to be tolerant to different types

of errors.

The first type of errors are sequencing errors resulting from wrong base calls. All technologies that use clusters for signal amplification (Illumina, SOLiD, Roche/454) require that each cluster contains thousands of identical molecules which are sequenced synchronously. In a synchronization loss the combined signal of one cluster consists of the sum of signals of the current base and their neighbors in the DNA template and leads to an increase of mis-calls towards the end of the reads. Technologies that use the signal intensity (Roche/454) or length (SMRT™) to determine the length homopolymer runs cannot reliably detect the length of large runs due to the resolution limit or speed variations of the DNA polymerase. Such technologies typically produce reads with insertions or deletions in homopolymer runs. Deletions of bases may also occur in technologies that omit a prior template amplification (HeliScope™, SMRT™) due to undetected signals. However, errors in the sequencing process can be discerned by using base-call quality values and redundancy like a high base coverage, e.g. 20 and higher [Dohm *et al.*, 2008].

The second type of error results from variations in the DNA between different organisms or between different cells of the same organism. These variations can be due to mutations in DNA which are part of the evolutionary process. Errors in the replication of DNA cause substitutions, deletions and insertions of nucleotides.

The recent advances in sequencing technology demand for novel approaches and efficient data structure specifically designed for the efficient analysis of mass data. In this work, we propose such data structures, called *(substring) indices*, that represent all substrings or substrings up to a certain length contained in a given text. Indices have applications in almost all above-mentioned HTS applications. We use them in filters to efficiently discard dissimilar regions and reduce the number of costly sequence alignments in DNA read mapping [Weese *et al.*, 2009, 2012], short RNA read mapping [Emde *et al.*, 2010], structural variation detection [Emde *et al.*, 2012], fast local alignment [Kehr *et al.*, 2011], or reference guided assembly [Rausch *et al.*, 2009]. Their applicability to repeat search in multiple large sequences allows us to efficiently find identical, conserved regions and extend them to multiple genome alignments [Rausch *et al.*, 2008]. We use their suffix tree representation to efficiently count frequencies in multiple databases for frequency based string mining [Weese and Schulz, 2008], to anchor reads with identical substrings to correct sequencing errors without a reference genome [Weese *et al.*, 2013], or to construct variable order Markov chains [Schulz *et al.*, 2008a].

1.5 Overview

In this thesis, we describe the design and implementation of three different index data structures and prove their applicability in two HTS applications. Chapter 2 gives fundamental mathematical definitions required throughout this thesis. Beginning with essential definitions of strings and alphabets in Section 2.1, we define orders on them in Section 2.2. Section 2.3 introduces the suffix tree, a fundamental index of all substrings of a single string or multiple strings. To be able to compare substrings with errors, we define string distances and how to efficiently compute them in Section 2.4. Finally, we

define the approximate pattern matching problem in Section 2.5 and how to find all text occurrences of a pattern within a given string distance via dynamic programming. The subsequent parts cover our own contributions, whereby Part II proposes three index data structures for high-throughput sequencing and Part III shows how to apply them to develop efficient HTS applications that can compete with available state-of-the-art tools.

All data structures, algorithms, and tools proposed in this thesis have been integrated into SeqAn [Döring *et al.*, 2008] the generic C++ template library for sequence analysis, which is publicly available under `http://www.seqan.de/` and supports Linux, Mac OS X, and Windows.

1.5.1 Index data structures

The suffix tree, first proposed by Weiner [1973], plays an important role in sequence analysis and comparative genomics. It represents all substrings of a text of length n in $\mathcal{O}(n)$ memory, can be constructed in $\mathcal{O}(n)$ time[1] [Weiner, 1973], and supports exact string searches in optimal time. In the following years, practically faster linear-time construction algorithms were proposed that use less memory [McCreight, 1976] or read the text in a sequential scan [Ukkonen, 1995]. However, its space consumption of roughly $20n$ bytes makes it inapplicable to large analyses of whole genomes.

In Chapter 3, we describe the enhanced suffix array [Abouelhoda *et al.*, 2002a], a more memory efficient representation of the suffix tree, and define in Section 3.1 and 3.2 the three contained tables: suffix array, lcp table, and child table. In the following sections (3.3–3.5), we show how to construct them in linear time and contribute new algorithmic variants for texts consisting of multiple strings and the efficient construction of large indices in external memory. In Section 3.6, we describe how to search the enhanced suffix array and contribute an easy-to-use iterator interface that allows traversing and accessing the suffix tree represented by the enhanced suffix array. We additionally provide three application-specific iterators for searching repeats.

The second index is the lazy suffix tree [Giegerich *et al.*, 2003], a deferred data structure proposed in Chapter 4. This deferred data structure is top-down constructed on demand and a more efficient alternative to the enhanced suffix array for applications where only an upper fraction of the suffix tree needs to be traversed. After proposing the original lazy suffix tree data structure and its construction algorithm in Sections 4.1 and 4.2, we introduce a new lazy suffix tree which is applicable to multiple strings and creates suffix tree nodes in lexicographical order. Providing the same suffix tree iterator interface we make it a transparent replacement of the enhanced suffix array in Section 4.3. Moreover, it enables to sample the suffixes used for the suffix tree construction, e.g. to construct a radix tree of multiple strings. At the end of the chapter, we show how to use two suffix trees in parallel for multiple approximate pattern matching.

Much simpler but adequate for many applications is the q-gram index introduced in Chapter 5. Its functionality is limited to counting and retrieving all occurrences of fixed-length patterns but it can be constructed and accessed much faster than the two previous

[1] assuming a constant-sized alphabet

indices. We first define contiguous and gapped q-grams in Section 5.1 and how to compute their ranks, which are required to address buckets of the direct addressing q-gram index described in Section 5.2. Since the $\mathcal{O}(|\Sigma|^q)$ memory footprint of the direct addressing index may become prohibitive for large alphabets or large values of q, we introduce the open addressing q-gram index in Section 5.4 which has a memory consumption linear in the size of the text. The chapter concludes with applications of q-gram indices in Section 5.5. We describe two filters based on q-gram counting capable of accelerating approximate pattern matching algorithms.

1.5.2 Read mapping

One of the challenges imposed by the new sequencing technologies is the so-called *read mapping problem* which is the first fundamental step in almost all sequencing-based assays. It is to find the genomic position of each sequenced read in a known, so-called *reference*, genome. Knowing the genomic origins of all reads enables in further downstream analyses the identification of structural variations (Figure 1.2), e.g. SNPs can be detected and distinguished from sequencing errors given a sufficiently high coverage [Dohm *et al.*, 2008]; analyzing the read coverage of the reference reveals regions that have been deleted or repeated in the sequenced genome; and unmapped or partially mapped reads may indicate an insertion in the sequenced genome. In RNA-seq or ChIP-seq experiments the read coverage can be used to detect unknown exons or protein-DNA-interactions. Mapping reads sequenced from environmental samples to a database of different reference genomes allows detection and abundance estimation of the contained organisms from individual coverages. In the reference guided assembly, reads that overlap in the reference are used to determine contiguous sequences (contigs) of the sequenced genome.

Solving the read mapping problem requires to overcome numerous related issues:

- To incorporate errors between reads and reference, resulting from base miscalls and differences between reference and sequenced genome, the reads have to be aligned semi-globally while tolerating a certain number of mismatches and indels.
- As sequenced reads typically stem from both DNA strands, they must be aligned to the forward strand and its reverse complement.
- Some reads originate from repetitive regions of the DNA and cannot uniquely be aligned to a single position in the reference. In this case, all possible positions should be reported.

The yield of hundreds of gigabases per sequencing run makes traditional alignment programs like BLAST [Altschul *et al.*, 1990] or the most efficient tools for mapping capillary reads like SSHAHA [Ning *et al.*, 2001] and BLAT [Kent, 2002] impractical to use.

In the last years many tools have been published for mapping short reads (30–60 bp) that exploit characteristics of a specific sequencing technology, e.g. the low indel error rate [Cox, 2006] or high-quality bases at the 5'-ends of Illumina reads [Li *et al.*, 2008a],

to outperform classic alignment tools in terms of speed. However, these tools often sacrifice accuracy for speed and are only applicable to the technology they were developed for. As a consequence, current read mappers have difficulties to map long reads with a high number of errors with a high sensitivity. Moreover, many use heuristics that lack a clear definition of the problem they solve, are hard to parametrize according to a specific biological problem, or output only best matches under complicated and hardwired rules [Li *et al.*, 2008a; Li and Durbin, 2009].

We give a detailed overview of existing read mappers and their characteristics in Section 6.1. In the subsequent sections, we formally define the read mapping problem we consider and propose the algorithmic ideas of RazerS, an efficient read mapping tool that allows the user to align single or paired-end reads of arbitrary length using either Hamming distance or edit distance. Our tool can work either lossless or with a user-defined loss rate at higher speeds. Given the loss rate, we present an approach that guarantees not to lose more reads than specified. This enables the user to adapt to the problem at hand and provides a seamless tradeoff between sensitivity and running time. RazerS utilizes the q-gram index and q-gram based filters, which we describe in Chapter 5, a banded variant of Myers' [1999] bit-vector algorithm, and multi-core parallelization. We evaluate the performance of our approach in comparison to other state-of-the-art read mapping tools in various real-world experiments in Section 6.10.

1.5.3 Frequency string mining

The storage of sequences in databases alone does not guarantee that all hidden information is readily available. A promising approach for knowledge discovery in databases is to mine frequent patterns, which was reviewed by Han *et al.* [2007]. This general paradigm can be applied in many application domains. For example, Hu and Liu [2004] search for frequent patterns to condense opinions of customers from positive and negative product reviews, whereas others suggest mining of customer data to optimize marketing strategies [Berry and Linoff, 1997]. Kobyliński and Walczak [2009] utilize frequent patterns in the context of image classification by mining vertical, horizontal, and diagonal sequences of discretized color and texture features of images. In [Birzele and Kramer, 2006] frequent patterns are used as features for classification of protein secondary structures. Other applications are the design of microarray probes that allow differentiation of groups of sequences under investigation [Fischer *et al.*, 2005] or the discovery of binding motifs of transcription factors [Mitašiūnaitė *et al.*, 2008].

A gene is regulated by proteins, so-called transcription factors, that bind to its promoter sequence. A common approach is to contrast promoter sequences of genes that are believed to be regulated by the same factor with promoters of unrelated genes to detect the transcription factor's binding motif. The rationale behind this is to find sequence motifs that are representative (frequent) for one set of sequences and absent (infrequent) in another, a method called discriminatory or contrast data mining [Redhead and Bailey, 2007; Fischer *et al.*, 2005; Han *et al.*, 2007].

In Chapter 7, we focus on string mining under frequency constraints and define in Section 7.2 predicates that evaluate solely the frequency of a pattern, i.e. the number of

Figure 1.2: *Multiple read alignment. Blue and red arrows represent reads mapped to the forward or reverse strand of a reference sequence (top line). The arrows are interspersed by dots or bases indicating gaps or mismatches in the alignment. The example shows an insertion of* `AATT` *supported by half of the overlapping reads indicating a mutation in one of the two diploid chromosomes. The second mutation is a SNP where a* `C` *in the reference was replaced by* `G` *in the sequenced genome.*

distinct sequences in a database that contain the pattern at least once. Frequency string mining was motivated by approaches for mining sets of items [Agrawal *et al.*, 1993; Han *et al.*, 2004]. There have been different definitions of the string domains being sought, e.g. gapped strings [Ji *et al.*, 2007] or approximate strings [Mitašiūnaitė *et al.*, 2008], while in this work we consider exact substrings of sequences.

Various algorithmic approaches to frequency string mining have been published over the last years. The first optimal algorithm was proposed by Fischer *et al.* [2006]. It is based on enhanced suffix arrays and quite fast in practice. We explain the fundamental idea and memory efficient variants in Section 7.4. In 2008, we presented a conceptually much simpler algorithm which is based on a lazy suffix tree, practically faster, and uses less memory at the same time [Weese and Schulz, 2008]. We give an in-depth presentation of this algorithm in Section 7.5 and show how to use it on multiple databases with a variety of frequency constraints. As such, we use the notion of entropy from information theory to devise the *entropy substring mining problem* (Section 7.2) which is a multiple database generalization of the *emerging substring mining problem* [Chan *et al.*, 2003]. In Section 7.6, we evaluate the performance of our implementation in comparison to other approaches on real-world datasets of various string domains, e.g. natural language, DNA,

or protein sequences. The experiments demonstrate the improvement of our algorithm in terms of running time and applicability to arbitrary frequency predicates on multiple databases.

Chapter 2 Mathematical Preliminaries

This chapter introduces data structures that we will use throughout the thesis. First, Section 2.1 gives some fundamental notations, whereas Section 2.2 introduces the lexicographical order which we will use to define the suffix tree in Section 2.3 and the suffix array in Chapter 3. Section 2.4 defines alignments, transcripts, and distances that are fundamental to read mapping and our sensitivity estimation approach in Chapter 6.

2.1 Notations

Let Σ be a non-empty, finite alphabet. A *string* over Σ is a finite sequence of characters from Σ. Σ^n denotes the set of all strings of length n over the alphabet Σ. Moreover, we define $\Sigma^* := \bigcup_{i=0}^{\infty} \Sigma^i$ the set of all finite strings over Σ where $\Sigma^0 = \{\epsilon\}$ and ϵ is the empty string. For $s \in \Sigma^n$ we denote the length of s by $|s| = n$. The concatenation of two strings s and t is denoted by st or $s \cdot t$. We define an *array* or a *table* to be a string over the alphabet $\mathbb{N}_0$, the set of non-negative integers. Strings of length q are also called q-tuples or q-grams. In the following, we use a zero-based indexing and define:

Definition 2.1 (substrings). Let $s \in \Sigma^n$ be a string and $i, j \in \mathbb{N}_0$ with $i \leq j$.

- $s[i]$ denotes the $(i+1)$'th character of s.
- $[i..j] := \{i, i+1, \dots, j\}$
- $[i..j) := [i..j-1]$
- $s[i..j] := s[i]s[i+1] \dots s[j]$ is called substring or infix of s.
- $s[i..j) := s[i..j-1]$
- $s_i := s[i..n)$ is called *suffix* of s. We may also write $\text{suf}(s, i)$.
- $s[0..i)$ is called prefix of s.
- $[j..i-1] := \emptyset$, $s[j..i-1] := \epsilon$.

For two strings $s, t \in \Sigma^*$ we write $s \preceq t$ if s is a substring of t, and $s \prec t$ if $s \neq t$ holds in addition. For single characters $x \in \Sigma$, we write $x \in s$ equivalently to *s contains a character x*. Analogously to the notation in [Dementiev *et al.*, 2008a] we canonically

extend the set definition to strings, e.g. $\langle\, 2i{+}1 \mid i \in [0..5) \,\rangle$ denotes the string of increasing odd numbers between 0 and 10.

Definition 2.2 (longest common prefix)**.** Given a non-empty set of strings $\mathcal{S} \subset \Sigma^*$. The string $p \in \Sigma^*$ is a *common prefix* of $\mathcal{S}$, if for every $s \in \mathcal{S}$ there exists a string $q \in \Sigma^*$ such that $s = pq$. The *longest common prefix* of $\mathcal{S}$ is uniquely defined and denoted by lcp $\mathcal{S}$.

2.2 Relations

Definition 2.3 (Cartesian order)**.** Given orders $<_1$ and $<_2$ on the sets M_1 and M_2. Unless otherwise stated, we define $<$ an order on the Cartesian product $M_1 \times M_2$ such that for any $(a_1, a_2), (b_1, b_2) \in M_1 \times M_2$ holds:

$$(a_1, a_2) < (b_1, b_2) \quad \Leftrightarrow \quad (a_1 <_1 b_1) \vee (a_1 = b_1 \wedge a_2 <_2 b_2). \tag{2.1}$$

For the Cartesian product $M_1 \times M_2 \times \ldots \times M_m$ of more than 2 sets we define $<$ recursively such that for any $(a_1, a_2, \ldots, a_m), (b_1, b_2, \ldots, b_m) \in M_1 \times M_2 \times \ldots \times M_m$ holds:

$$(a_1, a_2, \ldots, a_m) < (b_1, b_2, \ldots, b_m) \quad \Leftrightarrow \quad \big(a_1, (a_2, \ldots, a_m)\big) < \big(b_1, (b_2, \ldots, b_m)\big). \tag{2.2}$$

Let $<$ be a strict total order defined on Σ, i.e. for any $a, b \in \Sigma$ exactly one of $a < b$, $b < a$ or $a = b$ holds. We will now transform the relation over characters to a relation over strings.

Definition 2.4 (lexicographical order)**.** Let $s, t \in \Sigma^* \setminus \{\epsilon\}$ be two strings. The lexicographical order $<_{\text{lex}}$ on Σ^* is recursively defined as follows:

$$\neg\big(\epsilon <_{\text{lex}} \epsilon\big), \tag{2.3}$$
$$\epsilon <_{\text{lex}} s, \tag{2.4}$$
$$\neg\big(s <_{\text{lex}} \epsilon\big), \tag{2.5}$$
$$s <_{\text{lex}} t \quad \Leftrightarrow \quad \big(s[0], s_1\big) < \big(t[0], t_1\big). \tag{2.6}$$

With this definition $<$ becomes a strict total order on Σ^*, the *lexicographical order*. In addition to the lexicographical order we define the so-called *lexicographical prefix order* that compares at most the first q characters as $<_q$.

Definition 2.5. For a $q \in \mathbb{N}$ the lexicographical prefix order $<_q$ is defined on Σ^* such that for $s, t \in \Sigma^*$ holds:

$$s <_q t \quad \Leftrightarrow \quad s\big[0..\min(q, |s|)\big) < t\big[0..\min(q, |t|)\big), \tag{2.7}$$
$$s =_q t \quad \Leftrightarrow \quad s\big[0..\min(q, |s|)\big) = t\big[0..\min(q, |t|)\big). \tag{2.8}$$

For any $<$ order we define corresponding orders $>$, $\leq$, and $\geq$ as follows:

Definition 2.6. For an order $<$ on a set M and any $s, t \in M$ the orders $>$, $\leq$, and $\geq$ are defined as:

$$s > t \quad \Leftrightarrow \quad t < s, \tag{2.9}$$

$$s \leq t \quad \Leftrightarrow \quad s < t \ \vee \ s = t, \tag{2.10}$$

$$s \geq t \quad \Leftrightarrow \quad s > t \ \vee \ s = t. \tag{2.11}$$

The lexicographical order allows us to associate each element of a set of strings with its rank:

Definition 2.7 (lexicographical naming)**.** Given a finite set of strings $\mathcal{S} \subseteq \Sigma^*$. A function $\tau : \mathcal{S} \to [0, |\mathcal{S}|)$ is called a *lexicographical naming* of $\mathcal{S}$, if for any $s, t \in \mathcal{S}$ holds:

$$s <_{\text{lex}} t \quad \Leftrightarrow \quad \tau(s) < \tau(t). \tag{2.12}$$

The function τ is bijective and uniquely defined by the lexicographical order. For a string $s \in \mathcal{S}$ we call $\tau(s)$ the *lexicographical name* or *rank* of s (in $\mathcal{S}$). The following lemma allows to reduce the lexicographical order of concatenated tuples to the lexicographical order of strings of tuple names, the fundamental idea of the suffix array construction algorithm in Section 3.3.

Lemma 2.1. *Given a set $\mathcal{S} \subseteq \Sigma^q$ of strings having length q and a lexicographical naming τ for $\mathcal{S}$. Let $X_1, \dots, X_k \in \mathcal{S}$ and $Y_1, \dots, Y_l \in \mathcal{S}$ be strings from $\mathcal{S}$. The lexicographical relation of the concatenated strings $X_1 \cdot X_2 \cdots X_k$ and $Y_1 \cdot Y_2 \cdots Y_l$ equals the lexicographical relation of the strings of names:*

$$\begin{aligned} & X_1 \cdot X_2 \cdots X_k & <_{lex} \quad & Y_1 \cdot Y_2 \cdots Y_l \\ \Leftrightarrow \quad & \tau(X_1)\tau(X_2) \dots \tau(X_k) & <_{lex} \quad & \tau(Y_1)\tau(Y_2) \dots \tau(Y_l). \end{aligned} \tag{2.13}$$

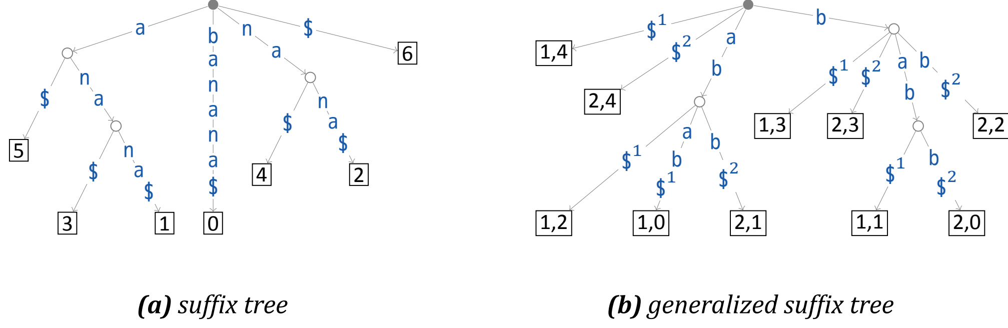
Proof. Trivial proof by induction over k and l using the definition of the lexicographical order. ∎

Example 2.1. Assume $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $\mathcal{S} = \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$, and the lexicographical naming $\tau(\mathtt{aa}) = 0$, $\tau(\mathtt{ab}) = 1$, $\tau(\mathtt{ba}) = 2$, and $\tau(\mathtt{bb}) = 3$. Now any concatenation of strings in $\mathcal{S}$ can be compared by comparing the concatenation of names:

$$\begin{aligned} & \mathtt{abaababaab} & <_{\text{lex}} \quad & \mathtt{abaabbaa} \\ \Leftrightarrow \quad & \mathtt{10221} & <_{\text{lex}} \quad & \mathtt{1030}. \end{aligned} \tag{2.14}$$

2.3 Suffix tree

A suffix tree is a data structure that represents all substrings of a string. To well-define the suffix tree of a string over the alphabet Σ it is necessary to append a (virtual) sentinel character that is smaller than every other alphabet character and not contained in the string to prevent a suffix from occurring more than once in s. This character is *virtual* as it not used in any implementation described in this thesis.



***(a)** suffix tree*

***(b)** generalized suffix tree*

***Figure 2.1:** The suffix tree of s = banana (a) and the generalized suffix tree of the strings s^1 = abab and s^2 = babb (b). In this example, the leaves are labeled with string positions i or (j, i) representing the suffixes* $\mathrm{suf}(s\$, i)$ *or* $\mathrm{suf}(s^j\$^j, i)$.

Definition 2.8 (suffix tree)**.** The suffix tree $ST(s)$ of a string $s \in \Psi^n$ is a rooted tree whose edges are labeled with strings over $\Sigma := \Psi \cup \{\$\}$, where \$ is a sentinel character with $\$ \notin \Psi$ and $\forall_{x \in \Psi}\ \$ < x$. The suffix tree fulfills the following properties:

1. Each internal node is *branching*, i.e. it has at least two children.

2. For branching nodes the labels of outgoing edges begin with distinct characters.

3. The suffix tree has $n + 1$ leaves numbered from 0 to n. The concatenation of edge labels from the root to leaf i yields the suffix $\mathrm{suf}(s\$, i)$.

As a consequence of property 2 every tree node v can uniquely be identified by the concatenation of edge labels on the path from the root to v. For a node v, we denote this string by *concat*(v) and call it the *concatenation string* or *representative* of v. Vice versa we denote with $\overline{\alpha}$, if existent, the tree node whose concatenation string is α. Implementations of suffix trees can take advantage of edge labels being substrings by storing only begin and end positions. In this way, suffix trees of strings with length n can be stored in $\mathcal{O}(n)$ memory. A *suffix trie* is defined by omitting property 1 in Definition 2.8 and labeling edges with characters instead of strings.

We will now extend the definition of the suffix tree from one to multiple strings and define the so-called *generalized suffix tree*. Again it is necessary to introduce a distinct sentinel character $\$^j$ for every string.

Definition 2.9 (generalized suffix tree)**.** The (generalized) suffix tree $ST(s^1, \dots, s^m)$ of multiple strings $s^1, \dots, s^m \in \Psi^*$ is a rooted tree whose edges are labeled with strings over $\Sigma := \Psi \cup \{\$^1, \dots, \$^m\}$, where $\$^j$ is a sentinel character with $\$^j \notin \Psi$ and $\$^1 < \$^2 < \dots < \$^m < x$ for all $x \in \Sigma$. The generalized suffix tree fulfills the following properties:

1. Each internal node is branching.

2. For branching nodes the labels of outgoing edges begin with distinct characters.

3. Leaves are labeled with pairs (j, i) such that $j \in [1..m]$ and $i \in [0..|s^j|]$. The concatenation of edge labels from the root to a leaf (j, i) yields the suffix $\mathrm{suf}(s^j\$^j, i)$.

Analogously to the suffix tree for one string, edge labels can be stored by string positions which is a string number and pair of begin and end position in the string. Figure 2.1 shows a suffix tree of one string and a generalized suffix tree of two strings. Please note that the implementations described in this thesis store only non-empty suffixes of s, i.e. suffix tree nodes whose concatenation strings are sentinels are implicitly removed from the suffix tree.

2.4 Transcripts and alignments

In this section, we define distance and similarity measures on strings and show how to efficiently compute them. As introduced in the Section 1.4, base miscalls during sequencing or mutations between sample and reference genome require to tolerate errors, e.g. when searching a sequenced read in the reference genome, or comparing homologous genes or whole genomes of different organisms.

This can be done by tolerating a certain number of edit operations, i.e. replacements, deletions, and insertions, to transform the read into a genomic substring. We define a *transcript* as a sequence of matches and edit operations to transform one string into another, see Figure 2.2.

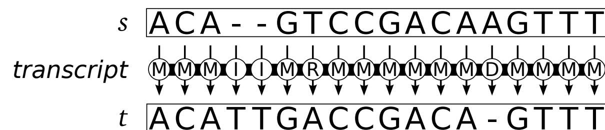


Figure 2.2: *A transcript from string s to string t. The upper and lower sequences are the two rows of the corresponding alignment matrix.*

Definition 2.10 (transcript)**.** For two strings $s, t \in \Sigma^*$, a *(edit) transcript* from s to t is a string over the alphabet $\Phi = \{\mathtt{M}, \mathtt{R}, \mathtt{D}, \mathtt{I}\}$ that describes a transformation between the two strings. The transcript is read and applied from left-to-right to single characters of s to produce t, whereby $\mathtt{M}, \mathtt{R}, \mathtt{D}$, and $\mathtt{I}$ correspond to a match (no change), a replacement, a deletion, and an insertion of a character in s.

For any transcript T we define $\|T\|_{\mathrm{E}} = \left|\{i \mid T[i] \in \{\mathtt{R}, \mathtt{D}, \mathtt{I}\}\}\right|$, the number of errors in T. The *edit distance*, also called Levenshtein [Levenshtein, 1966] distance, between two strings is the minimum number of errors in transcripts between these strings. A special case is the Hamming transcript with $\Phi = \{\mathtt{M}, \mathtt{R}\}$. It is defined uniquely for two strings of equal length and the *Hamming distance* is the number of errors in it. The distances are metrics and for strings s and t we denote the edit distance as $d_{\mathrm{E}}(s, t)$ and the Hamming distance as $d_{\mathrm{H}}(s, t)$. As we will show in the following, there is a one-to-one relationship between global pairwise alignments and transcripts.

Definition 2.11 (alignment)**.** For strings $s_1, \ldots, s_m \in \Sigma^*$, a *(global) multiple alignment* is an m-row matrix $A = (a_{ij})$ such that:

1. All matrix elements a_{ij} are characters from $\Sigma \cup \{\texttt{-}\}$.

2. The i-th matrix row is the string s_i interspersed with gap characters -.

3. There is no column with gaps in all rows.

The elements of an alignment column are aligned to each other and as there is no column containing only gap characters, the alignment has at most $\sum_{i=1}^{m} |s_i|$ columns.

Definition 2.12 (transcript-alignment-equivalence)**.** An alignment A of two sequences s_1 and s_2, also called pairwise alignment, corresponds to a transcript T from s_1 to s_2 if the following holds:

1. The transcript length is the same as the number of matrix columns.

2. For each matrix column $\binom{a_{1j}}{a_{2j}}$ the corresponding transcript character is:

$$T[j-1] \quad = \quad \begin{cases} \texttt{I}, & \text{if } a_{1j} = \texttt{-} \\ \texttt{D}, & \text{if } a_{2j} = \texttt{-} \\ \texttt{M}, & \text{if } a_{1j} = a_{2j} \\ \texttt{R}, & \text{else.} \end{cases} \tag{2.15}$$

It is obvious to see that transcripts are only a different representation of pairwise alignments and Definition 2.12 can be used to construct a valid transcript from a valid alignment and vice versa. The number of edit operations can also be defined for a pairwise alignment A. It is the sum of errors over all columns:

$$\|A\|_{\mathrm{E}} = \sum_j \delta(a_{1j}, a_{2j}), \quad \text{where} \quad \delta(x,y) = \begin{cases} 0, & \text{if } x = y \\ 1, & \text{else.} \end{cases} \tag{2.16}$$

Hence the edit distance can also be defined using alignments:

$$d_{\mathrm{E}}(s,t) = \min_{A,\text{ an alignment of } s,t} \|A\|_{\mathrm{E}}. \tag{2.17}$$

For two strings of length m and n, with $m < n$, the edit distance and the corresponding transcripts can be computed by the well-known alignment algorithm proposed by Sellers [1980] in $\mathcal{O}(mn)$ time and space. The algorithmic idea goes back to the more general algorithm by Needleman and Wunsch [1970] which computes alignments with maximal similarity in $\mathcal{O}(mn^2)$ time and $\mathcal{O}(mn)$ space allowing for arbitrary gap costs that depend on the gap length. If only the edit distance needs to be determined, the space consumption of Sellers' algorithm can be reduced to $\mathcal{O}(m)$ by only maintaining a single column instead of the whole DP matrix. Myers [1999] proposed an approach which can be adapted to compute the edit distance and is practically faster than both algorithms by exploiting bit-level parallelism to compute the edit distance in $\mathcal{O}\left(\frac{mn}{w} + m \cdot |\Sigma|\right)$ time and $\mathcal{O}\left(\frac{m}{w} \cdot |\Sigma|\right)$

space, where w is the processor word size. The fundamental idea of all edit distance algorithms is the following recurrence which holds for any two strings s and t:

$$d_{\mathrm{E}}\big(\epsilon,\epsilon\big) = 0, \tag{2.18}$$

$$d_{\mathrm{E}}\big(s[0..i],\epsilon\big) = i+1, \tag{2.19}$$

$$d_{\mathrm{E}}\big(\epsilon,t[0..j]\big) = j+1, \tag{2.20}$$

$$d_{\mathrm{E}}\big(s[0..i],t[0..j]\big) = \min\begin{cases} d_{\mathrm{E}}\big(s[0..i-1],t[0..j]\big) & +\ \delta(s[i],\text{-}),\\ d_{\mathrm{E}}\big(s[0..i],t[0..j-1]\big) & +\ \delta(\text{-},t[j]),\\ d_{\mathrm{E}}\big(s[0..i-1],t[0..j-1]\big) & +\ \delta(s[i],t[j]). \end{cases} \tag{2.21}$$

It uses the observation that the set of alignments of $s[0..i]$ and $t[0..j]$ is the union of 3 sets of alignments ending with either $\binom{s[i]}{\text{-}}$, $\binom{\text{-}}{t[j]}$, or $\binom{s[i]}{t[j]}$. If we cut off their last column, all possible alignments between $s[0..i-1]$ and $t[0..j]$, $s[0..i]$ and $t[0..j-1]$, and $s[0..i-1]$ and $t[0..j-1]$ remain. Thus the overall minimal number of errors equals the minimum of minimal number of errors in each of the 3 remaining sets plus errors in its cut column. Dynamic programming (DP) allows to efficiently compute $d_{\mathrm{E}}\big(s[0..i],t[0..j]\big)$ for each pair (i,j) and to store in a traceback matrix which of the 3 values reach the minimum.

Figure 2.3 shows the traceback matrix for two strings s and t. The 3 arrows $\leftarrow$, $\uparrow$, and $\nwarrow$ indicate that optimal alignments of $s[0..i]$ and $t[0..j]$ end with $\binom{s[i]}{\text{-}}$, $\binom{\text{-}}{t[j]}$, or $\binom{s[i]}{t[j]}$. Optimal alignments between s and t correspond to paths along arrows beginning in the bottom-right corner of the traceback matrix. Following a path and reverse concatenating the columns that correspond to each arrow yields an optimal alignment. The example shows that there are two optimal alignments both having 2 insertions, 1 replacement and 1 deletion in their respective transcripts from s to t. They differ only in which of the two A's is deleted from CAAG to CAG.

2.5 Approximate matching

With the definitions above we are now able to define the problem of finding a pattern in a text tolerating a limited number of errors. For the read mapping problem where all possible genomic origins of a read should be determined, we focus on edit and Hamming distance to incorporate sequencing errors and small variations between the reference and the sequenced genome as these well-established distances yield the most effective algorithms. There exist more complex models that consider additional transcript operations like transpositions [Damerau, 1964], the significance of a match [Altschul *et al.*, 1990], or positional error probabilities [Li *et al.*, 2008a]. Many of these models can easily be integrated as a post-filter to edit or Hamming distance based algorithms.

Definition 2.13 (k-error match)**.** Given two strings, a text t and a pattern p, and a fixed number k. A *k-error match* of p is a substring $t' \preceq t$ with $d_{\mathrm{E}}(t',p) \leq k$.

The *k-error problem* is to find all k-error matches of p in t. Analogously to [Gusfield, 1997] we define a *k-mismatch* to be a substring with Hamming distance at most k and the *k-mismatch problem* to find all k-mismatches of p in t. Whilst k-mismatches always have

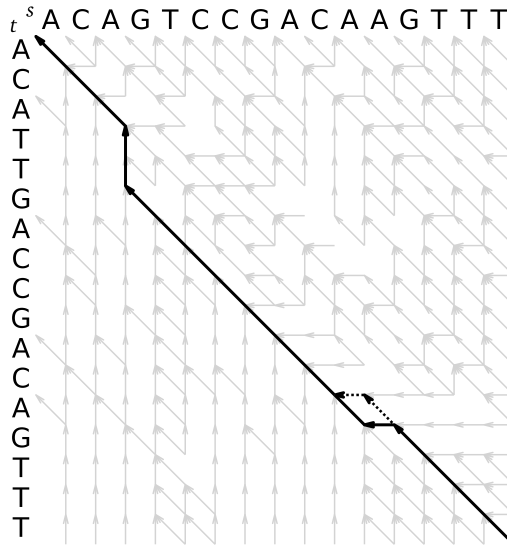


Figure 2.3: *The traceback matrix of an edit distance alignment between s and t from Fig. 2.2. The arrows indicate paths of minimal errors. Optimal global alignment traces are depicted by dark arrows whereas the solid path corresponds to the transcript of Fig. 2.2 and the dotted path is part of an alternative optimal trace.*

length $|p|$, k-error matches can have different lengths between $|p| - k$ and $|p| + k$ even with the same text end position. To reduce the number of k-error matches from $\mathcal{O}(kn)$ to $\mathcal{O}(n)$ we first concentrate on finding all end positions of k-error matches.

To decide whether a k-error match ends at position i and determine its minimal distance for each $i \in [0..n)$ would take $\mathcal{O}(kmn)$ time with a brute force algorithm that computes edit distances. Sellers [1980] proposed an approach that takes only $\mathcal{O}(mn)$ time to find all end positions of approximate matches independent of k. The DP algorithm adapts the edit distance recurrence of the previous section in order to allow a match to start anywhere in the text. The recursive function computes $V\big(p[0..i], t[0..j]\big)$, the minimal edit distance between $p[0..i]$ and all suffixes of $t[0..j]$, and sets $V\big(\epsilon, t[0..j]\big) = 0$ as the empty pattern matches the empty substring at any text position without error:

$$V\big(\epsilon,\epsilon\big) = 0, \tag{2.22}$$
$$V\big(p[0..i],\epsilon\big) = i+1, \tag{2.23}$$
$$V\big(\epsilon,t[0..j]\big) = 0, \tag{2.24}$$
$$V\big(p[0..i],t[0..j]\big) = \min\begin{cases} V\big(p[0..i-1],t[0..j]\big) & + \ \delta(p[i],\text{-}), \\ V\big(p[0..i],t[0..j-1]\big) & + \ \delta(\text{-},t[j]), \\ V\big(p[0..i-1],t[0..j-1]\big) & + \ \delta(p[i],t[j]). \end{cases} \tag{2.25}$$

A k-error match of p ends at position j in t iff $V(p, t[0..j]) \leq k$. Analogously to the edit distance computation the original $\mathcal{O}(mn)$ space consumption can be reduced to $\mathcal{O}(m)$. Ukkonen [1985] reduced the running time from $\mathcal{O}(mn)$ to $\mathcal{O}(kn)$ on average by computing the DP matrix column-wise from the topmost to the *last active cell*. A cell is called

active if it has a value at most k and the value of each active cell solely depends on active cells. Myers' [1999] bit-parallel approach has a running time of $\mathcal{O}\left(\frac{mn}{w} + m \cdot |\Sigma|\right)$ and can be combined with the idea of Ukkonen [1985] yielding an efficient algorithm for approximate pattern matching with $\mathcal{O}\left(\frac{kn}{w} + m \cdot |\Sigma|\right)$ running time.

In Section 6.6.2, we propose a modification of this algorithm that computes V on a diagonal band of the DP matrix yielding the practically fastest banded approximate string matching algorithm.

Part II

INDEX DATA STRUCTURES

Chapter 3 Enhanced Suffix Array

The enhanced suffix array (ESA) was first proposed in [Abouelhoda *et al.*, 2002a] as a memory efficient replacement for suffix trees. In general it consists of 3 tables: the suffix array, the lcp table, and the child table. Each table is a string of the text length n over the alphabet $[0..n)$ and thus requires $\mathcal{O}(n \log n)$ bits of memory without compression. After defining the 3 tables and their relation to suffix trees, we propose construction algorithms and our extensions to multiple sequences and the use of external memory. On different artificial and real-world datasets we analyze the performance of our construction algorithms and demonstrate their applicability to large datasets comprising of multiple mammal genomes (Tables 3.3–3.7). At the end of the chapter, we describe and compare different search algorithms and propose an interface that easily allows to access the enhanced suffix array like a suffix tree and provide 3 application-specific suffix tree iterators.

3.1 Definitions

3.1.1 Suffix array

The suffix array is a space-efficient representation of all non-empty suffixes s_i of a string s in lexicographical order [Manber and Myers, 1993]. As strings of different lengths are different by definition 2.4 on page 14, each suffix s_i of a string s can be uniquely identified by its begin position i and the set of suffixes can be uniquely ordered lexicographically. The sequence of begin positions of suffixes in lexicographical order is called the *suffix array* of s.

Definition 3.1 (suffix array)**.** Given a string s of length n, the suffix array suftab of s is a string of length n over the alphabet $[0..n)$. For every $i, j \in [0..n)$ holds:

$$i < j \quad \Leftrightarrow \quad s_{\mathsf{suftab}[i]} <_{\text{lex}} s_{\mathsf{suftab}[j]} \,. \tag{3.1}$$

For a suffix array suftab the *inverse suffix array* suftab^{-1} is a string over the alphabet $[0..n)$ with $\mathsf{suftab}^{-1}\big[\mathsf{suftab}[i]\big] = i$ for each $i \in [0..n)$.

3.1.2 LCP table

The lcp table stores the lengths of the longest-common prefix between every consecutive pair of suffixes in suftab.

Definition 3.2 (lcp table). Given a string s of length n and its suffix array suftab, the lcp table lcp of s is a string of length $n+1$ over the alphabet $[0..n)$. For every $i \in [1..n)$ holds:

$$\begin{aligned} \mathsf{lcp}[0] &= -1, && (3.2)\\ \mathsf{lcp}[i] &= \left|\mathrm{lcp}\left\{s_{\mathsf{suftab}[i-1]}, s_{\mathsf{suftab}[i]}\right\}\right|, && (3.3)\\ \mathsf{lcp}[n] &= -1. && (3.4) \end{aligned}$$

For the sake of simplicity, we extended the lcp table by two boundary values (-1) which are implicitly needed by some algorithms, e.g. Algorithms 3.9 and 3.10, if the text does not end with a $-character. However, they are not explicitly required in the implementations of these algorithms. We call the table entries *lcp values* and $\mathsf{lcp}[i]$ the lcp value of $s_{\mathsf{suftab}[i]}$ or the i-th lcp value. Manber and Myers [1993] introduced the lcp table and how to construct it as a byproduct of the $\mathcal{O}(n \log n)$ suffix array construction. The first optimal algorithm proposed in [Kasai *et al.*, 2001] constructs the lcp table for a text and a given suffix array in linear-time.

3.1.3 Child table

Besides the linear-time algorithm to construct lcp, Kasai *et al.* proposed a method to traverse the suffix tree $ST(s)$ in a bottom-up fashion by solely scanning lcp from left to right and updating a stack that represents the path from the traversed node to the root. This method is used in [Abouelhoda *et al.*, 2002a] to construct the child table cld, which contains links to the siblings and children of a node and thus represents the structure of the suffix tree. To understand the child table we first need to introduce lcp-intervals.

Definition 3.3 (lcp-interval). An interval $[i..j) \subseteq [0..n)$ with $i+1 < j$ is called an *lcp-interval* of value ℓ or *ℓ-interval* $[i..j)$ if the following holds:

1. $\mathsf{lcp}[i] < \ell$,
2. $\mathsf{lcp}[j] < \ell$,
3. $\forall_{k \in (i..j)}\ \mathsf{lcp}[k] \geq \ell$,
4. $\exists_{k \in (i..j)}\ \mathsf{lcp}[k] = \ell$.

For completeness, we also define $[i..i+1)$ to be a (singleton) *ℓ-interval* with $\ell = |s_{\mathsf{suftab}[i]}|$ and $i \in [0..n)$.

With an ℓ-interval $[i..j)$ we associate the set of suffixes $s_{\mathsf{suftab}[i]}, s_{\mathsf{suftab}[i+1]}, \ldots, s_{\mathsf{suftab}[j-1]}$. Then the following holds: (1) The longest-common prefix ω of these suffixes has length ℓ by properties 3 and 4 and we also call this ℓ-interval an *ω-interval*. (2) A non-singleton ℓ-interval $[i..j)$ is maximal by properties 1 and 2, i.e. every extension to the left or right is no longer an ℓ-interval.

Lemma 3.1 (node-interval-duality). *For every suffix tree node v in $ST(s)$ there is an ω-interval $[i..j)$ and vice versa. If v is an inner node it holds $\omega = concat(v)$ and otherwise $\omega\$ = concat(v)$.*

	0	1	2	3	4	5	6	7	8	9	
$s =$	t	t	a	t	c	t	c	t	t	a	
	0	1	2	3	4	5	6	7	8	9	
suftab $=$	9	2	4	6	8	1	3	5	7	0	
lcp $=$	-1	1	0	2	0	2	1	3	1	3	-1
	a	atctctta	ctctta	ctta	ta	tatctctta	tctctta	tctta	tta	ttatctctta	

***Figure 3.1:** Suffix array and lcp table of* `ttatctctta`*.*

Proof. For each singleton ω-interval holds: ω is a suffix of s and hence there is a leaf v in the suffix tree of s with $concat(v) = \omega\$$. Analogously, for each leaf there is an ω-interval. Now, we consider non-singleton ω-intervals and their corresponding inner nodes v.

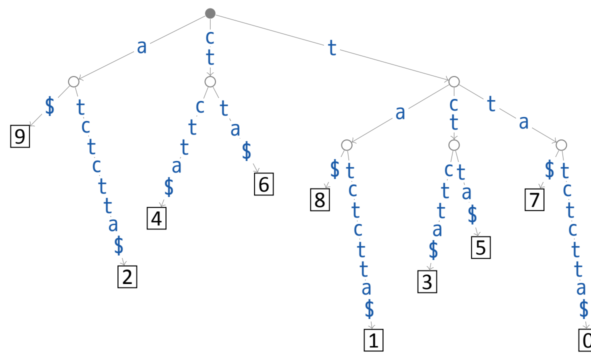
($\rightarrow$) For every inner node v of the suffix tree $ST(s)$ there is a set of suffixes of s beginning with $concat(v)$. Let $S(v)$ denote this set. The suffixes in $S(v)$ correspond to the leaves in the subtree rooted at v. As an inner node, v has at least two outgoing edges beginning with $a, b \in \Sigma \cup \{\$\}$ and $a \neq b$. Therefore there are at least two suffixes in $S(v)$ that begin with $concat(v)\alpha$ and $concat(v)\beta$, where $\alpha, \beta \in \Sigma^0 \cup \Sigma^1$ and $\alpha \neq \beta$. Thus the longest-common prefix of $S(v)$ is $concat(v)$. As all suffixes beginning with a common prefix are stored in a contiguous interval, there are i and j, such that $\mathsf{suftab}[i..j)$ stores the begin positions of all suffixes in $S(v)$. Suffixes that are not included in $S(v)$ are not in the subtree rooted at v and do not begin with $concat(v)$. Therefore $\mathsf{lcp}[i] < \ell$ and $\mathsf{lcp}[j] < \ell$ and $[i..j)$ is an ω-interval with $\omega = concat(v)$.

($\leftarrow$) Let $[i..j)$ be a non-singleton ω-interval and $T(\omega)$ denote the set of suffixes of s beginning with ω. By definition holds $\{s_{\mathsf{suftab}[i]}, \dots, s_{\mathsf{suftab}[j-1]}\} \subseteq T(\omega)$. As the ω-interval is maximal it follows $T(\omega) = \{s_{\mathsf{suftab}[i]}, \dots, s_{\mathsf{suftab}[j-1]}\}$. By property 4 exists a k such that $s_{\mathsf{suftab}[k-1]}$ and $s_{\mathsf{suftab}[k]}$ begin with $\omega\alpha$ and $\omega\beta$, where $\alpha, \beta \in \Sigma^0 \cup \Sigma^1$ and $\alpha \neq \beta$. As a consequence, the lowest common ancestor (lca) of the leaves representing these suffixes is a branching node v with $concat(v) = \omega$ and $S(v) = T(\omega)$. ∎

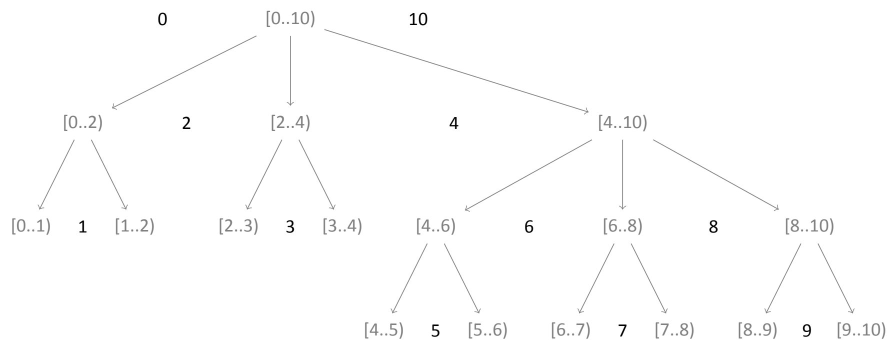
In the following, we consider the suffix tree $ST(s)$ and for each node v denote with $S(v)$ the suffixes represented by the leaves below v.

Corollary 3.1. *Every suffix tree node v can be identified by an lcp-interval $[i..j)$ and both represent the same set of suffixes $S(v) = s_{\mathsf{suftab}[i]}, s_{\mathsf{suftab}[i+1]}, \dots, s_{\mathsf{suftab}[j-1]}$.*

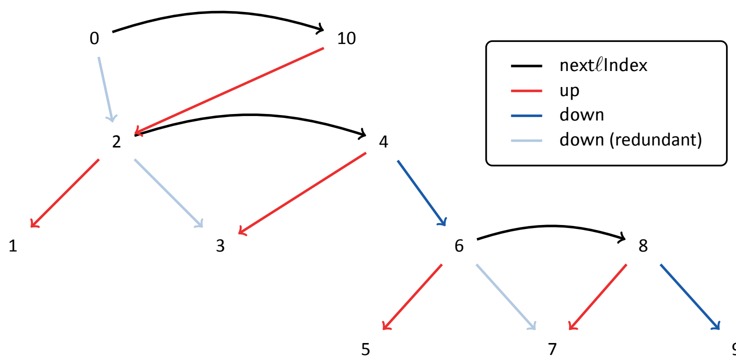
Let v be an inner suffix tree node with children $w_1, \dots, w_m$. W.l.o.g. let $concat(w_1) <_{\text{lex}} concat(w_2) <_{\text{lex}} \dots <_{\text{lex}} concat(w_m)$. Obviously the sets $S(w_1)$, $S(w_2)$, ..., $S(w_m)$ form a partition of the set $S(v)$. As a consequence of Corollary 3.1, the lcp-intervals of the children (child intervals) are subintervals that form a partition $[l_0..l_1)$, $[l_1..l_2)$, ..., $[l_{m-1}..l_m)$



(a) *suffix tree*



(b) *lcp-interval tree*



(c) *linked ℓ-indices*

i	u.	d.	n.	cld
0		2	10	10
1				1
2	1	3	4	4
3				3
4	3	6		6
5				5
6	5	7	8	8
7				7
8	7	9		9
9				2
10	2			

(d) *child table*

Figure 3.2: *Suffix tree (a) and lcp-interval tree (b) of s = `ttatctctta`. The bold numbers between lcp-interval nodes (b) are ℓ-indices of the parent interval above. The child table (d) stores for every ℓ-index i the* up*,* down*, and* nextℓIndex *values (compare with c), which are the first ℓ-indices in the lcp-interval left of i, right of i, or the next ℓ-index in the same lcp-interval, respectively. After the removal of redundant* down *links, the three columns (d) can be stored as a single string* cld *of length n.*

of the ℓ-interval $[i..j)$ of v, where $l_0 = i$ and $l_m = j$. The length of the longest-common prefix of suffixes from different child subtrees is $\ell = |concat(v)|$, whereas the lcp-length of suffixes from the same subtree is greater than ℓ. Thus for $x \in [i, j)$ it holds $\mathsf{lcp}[x] = \ell$ only if $x \in \{l_1, \dots, l_{m-1}\}$ and $\mathsf{lcp}[x] > \ell$, otherwise. The indices $l_1, \dots, l_{m-1}$ uniquely define the partition into subintervals and are called *ℓ-indices* of the lcp-interval $[i..j)$. The set $\{l_1, \dots, l_{m-1}\}$ is denoted by ℓ-indices(i, j).

The parent-child relationship of lcp-intervals corresponds to the parent-child relationship of suffix tree nodes and constitutes the so-called lcp-interval tree [Abouelhoda *et al.*, 2002a], compare Figure 3.2a and Figure 3.2b. The child table is a linked list of ℓ-indices and stores for each ℓ-index so-called up, down, and nextℓIndex values, see Figure 3.2c. It can be represented as 3 subtables which are strings of length $n + 1$ over the alphabet $[0..n]$ (columns u, d, and n in Figure 3.2d).

For $l_k \in \ell$-indices(i, j), nextℓIndex(l_k), if existent, is the next greater ℓ-index l_{k+1} in the set ℓ-indices(i, j). up(l_k) and down(l_k), if existent, are the smallest ℓ-indices in the sets ℓ-indices(l_{k-1}, l_k) and ℓ-indices(l_k, l_{k+1}). For an arbitrary ℓ-index i, the values up, down, and nextℓIndex can formally be defined as follows [Abouelhoda *et al.*, 2002a]:

$$\mathsf{up}(i) = \min\{q \in [0..i) \mid \mathsf{lcp}[q] > \mathsf{lcp}[i] \land \forall_{k \in (q..i)} \mathsf{lcp}[k] \geq \mathsf{lcp}[q]\}, \quad (3.5)$$

$$\mathsf{down}(i) = \max\{q \in (i..n] \mid \mathsf{lcp}[q] > \mathsf{lcp}[i] \land \forall_{k \in (i..q)} \mathsf{lcp}[k] > \mathsf{lcp}[q]\}, \quad (3.6)$$

$$\mathsf{next}\ell\mathsf{Index}(i) = \min\{q \in (i..n] \mid \mathsf{lcp}[q] = \mathsf{lcp}[i] \land \forall_{k \in (i..q)} \mathsf{lcp}[k] > \mathsf{lcp}[i]\}. \quad (3.7)$$

It is easy to see how the child table can be used to enumerate all child intervals for an arbitrary lcp-interval and we will devise an iterator in Section 3.6.2 that can go down and go right in the suffix tree.

3.2 Representation

As explained above, the enhanced suffix array can be represented as strings of length $\mathcal{O}(n)$ over the alphabet $[0..n]$ and thus has a memory consumption of $\mathcal{O}(n \log n)$ bits in total. There exist different approaches, called *succinct indices*, *compressed indices*, or *self-indices*, which have a memory consumption linear to the size of the uncompressed text (succinct), the compressed text (compressed), or even replace the text (self-index) by providing functionality to efficiently reproduce any text substring. Each of the approaches compresses the tables of the enhanced suffix array at the expense of practical access time. We will not go into details and instead refer the interested reader to [Grossi *et al.*, 2003; Sadakane, 2003; Navarro and Mäkinen, 2007].

Abouelhoda *et al.* proposed an easy and elegant way to reduce the memory consumption of the child table by two-thirds. We decide to use their representation as it is coupled with only a small increase in access time. Instead of using 3 strings of length $n + 1$, they merge the 3 subtables into a single string cld of length n over the alphabet $[0..n]$. Their method benefits from the following two observations:

Observation 3.1. *Each defined value* nextℓIndex(i) *can be stored at* cld$[i]$*. For the last ℓ-index i in every lcp-interval* nextℓIndex(i) *is undefined and the entry* cld$[i]$ *can be used*

to store $\mathsf{down}(i)$ *instead. For all other* ℓ*-indices the* down*-value equals the* up*-value of its successor and needs not to be stored explicitly.*

Proof. Trivial. ∎

Observation 3.2. *For every* ℓ*-index* i *that has a defined* up*-value,* $i-1$ *is an* ℓ'*-index with undefined values for* down *and* nextℓIndex*. Thus the* up*-value can be stored at* $\mathsf{cld}[i-1]$*.*

Proof. Let $q := \mathsf{cld}[i].\mathsf{up}$ be the up-value of i. By the formal definition holds $\mathsf{lcp}[q] > \mathsf{lcp}[i]$ and $\mathsf{lcp}[k] \geq \mathsf{lcp}[q] > \mathsf{lcp}[i]$ for every $k \in (q..i)$. It especially holds $\mathsf{lcp}[i-1] > \mathsf{lcp}[i]$ and thus the values $\mathsf{down}(i-1)$ and $\mathsf{next\ell Index}(i-1)$ are undefined as for all $q \geq i+1$ and $k = i+1$ the necessary condition $\mathsf{lcp}[k] > \mathsf{lcp}[q] \geq \mathsf{lcp}[i]$ is violated. ∎

Abouelhoda *et al.* showed how to retrieve the original three values from the merged child table cld:

$$\mathsf{up}[i] = \begin{cases} \mathsf{cld}[i-1], & \text{if } i > 0 \text{ and } \mathsf{lcp}[i-1] > \mathsf{lcp}[i] \\ \perp, & \text{else,} \end{cases} \tag{3.8}$$

$$\mathsf{down}[i] = \begin{cases} \mathsf{cld}[i], & \text{if } \mathsf{lcp}[\mathsf{cld}[i]] > \mathsf{lcp}[i] \\ \perp, & \text{else,} \end{cases} \tag{3.9}$$

$$\mathsf{next\ell Index}[i] = \begin{cases} \mathsf{cld}[i], & \text{if } \mathsf{lcp}[\mathsf{cld}[i]] = \mathsf{lcp}[i] \\ \perp, & \text{else.} \end{cases} \tag{3.10}$$

For each non-singleton lcp-interval $[i..j)$ either $\mathsf{down}[i]$ or $\mathsf{up}[j]$ is defined and equals the first ℓ-index l_1. The other ℓ-indices can be determined by $l_{k+1} = \mathsf{next\ell Index}[l_k]$.

3.3 Construction of the suffix array

Constructing a suffix array differs from ordinary sorting of strings in that suffixes are overlapping substrings of a single text of length n. Generic string sorting algorithms cannot benefit from this information and alone for inspecting all suffix characters need $\Omega(n^2)$ time in the worst case. In the last 20 years plenty of more efficient suffix array construction algorithms were published that avoid redundant character comparisons by reusing the relation of already compared suffixes in subsequent comparisons. Manber and Myers [1993] not only first introduced the concept of suffix arrays but also proposed the first $\mathcal{O}(n \log n)$ construction algorithm. It uses prefix-doubling that in $\mathcal{O}(\log n)$ steps doubles the prefix length the suffixes are sorted by. The algorithm was superseded by many more, practically faster algorithms in the following years summarized in [Puglisi *et al.*, 2007]. By reducing the comparison-based sorting problem [Knuth, 1998] to the problem of sorting suffixes, it can be shown that an $\mathcal{O}(n \log n)$ running time is optimal for arbitrary alphabets. For *integer alphabets*, i.e. subsets of integers from a linear-sized range, the rank of a character can directly be used like in radix sort [Cormen *et al.*, 2001] instead of comparing two characters. Independently from each other, in 2003 three algorithms were proposed that benefit from using character ranks to construct a suffix array in optimal $\Theta(n)$ time [Ko and Aluru, 2005; Kim *et al.*, 2005; Kärkkäinen *et al.*, 2006]. In the following, we will discuss one of them, the *skew algorithm*[1] [Kärkkäinen and Sanders,

[1] The skew algorithm is called DC3 (for Difference Cover modulo 3) in [Kärkkäinen *et al.*, 2006].

2003]. We choose it as the default suffix array construction algorithm in SeqAn as it is fast, robust, and generic. Furthermore, its simplicity enables it to be adapted to obtain an optimal algorithm for external memory [Dementiev *et al.*, 2008a].

At first, we describe the original skew algorithm and afterwards we propose our extension by difference covers, external memory variants and an adaptation to multiple sequences.

3.3.1 The linear-time algorithm by Kärkkäinen et al.

The skew algorithm proposed in [Kärkkäinen and Sanders, 2003; Kärkkäinen *et al.*, 2006] is a recursive algorithm for integer alphabets that consists of the following 3 steps, which we explain in detail below:

1. Construct the suffix array suftab^{12} of suffixes starting at positions $i \not\equiv 0 \pmod 3$. This is done by a recursive call of the skew algorithm for a string of two thirds the length of the text.
2. Construct the suffix array suftab^{0} of the remaining suffixes using the result of the first step.
3. Merge the two suffix arrays into one.

Step 1: Construct the suffix array suftab^{12}

Given a text string s of length n. Let \$ be a character smaller than any character in the text. Consider the triples $s[i..i+2]$ starting at positions $i \not\equiv 0 \pmod 3$ in the text for $0 < i < n$. Append \$\$\$ to s to obtain well-defined triples also for $i \in [n-2..n]$. For $n \equiv 1 \pmod 3$ the appended triple $s[n..n+2]$ is also considered. Determine a lexicographical naming (see Definition 2.7 on page 15) of the triples and assign τ_i the rank of the triple $s[i..i+2]$. This can be done in a linear scan after sorting the triples in linear time with three passes of radix sort (see Figure 3.3).

If the triple names are pairwise distinct, set $\mathsf{suftab}^{12}[\tau_i] := i$ and step 1 is done. Otherwise, recursively construct suftab', the suffix array of the text:

$$\begin{aligned} s' &= \langle \tau_i \mid i \equiv 1 \pmod 3 \rangle \cdot \langle \tau_i \mid i \equiv 2 \pmod 3 \rangle \\ &= \underbrace{\tau_1\, \tau_4\, \dots\, \tau_{1+3(n_0-1)}}_{t_1} \cdot \underbrace{\tau_2\, \tau_5\, \dots\, \tau_{2+3(n_2-1)}}_{t_2}, \quad \text{with } n_j = \left\lceil \frac{n-j}{3} \right\rceil. \end{aligned} \tag{3.11}$$

s' is a string of length $n_0 + n_2 = \lceil \frac{2n-1}{3} \rceil$ over the alphabet $\big[0..|s'|\big)$, where n_j is the number of triples starting at positions $i \equiv j \pmod 3$ that overlap with the first n text characters. There is a one-to-one correspondence between suffixes of s' and the (possibly empty) suffixes s_i with $i \not\equiv 0 \pmod 3$. In case $n \not\equiv 1 \pmod 3$, the last name in t_1 is unique in s' and corresponds to a triple ending with \$ or \$\$. To ensure that a lexicographical comparison between suffixes of s' never exceed the end of t_1, the extra triple \$\$\$ is included into the set of considered triples in case $n \equiv 1 \pmod 3 \Leftrightarrow n_0 - n_1 = 1$. Therefore t_1 consists of

i	$s[i..i+2]$
1	tat
2	atc
4	ctc
5	tct
7	tta
8	ta$
10	$$$

radix pass →

i	$s[i..i+2]$
8	ta$
10	$$$
7	tta
2	atc
4	ctc
1	tat
5	tct

radix pass →

i	$s[i..i+2]$
10	$$$
8	ta$
1	tat
5	tct
7	tta
2	atc
4	ctc

radix pass →

i	$s[i..i+2]$	τ_i
10	$$$	0
2	atc	1
4	ctc	2
8	ta$	3
1	tat	4
5	tct	5
7	tta	6

Figure 3.3: *Skew step 1. Sort the triples of the text $s =$ `ttatctctta`. The three radix passes stably sort the triples by their last, middle, and first character. After that, the lexicographically sorted triples can be named in a linear scan. If non-unique the names are recursively extended to names of suffixes and used to create* suftab^{12}.

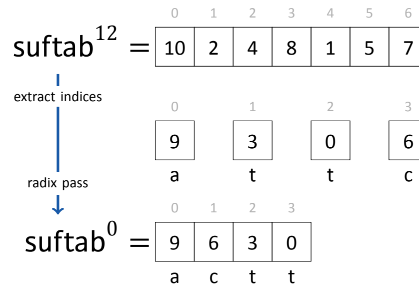


Figure 3.4: *Skew step 2. Suffix start positions $i+1$ with $i+1 \equiv 1 \pmod 3$ are extracted from* suftab^{12} *and stored as i in the same order. A radix pass stably sorts them by the first suffix character $s[i]$ (shown below the boxes) and creates* suftab^{0}.

$n_1 + (n_0 - n_1) = n_0$ triple names and the ranks of suffixes starting in t_1 are not influenced by their t_2 tail. By Lemma 2.1 on page 15 the lexicographical rank of suffix s_i with $i \not\equiv 0 \pmod 3$ equals its corresponding suffix of s'. Thus suftab$'$ can be transformed into suftab^{12} as follows:

$$\mathsf{suftab}^{12}[i] = \begin{cases} 1 + 3\mathsf{suftab}'[i], & \text{if } \mathsf{suftab}'[i] < n_0 \\ 2 + 3(\mathsf{suftab}'[i] - n_0), & \text{else.} \end{cases} \tag{3.12}$$

Step 2: Construct the suffix array suftab^{0}

The remaining suffixes s_i with $i \equiv 0 \pmod 3$ can be sorted by sorting the pairs $(s[i], s_{i+1})$. As the order of the suffixes s_{i+1} is implicitly given in suftab^{12}, suftab^{0} can be constructed by extracting entries $i+1$, with $i+1 \equiv 1 \pmod 3$, from suftab^{12} and writing the entries i in the same order into suftab^{0}. Afterwards stably sort them by $s[i]$ in a single radix pass. These steps are shown in Figure 3.4.

Step 3: Merge suftab^{12} **and** suftab^{0}

Finally, the two sorted suffix arrays need to be merged into the complete suffix array suftab of s. This can be done by scanning them simultaneously and comparing suffixes from suftab^{0} with suffixes from suftab^{12}. If $n \equiv 1 \pmod 3$, the first suffix of suftab^{12} represents the empty suffix and must be skipped. The suffix comparison can be reduced to $\mathcal{O}(1)$ character comparisons and a rank comparison of suffixes from suftab^{12}. To determine the ranks in $\mathcal{O}(1)$ time, we construct R^{12}, the inverse suffix array of suftab^{12}, such that $\mathsf{R}^{12}\big[\mathsf{suftab}^{12}[i]\big] = i$ holds. Two suffixes $i \in \mathsf{suftab}^{0}$ and $j \in \mathsf{suftab}^{12}$ can now be compared in $\mathcal{O}(1)$ time as follows:

$$s_i <_{\text{lex}} s_j \Leftrightarrow \begin{cases} \big(s[i], \mathsf{R}^{12}[i+1]\big) < \big(s[j], \mathsf{R}^{12}[j+1]\big), & \text{if } j \equiv 1 \pmod 3 \\ \big(s[i..i+1], \mathsf{R}^{12}[i+2]\big) < \big(s[j..j+1], \mathsf{R}^{12}[j+2]\big), & \text{if } j \equiv 2 \pmod 3. \end{cases} \quad (3.13)$$

Figure 3.5 shows the merging step and both comparison cases. It is easy to verify that the rank comparison is possible in both cases. Neglecting the recursion, each step of the skew algorithm takes $\mathcal{O}(n)$. The overall running time can be estimated by a geometric series with an asymptotic upper bound in $\mathcal{O}(n)$.

3.3.2 Difference covers

The general idea of the skew algorithm is to partition the set of suffixes into two (or more) subsets such that from every pair of suffixes s_i, s_j a certain prefix of length $\Delta_{i,j} \in \mathcal{O}(1)$ can be cut and the remaining suffixes $s_{i+\Delta_{i,j}}$, $s_{j+\Delta_{i,j}}$ are from one subset. The skew algorithm partitions the suffixes according to their start positions into subsets of residue classes modulo $m = 3$. In Section 5 in [Kärkkäinen *et al.*, 2006] the authors theoretically describe an extension of their algorithm to arbitrary natural numbers m based on difference covers [Haanpää, 2004] of the residue class ring $\mathbb{Z}_m$.

Their proposal considers the set of integers $\mathbb{Z}$ and $\mathbb{Z}_m = \{0, 1, \dots, m-1\}$, the residue class ring modulo m, where each element i represents the residue class $i + m\mathbb{Z}$, the set of integers congruent i modulo m. Under addition $\mathbb{Z}_m$ is a finite cyclic abelian group.

Definition 3.4 (difference cover)**.** Given a finite abelian group G and a subset $D \subseteq G$. D is called a *difference cover* of G, if for every $x \in G$ there exist $d_1, d_2 \in D$ such that $x = d_1 - d_2$.

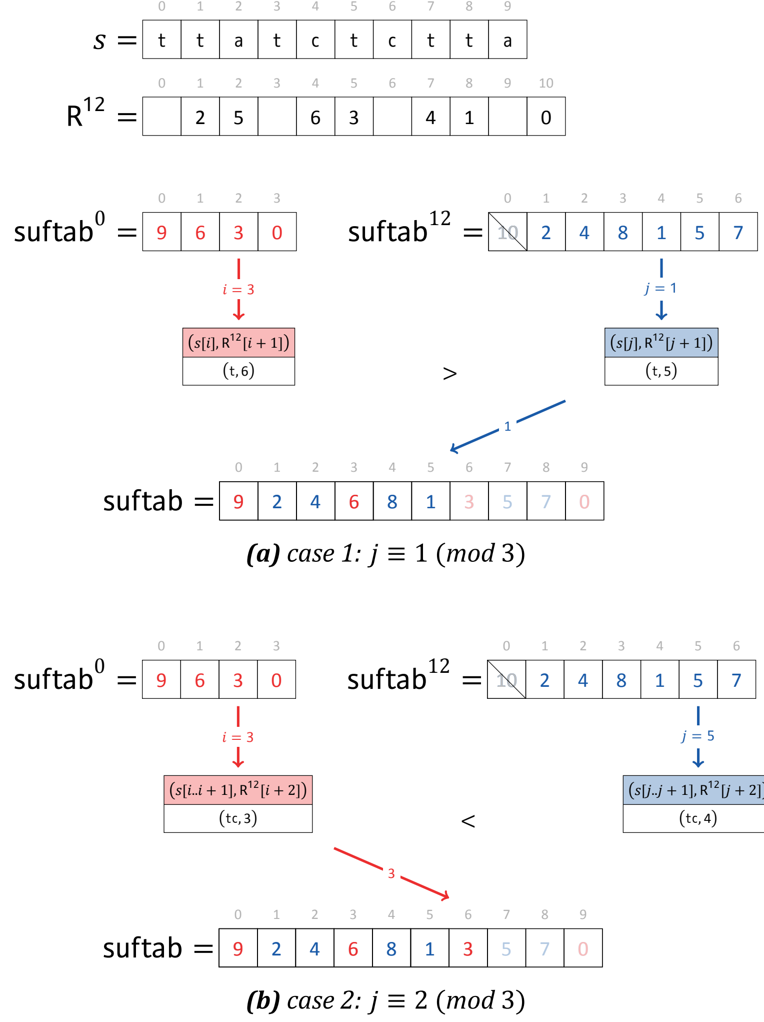
The following lemma shows that the third step of the skew algorithm is always feasible, if in step 1 suffixes are sorted that start at positions in residue classes of a difference cover of $\mathbb{Z}_m$. In that case, arbitrary suffixes can be compared by comparing $\mathcal{O}(m)$ characters and ranks of suffixes from the first step.

Lemma 3.2. *Given a difference cover D of a finite abelian group G. For any $x, y \in G$ there exists a $\Delta \in G$, such that $x + \Delta \in D$ and $y + \Delta \in D$.*

Proof. For any $x, y \in G$ let $z := x - y$. As D is a difference of G and $z \in G$, there exist $a, b \in D$ such that $a - b = z$. For $\Delta := a - x$ holds:

$$(x + \Delta) = x + (a - x) = a \quad \Rightarrow \quad (x + \Delta) \in D, \quad (3.14)$$

$$(y + \Delta) = y + (a - x) = a - (x - y) = a - z = b \quad \Rightarrow \quad (y + \Delta) \in D. \quad (3.15)$$



***(a)** case 1:* $j \equiv 1$ *(mod 3)*

***(b)** case 2:* $j \equiv 2$ *(mod 3)*

Figure 3.5: *Skew step 3. The empty suffix is removed from* suftab^{12} *and the two sorted arrays are merged into the final suffix array. Depending on the residue class of the suffix in* suftab^{12} *either one or two characters and a rank in* R^{12} *are compared.*

■

As a consequence, it is sufficient to compute R^D, such that $\mathsf{R}^D[l]$ is the lexicographical rank of suffix s_l in the set of suffixes starting at positions $\bigcup_{X \in D} X$. We can then determine $\Delta_{i,j} \in [0..m)$ with the above lemma such that for arbitrary suffixes s_i and s_j holds:

$$s_i <_{\text{lex}} s_j \quad \Leftrightarrow \quad \left(s\left[i..i + \Delta_{i,j}\right), \mathsf{R}^D\left[i + \Delta_{i,j}\right]\right) < \left(s\left[j..j + \Delta_{i,j}\right), \mathsf{R}^D\left[j + \Delta_{i,j}\right]\right). \qquad (3.16)$$

As the problem size decreases in every recursion step by a factor of $\lambda = \frac{|D|}{m}$, we are for a given m interested in minimal difference covers and for a fixed size $|D|$ the maximal group $\mathbb{Z}_m$ that can be covered. A difference cover D is called *minimal* for a group if there is no other difference cover of smaller cardinality. By combinatorics the size of a group

G	=	$\mathbb{Z}_3$
D	=	$\{1,2\}$
0	=	$1-1$
1	=	$2-1$
2	=	$1-2$

***(a)** difference cover of* $\mathbb{Z}_3$

$\lvert D\rvert$	G	minimal difference cover of G	λ
2	$\mathbb{Z}_3$	$\{\mathbf{1},\mathbf{2}\}$	0,6666...
3	$\mathbb{Z}_7$	$\{\mathbf{1},\mathbf{2},\mathbf{4}\}$	0,4285...
4	$\mathbb{Z}_{13}$	$\{\mathbf{1},\mathbf{2},\mathbf{4},\mathbf{10}\}$	0,3076...
5	$\mathbb{Z}_{21}$	$\{\mathbf{1},\mathbf{2},\mathbf{7},\mathbf{9},\mathbf{19}\}$	0,2380...
6	$\mathbb{Z}_{31}$	$\{\mathbf{1},\mathbf{2},\mathbf{4},\mathbf{9},\mathbf{13},\mathbf{19}\}$	0,1935...
7	$\mathbb{Z}_{39}$	$\{1,2,17,21,23,28,31\}$	0,1794...
8	$\mathbb{Z}_{57}$	$\{\mathbf{1},\mathbf{2},\mathbf{10},\mathbf{12},\mathbf{15},\mathbf{36},\mathbf{40},\mathbf{52}\}$	0,1403...
9	$\mathbb{Z}_{73}$	$\{\mathbf{1},\mathbf{2},\mathbf{4},\mathbf{8},\mathbf{16},\mathbf{32},\mathbf{37},\mathbf{55},\mathbf{64}\}$	0,1232...
10	$\mathbb{Z}_{91}$	$\{\mathbf{1},\mathbf{2},\mathbf{8},\mathbf{17},\mathbf{28},\mathbf{57},\mathbf{61},\mathbf{69},\mathbf{71},\mathbf{74}\}$	0,1098...
11	$\mathbb{Z}_{95}$	$\{1,2,6,9,19,21,30,32,46,62,68\}$	0,1157...
12	$\mathbb{Z}_{133}$	$\{\mathbf{1},\mathbf{2},\mathbf{33},\mathbf{43},\mathbf{45},\mathbf{49},\mathbf{52},\mathbf{60},\mathbf{73},\mathbf{78},\mathbf{98},\mathbf{112}\}$	0,0902...

***(b)** minimal difference covers*

***Table 3.1:** Difference cover used in the original skew algorithm (a). The right table (b) shows the maximal cyclic groups that can be covered by difference covers from size 2 to 12 [Haanpää, 2004]. The difference covers in bold are perfect.*

G that can be covered by D is limited to:

$$|G| \leq 2 \cdot \binom{|D|}{2} + 1 = |D|^2 - |D| + 1. \tag{3.17}$$

We call a difference cover D *perfect*, if the equality $|G| = |D|^2 - |D| + 1$ holds. Table 3.1b shows the maximal groups for difference covers of sizes 2,...,12 [Haanpää, 2004] and corresponding values of λ. Note that there is no perfect difference cover of size 7 or 11.

3.3.3 Our algorithms

The original skew algorithm uses a perfect difference cover of $\mathbb{Z}_3$ and is hence called DC3 in [Kärkkäinen *et al.*, 2006]. In SeqAn we implemented a generic variant of DC3, that works with arbitrary alphabets and no longer requires the alphabet to contain a smallest \$ character which must only appear as a \$\$\$ triple behind the text. Additionally, we for the first time implemented the skew algorithm for the next greater perfect difference cover $D = \{1,2,4\}$ of the group $\mathbb{Z}_7$. To distinguish our implementations from the original DC3 algorithm, we call them SKEW3 and SKEW7. In the following, we describe both algorithms as well as our external memory and multiple sequence extensions.

SKEW3

In order to provide a generic suffix array construction algorithm in SeqAn, we needed to resolve the constraints that the original algorithm put on the text string and alphabet by introducing a smallest character \$ which is allowed to only occur in the triple \$\$\$ appended to text. Therefore, we first adapted the bucket sort algorithm used for the radix passes to count character accesses behind the end of s in an extra bucket, which becomes the first bucket in the sorted sequence. After the tuples have been sorted, lexicographically adjacent tuples are compared and increasingly named. Tuples that exceed the end of s need not to be compared as they are unique and hence can be assigned to a unique name. Merely in the implementation of the third step when comparing the tuples $s[i..i+\Delta_{i,j})$

and $s[j..j+\Delta_{i,j})$, we must ensure not to exceed the end of s and instead return $s_{\max(i,j)}$ as the lexicographically smaller suffix in that case. To avoid the insertion of an artificial separator between the two halves of s' in SKEW3 or the three thirds of s' in SKEW7, as done in case $n \equiv 1 \pmod 3$ of DC3, we partition the suffixes into residue classes not according to their start position but according to their length. In this way, independent of the length of s the rightmost tuple of each subpart of s' has a unique name. Hence a separator is not necessary as comparisons do not exceed the end of a subpart. Step 3 is still feasible, the only difference is that the comparison of suffixes of length a and b must reduced to a prefix comparison and a rank comparison of suffixes that are *shorter* and have lengths $a-\bar{\Delta}_{a,b}$ and $b-\bar{\Delta}_{a,b}$ in the residue classes of the difference cover. For a perfect difference cover of $\mathbb{Z}_m$ the smallest adequate value of $\bar{\Delta}_{a,b}$ is:

$$\bar{\Delta}_{a,b} := (m - \Delta_{a,b}) \bmod m. \tag{3.18}$$

Algorithm 3.1 shows the SKEW3 pseudo-code as implemented in SeqAn. In step 1 we choose suffixes of a length congruent 1 or 2 modulo 3 and sort their prefix triples in 3 passes of radix sort. The triple names are either used to recursively sort the suffixes or if unique to derive the suffix ranks directly. If constructed recursively, in line 10 the suffix array suftab' of s' is transformed into the suffix array suftab^{12} of suffixes of s with $\mathsf{suftab}^{12}[i] = n - \psi(\mathsf{suftab}'[i])$, where ψ is defined as follows:

$$\psi(i) = \begin{cases} 3(n_2 - i) - 1, & \text{if } i < n_2 \\ 3(n_2 + n_1 - i) - 2, & \text{else.} \end{cases} \tag{3.19}$$

Step 2 is implemented analogously to DC3. The case distinction in step 3 is done according to the length of the suffixes from step 1. A suffix of length congruent 0 modulo 3 is compared with a suffix of length congruent 1 or 2 modulo 3 by a reduction to suffixes which are $\bar{\Delta}_{0,1}$ or $\bar{\Delta}_{0,2}$ characters shorter. Compare line 20 with the values $\bar{\Delta}_{0,1}$ and $\bar{\Delta}_{0,2}$ in Table 3.2a. If one of the suffix sets is completely merged into suftab, the remaining set is appended to suftab in line 24.

SKEW7

SKEW7 is the extension of SKEW3 to the next bigger perfect difference cover. According to Table 3.1b we chose $D = \{1, 2, 4\}$ as a difference cover of $\mathbb{Z}_7$. Step 1 of the pseudo-code in Algorithm 3.2 differs from Algorithm 3.1 only in the choice of suffixes, the length of tuples we sort and s' which is the concatenation of 3 instead of 2 strings of tuple names. Analogously to SKEW3, all of the three strings end with unique 7-tuple names and by Lemma 2.1 the suffix array suftab' of s' reflects the order of suffixes whose length is congruent 1, 2, or 4 modulo 7. It is transformed by $\mathsf{suftab}^{124}[i] = n - \psi(\mathsf{suftab}'[i])$ in line 10 using the following function ψ:

$$\psi(i) = \begin{cases} 7(n_4 - i) - 3, & \text{if } i < n_4 \\ 7(n_4 + n_2 - i) - 5, & \text{if } n_4 \le i < n_4 + n_2 \\ 7(n_4 + n_2 + n_1 - i) - 6, & \text{else.} \end{cases} \tag{3.20}$$

Algorithm 3.1: SKEW3(s)

```
   input   : text string s over the alphabet Σ
   output  : suffix array suftab
                                                                   // Step 1
 1 n ← |s|, n_1 = ⌈n/3⌉, n_2 = ⌈(n−1)/3⌉
 2 A ← ⟨ i | i ∈ [0..n) and (n − i) mod 3 ∈ {1,2} ⟩            // compute triple positions
 3 for j ← 2 downto 0 do                                           // sort triples
 4     sort A stably by s[A[i] + j]
 5 name tuples and let τ_i be the rank of s[n − i..n − i + 2]
 6 if names are unique then
 7     foreach i do suftab^12[τ_i] ← n − i                         // no recursion
 8 else
 9     suftab′ ← SKEW3(τ_{3n_2−1} ... τ_5 τ_2 · τ_{3n_1−2} ... τ_4 τ_1)   // recurse
10     transform suftab′ into suftab^12
                                                                   // Step 2
11 suftab^0 ← ϵ
12 for i ← 0 to |suftab^12| − 1 do             // in-order extract remaining triple position
13     if n − suftab^12[i] ≡ 2 (mod 3) and suftab^12[i] > 0 then
14         suftab^0 ← suftab^0 · (suftab^12[i] − 1)
15 sort suftab^0 stably by s[suftab^0[i]]
                                                                   // Step 3
16 k ← 0, l ← 0, suftab ← ϵ
17 foreach i do R^12[suftab^12[i]] ← i                             // compute ranks
18 while k < |suftab^0| and l < |suftab^12| do                     // merge suffix arrays
19     i ← suftab^0[k], j ← suftab^12[l]
20     if (n − j ≡ 2 (mod 3) and (s[i], R^12[i + 1]) < (s[j], R^12[j + 1])) or          then
          (n − j ≡ 1 (mod 3) and (s[i..i + 1], R^12[i + 2]) < (s[j..j + 1], R^12[j + 2]))
21         suftab ← suftab · i, k ← k + 1                          // s_i is less
22     else
23         suftab ← suftab · j, l ← l + 1                          // s_j is less
24 suftab ← suftab · suftab^0_k · suftab^12_l                      // fill up suftab
25 return suftab
```

a,b	0	1	2	$\max \bar{\Delta}_{a,\cdot}$
0	1	2	1	2
1	2	0	0	2
2	1	0	0	1

(a) $D = \{1,2\}, m = 3$

a,b	0	1	2	3	4	5	6	$\max \bar{\Delta}_{a,\cdot}$
0	3	6	5	6	3	3	5	6
1	6	0	0	6	0	4	4	6
2	5	0	0	1	0	1	5	5
3	6	6	1	1	2	1	2	6
4	3	0	0	2	0	3	2	3
5	3	4	1	1	3	1	4	4
6	5	4	5	2	2	4	2	5

(b) $D = \{1,2,4\}, m = 7$

Table 3.2: *Shift values $\bar{\Delta}_{a,b}$ used in SKEW3 (a) and SKEW7 (b). The rightmost column shows the maximal shift values in each row.*

In step 2 the orders of suffixes of length congruent 3 or 5 modulo 7 are computed in radix passes from suffixes of length congruent 2 or 4 modulo 7 sorted in step 1. The remaining suffixes of length congruent 6 or 0 are sorted in two additional radix passes from suffixes of length congruent 5. The 5 sets of ordered suffixes are merged in step 3 into suftab the totally ordered set of suffixes using a 5-way merge. A priority queue is used to efficiently determine the smallest of 5 or less suffixes. To compare two suffixes of length congruent a and b modulo 7 we compare their prefixes of length $\bar{\Delta}_{a,b}$ and the ranks of suffixes that are $\bar{\Delta}_{a,b}$ characters shorter. Table 3.2b shows all the values of $\bar{\Delta}_{a,b}$ for the used difference cover.

Algorithm 3.2: SKEW7(s)

```
input  : text string s over the alphabet Σ
output : suffix array suftab
                                                                          // Step 1
1   n ← |s|, n_1 = ⌈n/7⌉, n_2 = ⌈(n−1)/7⌉, n_4 = ⌈(n−3)/7⌉
2   A ← ⟨ i | i ∈ [0..n) and (n − i) mod 7 ∈ {1,2,4} ⟩          // compute tuple positions
3   for j ← 6 downto 0 do                                               // sort tuples
4       sort A stably by s[A[i] + j]
5   name tuples and let τ_i be the rank of s[n − i..n − i + 6]
6   if names are unique then
7       foreach i do suftab^124[τ_i] ← n − i                           // no recursion
8   else
9       suftab' ← SKEW7(τ_{7n_4−3} … τ_11 τ_4 · τ_{7n_2−5} … τ_9 τ_2 · τ_{7n_1−6} … τ_8 τ_1)   // recurse
10      transform suftab' into suftab^124
                                                                          // Step 2
11  suftab^0 ← suftab^3 ← suftab^5 ← suftab^6 ← ε
12  for i ← 0 to |suftab^124| − 1 do          // in-order extract remaining tuple position
13      j ← suftab^124[i]
14      if j > 0 then
15          if n − j ≡ 2 (mod 7) then suftab^3 ← suftab^3 · (j − 1)
16          if n − j ≡ 4 (mod 7) then suftab^5 ← suftab^5 · (j − 1)
17  sort suftab^3 stably by s[suftab^3[i]]
18  sort suftab^5 stably by s[suftab^5[i]]
19  foreach i do if suftab^5[i] > 0 then suftab^6 ← suftab^6 · (suftab^5[i] − 1)
20  sort suftab^6 stably by s[suftab^6[i]]
21  foreach i do if suftab^6[i] > 0 then suftab^0 ← suftab^0 · (suftab^6[i] − 1)
22  sort suftab^0 stably by s[suftab^0[i]]
                                                                          // Step 3
23  foreach i do R^124[suftab^124[i]] ← i                              // compute ranks
24  5-way merge suftab^0, suftab^124, suftab^3, suftab^5, suftab^6 into suftab
    // compare two suffixes s_i, s_j by: s_i <_lex s_j ⇔
    // (s[i..i + Δ̄_{n−i,n−j}), R^124[i + Δ̄_{n−i,n−j}]) < (s[j..j + Δ̄_{n−i,n−j}), R^124[j + Δ̄_{n−i,n−j}])
25  return suftab
```

3.3.4 External memory variant

To support the efficient and generic construction of suffix arrays of large texts and whole genomes in SeqAn, we developed variants of SKEW3 and SKEW7 for external memory. The first I/O optimal external memory algorithm for the construction suffix arrays was published in [Dementiev *et al.*, 2005, 2008a] and is a variant of the DC3 algorithm, we denote it as DC3_EXTMEM. It uses the STXXL [Dementiev *et al.*, 2008b] and a pipelining paradigm in which an external memory algorithm is only allowed to sort or sequentially scan the input and intermediate results to produce the output. In this way, the output of one algorithmic component can directly be streamed to succeeding components without intermediate buffering on disk and random I/O accesses occur solely in sorting operations executed by generic I/O efficient STXXL sorting algorithms.

As at the time implementing, the STXXL had no support for Windows platforms and no dedicated algorithm to permute elements in external memory, we decided to reimplement the STXXL pipelining interface in SeqAn as well as a two-pass sorting algorithm and a two-pass permuting algorithm described in [Weese, 2006]. The latter algorithm is important as some I/O intensive sorting operations are in fact permutations with a function π that maps an element x to its position $\pi(x)$ in the output. Our permutation algorithm is not only asymptotically optimal in terms of computing time and I/O accesses but also shows a better practical running time than the external sorting algorithm in [Dementiev and Sanders, 2003]. The algorithm has no random but only bulk read accesses and thus requires no complicated prefetching as necessary in the multiway merge step of external sorting [Dementiev and Sanders, 2003]. To make random write accesses of partially permuted blocks non-blocking, they are written asynchronously with a FIFO.

Analogously to the external memory adaptation of DC3 we transformed the SKEW3 algorithm to comply with the pipeline interface. The pseudo-code is given in [Weese, 2006] and differs from DC3_EXTMEM in that half of the sorting operations are replaced by permutations and suffixes are grouped into congruence classes according to their length, as described above.

The adaptation of SKEW7 for external memory is shown in Algorithm 3.3. At first, lexicographical names for difference cover tuples are determined in lines 1–4. If the tuple names are unique, S' stores pairs of length and rank of difference cover suffixes. Otherwise, analogously to SKEW7, the suffix array of a string s' of tuple name is constructed in a recursive call of SKEW7_EXTMEM and used to determine S' the string of length-rank pairs. In lines 10–14 the difference cover set and the 4 remaining sets of suffixes are equipped with the information required to compare any two suffixes from the same set (step 2) or from different sets (step 3). In order to access suffix ranks while sequentially scanning the text s, we first permute S' in line 15 to store ranks of difference cover suffixes in decreasing length. For a suffix of length a we additionally store its prefix of length $\max \bar{\Delta}_{a,\cdot}$ (compare with Table 3.2b) and the ranks of the 3 next shorter or equal suffixes of lengths in the difference cover. This enables us to compare the suffix with any other suffix of length b by comparing their $\bar{\Delta}_{a,b}$ prefix and ranks of suffixes that are $\bar{\Delta}_{a,b}$ characters shorter, as $\bar{\Delta}_{a,b} \leq \max \bar{\Delta}_{a,\cdot}$ holds and $a - \bar{\Delta}_{a,b}$ is in one of the 3 congruence classes of the difference cover. Before the suffix sets can be merged in step 3, they are lexicographically

sorted in lines 15–19.

SKEW7_EXTMEM is not only asymptotically I/O optimal but also has the least I/O volume (amount of written and read external memory) over all possible difference cover algorithms [Weese, 2006].

Algorithm 3.3: SKEW7_EXTMEM(s)

```
input  : text string s over the alphabet Σ
output : suffix array suftab
                                                                                  // Step 1
1  n ← |s|, n_1 = ⌈n/7⌉, n_2 = ⌈(n−1)/7⌉, n_4 = ⌈(n−3)/7⌉
2  A ← ⟨ (n − i, s[i..i + 6]) | i ∈ [0..n) and (n − i) mod 7 ∈ {1, 2, 4} ⟩
3  sort A by second component
4  S′ ← ⟨ (a, τ_a) | (a, x) ∈ A, where τ_a is lex. name of x ⟩
5  if names are not unique then
6      s′ ← permute S′ such that (a, τ_a) is moved to { n_4 − ⌈a/7⌉,              if a ≡ 4 (mod 7)
                                                     { n_4 + n_2 − ⌈a/7⌉,        if a ≡ 2 (mod 7)
                                                     { n_4 + n_2 + n_1 − ⌈a/7⌉,  else
7      suftab′ ← SKEW7_EXTMEM(⟨τ_a | (a, τ_a) ∈ s′⟩)                             // recurse
8      S′ ← ⟨ (ψ(suftab′[i]), i) | i ∈ [0..|suftab′|) ⟩
                                                                                  // Step 2
9  R^124 ← permute S′ to be descending in the first component                    // compute ranks
                                         // prepare 5-way merge in a linear scan over s and R^124
10 S^124 ← ⟨ (a, s[n − a..n − a + 5], τ_a, τ_{a−4}, τ_{a−6}) | a = n, …, 1 and a ≡ 1 (mod 7) ⟩
         · ⟨ (a, s[n − a..n − a + 4], τ_a, τ_{a−1}, τ_{a−5}) | a = n, …, 1 and a ≡ 2 (mod 7) ⟩
         · ⟨ (a, s[n − a..n − a + 2], τ_a, τ_{a−2}, τ_{a−3}) | a = n, …, 1 and a ≡ 4 (mod 7) ⟩
11 S^3 ← ⟨ (a, s[n − a..n − a + 5], τ_{a−1}, τ_{a−2}, τ_{a−6}) | a = n, …, 1 and a ≡ 3 (mod 7) ⟩
12 S^5 ← ⟨ (a, s[n − a..n − a + 3], τ_{a−1}, τ_{a−3}, τ_{a−4}) | a = n, …, 1 and a ≡ 5 (mod 7) ⟩
13 S^6 ← ⟨ (a, s[n − a..n − a + 4], τ_{a−2}, τ_{a−4}, τ_{a−5}) | a = n, …, 1 and a ≡ 6 (mod 7) ⟩
14 S^0 ← ⟨ (a, s[n − a..n − a + 5], τ_{a−3}, τ_{a−5}, τ_{a−6}) | a = n, …, 1 and a ≡ 0 (mod 7) ⟩
15 permute S^124 such that (a, …, τ_a, …) is moved to τ_a                       // sort suffix sets
16 sort S^3 by (s[n − a], τ_{a−1})
17 sort S^5 by (s[n − a], τ_{a−1})
18 sort S^6 by (s[n − a..n − a + 1], τ_{a−2})
19 sort S^0 by (s[n − a..n − a + 2], τ_{a−3})
                                                                                  // Step 3
20 5-way merge S^0, S^124, S^3, S^5, S^6 into suftab
   // compare two suffixes by: s_{n−a} <_lex s_{n−b} ⇔
   // (s[n − a..n − a + Δ̄_{a,b}), τ_{a−Δ̄_{a,b}}) < (s[n − b..n − b + Δ̄_{a,b}), τ_{b−Δ̄_{a,b}})
21 return suftab
```

3.3.5 Extension to multiple sequences

We now want to extend the notion of the suffix array from one to multiple sequences [Shi, 1996]. Given a set of strings $\mathcal{S} = \{s^1, \dots, s^m\}$ over Σ, we define $\mathcal{S}_{(i,j)} := s^i_j$ to be the suffix of s^i starting at position j. Further, let $\dot{\mathcal{S}} := \{s^1\$^1, \dots, s^m\$^m\}$ where $\$^i$ are sentinels not

contained in any of the strings and $\$^1 < \ldots < \$^m < \min\Sigma$. The sentinels are (conceptually) appended to well-define the order of elements in the suffix array as suffixes from different sequences otherwise might be equal.

Definition 3.5 (generalized suffix array)**.** For a set of multiple strings $\mathcal{S} = \{s^1, \ldots, s^m\}$, the *(generalized) suffix array* is a string of length $n = \sum_{i=1}^{m} |s^i|$ of pairs (i,j) with $i \in [1..m]$ and $j \in [0..|s^i|)$ such that holds:

$$i < j \quad \Leftrightarrow \quad \dot{\mathcal{S}}_{\mathsf{suftab}[i]} <_{\mathrm{lex}} \dot{\mathcal{S}}_{\mathsf{suftab}[j]}\,. \tag{3.21}$$

For $n^{\max}$ being the length of the longest sequence, the generalized suffix array can be constructed in $\mathcal{O}(n \log n^{\max})$ time with the rank-based Manber and Myers [1993] algorithm adapted to multiple sequences [Shi, 1996]. Another approach is to construct the suffix array of $t = s^1\$^1 s^2\$^2 \ldots s^m\$^m$ and transform it to the generalized suffix array. However, the conversion would require $\mathcal{O}(n \log m)$ additional time and in practice many construction algorithms are implemented for $\Sigma = [0..256)$ and prohibit extending the alphabet by m more characters.

We developed an algorithm that directly constructs the generalized suffix array for multiple sequences over arbitrary integer alphabets in external memory. It is a variant of SKEW7_EXTMEM modified in the first recursion level and has the same asymptotic I/O complexity and is thus also optimal. The pseudo-code is shown in Algorithm 3.4. Again, we partition all non-empty suffixes into residue classes modulo 7 according to their length and consider their prefixes of length 7 as tuples. In lines 3–5 difference cover tuples are lexicographically sorted and named. In order to sort suffixes according to Definition 3.5, tuples from s^i and s^j of 6 or less characters are sorted as if a $\$^i$ and $\$^j$ would have been appended to their ends. If the names are not unique, we recursively construct the suffix array of a single string s' which is the concatenation of 3 strings of names $s' = t_4 \cdot t_2 \cdot t_1$, where $t_j := t_j^1 \cdot t_j^2 \cdot \ldots \cdot t_j^m$ contains only names of tuples in residue class j. Each $t_j^i := \tau_{(i,7(n_j^i-1)+j)} \ldots \tau_{(i,7+j)}\tau_{(i,j)}$ is the concatenation of tuple names in residue class j from string s^i, where e.g. $\tau_{(i,a)}$ is the name of $s^i[n^i{-}a..n^i{-}a{+}7)$ and $n^i := |s^i|$. Given n_j^i, the length of t_j^i, the function π determines for tuple name $\tau_{i,a}$ its position $\pi(i,a)$ in s':

$$\pi(i,a) = \begin{cases} \sum_{k=1}^{i} n_4^k - \lceil \frac{a}{7} \rceil, & \text{if } a \equiv 4 \pmod 7 \\ n_4 + \sum_{k=1}^{i} n_2^k - \lceil \frac{a}{7} \rceil, & \text{if } a \equiv 2 \pmod 7 \\ n_4 + n_2 + \sum_{k=1}^{i} n_1^k - \lceil \frac{a}{7} \rceil, & \text{else,} \end{cases} \qquad \text{with } n_j = \sum_{i=1}^{m} n_j^i. \tag{3.22}$$

After the recursion or if names were unique, S' contains pairs of start positions and ranks of suffixes of s'. In lines 10–14, these pairs are assigned to strings R^j which store lexicographical ranks of suffixes of length congruent j modulo 7 of the original strings $s^1, \ldots, s^m$. For the preparation of the 5-way merge of suffixes in step 3, we permute R^j such that ranks are stored in the same order as the corresponding suffixes appear in $s^1 \cdot s^2 \cdot \ldots \cdot s^m$, ascending in the sequence index i and if equal, descending in the length of the s^i-suffix. In this way, the preparation can be carried out by simultaneously scanning $s^1 \cdot s^2 \cdot \ldots \cdot s^m$, R^1, R^2, and R^4. The actual preparation and the 5-way merge in step 3 work analogously to SKEW7_EXTMEM. The only difference is that positions are pairs.

Algorithm 3.4: SKEW7_MULTI($s^1, \ldots, s^m$)

```
input   : multiple text strings s^1, ..., s^m over the alphabet Σ
output  : suffix array suftab
                                                                  // Step 1
1  for i ← 1 to m do
2      n^i ← |s^i|, n^i_1 = ⌈n^i/7⌉, n^i_2 = ⌈(n^i−1)/7⌉, n^i_4 = ⌈(n^i−3)/7⌉
3  A ← ⟨ ((i, n^i − j), s^i[j..j + 6]) | i ∈ [0..m) and j ∈ [0..n^i) and (n^i − j) mod 7 ∈ {1,2,4} ⟩
4  sort A by second component as if $^i was appended to s^i with $^1 < ... < $^m < min Σ
5  S' ← ⟨ (π(i,a), τ_(i,a)) | ((i,a), x) ∈ A, where τ_(i,a) is lex. name of x ⟩
6  if names are not unique then
7      s' ← permute S' such that (p, τ) is moved to p
8      suftab' ← SKEW7_EXTMEM(⟨τ | (a, τ) ∈ s'⟩)                  // recurse
9      S' ← ⟨ (suftab'[i], i) | i ∈ [0..|suftab'|) ⟩
                                                                  // Step 2
10 b ← 0
11 for j ∈ {1,2,4} do
12     a ← b, b ← a + Σ_{i=1}^{m} n^i_j
13     B ← ⟨ (p, τ) | (p, τ) ∈ S' and a ≤ p < b ⟩       // extract suffixes of length ≡ j
14     R^j ← permute B to be ascending in the first component     // compute ranks
15 S^124 ←       // prepare 5-way merge in a linear scan over s^1 · ... · s^m, R^1, R^2, and R^4
     ∏_{i=1}^{m} (   ⟨ ((i,a), s^i[n^i−a..n^i−a+5], τ_{i,a}, τ_{i,a−4}, τ_{i,a−6}) | a = n^i, ..., 1 ∧ a ≡ 1 (mod 7) ⟩
                   · ⟨ ((i,a), s^i[n^i−a..n^i−a+4], τ_{i,a}, τ_{i,a−1}, τ_{i,a−5}) | a = n^i, ..., 1 ∧ a ≡ 2 (mod 7) ⟩
                   · ⟨ ((i,a), s^i[n^i−a..n^i−a+2], τ_{i,a}, τ_{i,a−2}, τ_{i,a−3}) | a = n^i, ..., 1 ∧ a ≡ 4 (mod 7) ⟩ )
16 S^5 ← ∏_{i=1}^{m} ⟨ ((i,a), s^i[n^i−a..n^i−a+3], τ_{i,a−1}, τ_{i,a−3}, τ_{i,a−4}) | a = n^i, ..., 1 ∧ a ≡ 5 (mod 7) ⟩
17 S^6 ← ∏_{i=1}^{m} ⟨ ((i,a), s^i[n^i−a..n^i−a+4], τ_{i,a−2}, τ_{i,a−4}, τ_{i,a−5}) | a = n^i, ..., 1 ∧ a ≡ 6 (mod 7) ⟩
18 S^0 ← ∏_{i=1}^{m} ⟨ ((i,a), s^i[n^i−a..n^i−a+5], τ_{i,a−3}, τ_{i,a−5}, τ_{i,a−6}) | a = n^i, ..., 1 ∧ a ≡ 0 (mod 7) ⟩
19 S^3 ← ∏_{i=1}^{m} ⟨ ((i,a), s^i[n^i−a..n^i−a+5], τ_{i,a−1}, τ_{i,a−2}, τ_{i,a−6}) | a = n^i, ..., 1 ∧ a ≡ 3 (mod 7) ⟩
20 permute S^124 such that ((i,a), ..., τ_{i,a}, ...) is moved to τ_{i,a}     // sort suffix sets
21 sort S^3 by (s^i[n^i − a], τ_{i,a−1})
22 sort S^5 by (s^i[n^i − a], τ_{i,a−1})
23 sort S^6 by (s^i[n^i − a..n^i − a + 1], τ_{i,a−2})
24 sort S^0 by (s^i[n^i − a..n^i − a + 2], τ_{i,a−3})
                                                                  // Step 3
25 5-way merge S^0, S^124, S^3, S^5, S^6 into suftab
   // compare two suffixes by: Ṡ_(i,n^i−a) <_lex Ṡ_(j,n^j−b) ⇔
   // (s^i[n^i − a..n^i − a + Δ̄_{a,b}), τ_{i,a−Δ̄_{a,b}}) < (s^j[n^j − b..n^j − b + Δ̄_{a,b}), τ_{j,b−Δ̄_{a,b}})
26 return suftab
```

3.4 Construction of the lcp table

The first suffix array construction algorithm [Manber and Myers, 1993] as well as the skew algorithm [Kärkkäinen *et al.*, 2006] can be extended to construct the lcp table as a byproduct with auxiliary data structures. The first optimal approach was a standalone linear-time algorithm published by Kasai *et al.* [2001].

3.4.1 The linear-time algorithm by Kasai et al.

The basic idea of the lcp table construction algorithm proposed in [Kasai *et al.*, 2001] is to use the lcp length of a suffix and its lexicographical predecessor for the comparison of the next shorter suffix and its predecessor. The linear running time is possible due to the following lemma.

Lemma 3.3. *Given a string s of length n, the corresponding suffix array, and the lcp table. For every $j \in [0..n-1)$ with $\mathsf{lcp}\left[\mathsf{suftab}^{-1}[j]\right] = l$ and $\mathsf{suftab}^{-1}[j+1] \neq 0$ holds $\mathsf{lcp}\left[\mathsf{suftab}^{-1}[j+1]\right] \geq l-1$.*

Proof. Let s_j be a suffix with $\mathsf{lcp}\left[\mathsf{suftab}^{-1}[j]\right] = l$. The assumption obviously holds for $l \leq 0$. For $l > 0$, s_j has a lexicographical predecessor, say s_i, and for the next shorter suffixes s_{i+1} and s_{j+1} holds $\left|\mathsf{lcp}\{s_{i+1}, s_{j+1}\}\right| = l-1$ and $s_{i+1} <_{\text{lex}} s_{j+1}$. The lexicographical predecessor of s_{j+1} is $s_{\mathsf{suftab}[\mathsf{suftab}^{-1}[j+1]-1]}$ and it holds $s_{i+1} \leq_{\text{lex}} s_{\mathsf{suftab}[\mathsf{suftab}^{-1}[j+1]-1]} <_{\text{lex}} s_{j+1}$. From the latter follows:

$$\mathsf{lcp}\left[\mathsf{suftab}^{-1}[j+1]\right] = \left|\mathsf{lcp}\left\{s_{\mathsf{suftab}[\mathsf{suftab}^{-1}[j+1]-1]}, s_{j+1}\right\}\right| \geq \left|\mathsf{lcp}\left\{s_{i+1}, s_{j+1}\right\}\right| = l-1. \quad (3.23)$$

■

As a consequence, the lcp values of suffixes s_j can be computed for increasing j beginning with $j = 0$ and the pairwise suffix comparison can skip the common prefix of at least $\max(l-1, 0)$ characters, where l is the lcp value of the previous comparison. In this way, the overall number of character comparisons is less than $2n$ and the inner loop in line 9 of Algorithm 3.5 takes $\mathcal{O}(n)$ overall time as well as the whole algorithm.

Algorithm 3.5: CONSTRUCTLCPTABLE(s, suftab)

input : text string s, suffix array suftab
output : lcp table lcp

```
1   n ← |s|, lcp[0] ← −1, lcp[n] ← −1
2   for i ← 0 to n − 1 do                                   // invert suffix array
3       suftab⁻¹[suftab[i]] ← i
4   l ← 0
5   for j ← 0 to n − 1 do
6       if suftab⁻¹[j] ≠ 0 then
7           i ← suftab[suftab⁻¹[j] − 1]          // determine the lex. predecessor s_i of s_j
8           while min(i, j) + l < n and s[i + l] = s[j + l] do      // compute |lcp{s_i, s_j}|
9               l ← l + 1
10          lcp[suftab⁻¹[j]] ← l
11      if l > 0 then l ← l − 1              // skip max(l − 1, 0) prefix in the next round
12  return lcp
```

3.4.2 Space-saving variant

Manzini [2004] found a way to save the $4n$ bytes[2] of additional memory consumed by the inverse suffix array by reusing the memory of lcp. Before the lcp values are written, lcp stores for each suffix rank k the rank of the next shorter suffix, i.e. $\mathsf{RankNext}[k] = \mathsf{suftab}^{-1}[\mathsf{suftab}[k]+1]$. After substitution of j by $\mathsf{suftab}[k]$ and $\mathsf{suftab}^{-1}[j]$ by k in Algorithm 3.5, Manzini replaces lines 2–3 by a RankNext construction algorithm which utilizes the rank-preservation property of the Burrows-Wheeler transform [Burrows and Wheeler, 1994].

Algorithm 3.6: CONSTRUCTLCPTABLE_INPLACE(s, suftab)

```
   input   : text string s, suffix array suftab
   output  : lcp table lcp
1  n ← |s|
2  for i ← 0 to n − 1 do                                  // invert suffix array
3      lcp[suftab[i]] ← i
4  l ← 0
5  for j ← 0 to n − 1 do
6      if lcp[j] ≠ 0 then
7          i ← suftab[lcp[j] − 1]              // determine the lex. predecessor s_i of s_j
8          while min(i, j) + l < n and s[i + l] = s[j + l] do      // compute |lcp{s_i, s_j}|
9              l ← l + 1
10     lcp[j] ← −(l + 1)
11     if l > 0 then l ← l − 1              // skip max(l − 1, 0) prefix in the next round
12 for j ← 0 to n − 1 do          // transform lcp values from text to suffix array order
13     if lcp[j] < 0 then                          // find a cycle that needs to be permuted
14         i ← j, t_tmp ← lcp[j]
15         while suftab[i] ≠ j do          // for k = suftab[j], suftab[suftab[j]], ..., j
16             lcp[i] ← −lcp[suftab[i]] − 1          // move lcp[suftab^-1[k]] ← lcp[k]
17             i ← suftab[i]
18         lcp[i] ← −t_tmp − 1
19 lcp[0] ← −1, lcp[n] ← −1
20 return lcp
```

Independent from Manzini's approach, we found another simple way to reuse the memory of lcp by storing suftab^{-1} in it [Weese, 2006]. As the values of suftab^{-1} are read only once and in sequential order, each entry can be used after reading to store the computed lcp value. However, after all lcp values have been computed they are in text order and must be permuted in-place to be in suffix array order, i.e. an lcp value at position j must be moved to position $\mathsf{suftab}^{-1}[j]$. To permute elements without overwriting others, we swap them along cycles $j, \mathsf{suftab}[j], \mathsf{suftab}[\mathsf{suftab}[j]], \ldots, j$. To permute all cycles exactly once we iterate over all cycle start positions j and mark non-permuted elements with negative values. The algorithmic details are shown in Algorithm 3.6. A similar algorithm that constructs a sparse lcp table was later published in [Kärkkäinen *et al.*, 2009].

[2] We assume that $n \leq 2^{32}$ holds, otherwise suftab^{-1} consumes $8n$ bytes.

3.4.3 Adaptation to external memory

Adapting the algorithm in [Kasai *et al.*, 2001] to efficiently use external memory is challenging, as it shows a poor locality behavior. Although in the main loop all accesses to suftab^{-1} and text accesses via $s[j+l]$ are in sequential order, accesses to suftab, lcp, and $s[i+l]$ are random. Our in-place algorithm, described in the previous section, suggests how lcp can be permuted such that accesses to it become sequential. A similar permutation is possible for suftab as it is clear beforehand in which pattern suftab values will be accessed. For text accesses via $s[i+l]$ this does not hold and yet all approaches to an external memory lcp construction [Kasai *et al.*, 2001; Kärkkäinen *et al.*, 2009; Gog and Ohlebusch, 2011] are semi-external, i.e. they require the whole text [Gog and Ohlebusch, 2011] and an additional array of n [Gog and Ohlebusch, 2011] or $4n$ byte [Kasai *et al.*, 2001; Kärkkäinen *et al.*, 2009] to reside in main memory.

We developed a window based approach that is applicable even if the text does not fit into main memory. It processes consecutive non-overlapping text windows of an arbitrary size w in $\lceil \frac{n}{w} \rceil$ rounds. If $s[a..b)$ is the current window, then character comparisons between $s[i+l]$ and $s[j+l]$ can only be conducted if $i+l \in [a..b)$. However, some suffix comparisons may exceed the window border. Those comparisons must be interrupted at the end of the current window and resumed in the next window. The following lemma will help to easily keep track of suffixes s_i whose comparisons were interrupted. Whereas Lemma 3.3 states a relation between lcp lengths of suffixes and their lexicographical successors, the next lemma is its counterpart and gives a relation of suffixes and their successors.

Lemma 3.4. *Given a string s of length n and the corresponding suffix array and lcp table. For every $j \in [0..n-1)$ with $\mathsf{lcp}\left[\mathsf{suftab}^{-1}[j]+1\right] = l$ and $\mathsf{suftab}^{-1}[j+1] \neq n-1$ holds $\mathsf{lcp}\left[\mathsf{suftab}^{-1}[j+1]+1\right] \geq l-1$.*

Proof. This lemma can be proven analogously to Lemma 3.3. ∎

A direct consequence of Lemma 3.4 is that if an lcp comparison of a suffix s_i with its lexicographical successor exceeds the window end b, comparisons of all shorter suffixes $s_{i'}$ with $i < i' < b$ will leave the window as well. Let $\omega(b)$ be defined as the leftmost start position of such suffixes:

$$\omega(b) = \min\left\{ i \,\middle|\, i \in [0..n) \wedge i \leq b \leq i + \mathsf{lcp}\left[\mathsf{suftab}^{-1}[i]+1\right] \right\}. \tag{3.24}$$

Clearly, comparisons of suffixes s_i will end left of b if $i < \omega(b)$ and exceed b if $\omega(b) \leq i < b$. This allows to stop comparisons of suffixes s_i at the window end b and to determine $\omega(b)$, the smallest of such i. Comparisons of suffixes s_i with $\omega(a) \leq i < a$ were interrupted at end of the previous window and can be resumed by setting l to at least $a-i$.

Algorithm 3.7 shows the pseudo-code of our implementation. Lines 1–3 prepare values $\mathsf{suftab}^{-1}[j]$ and $\mathsf{suftab}\left[\mathsf{suftab}^{-1}[j]-1\right]$ for increasing j. The main loop from line 4 to 16 iterates over all non-overlapping windows $s[a..b)$, where ω_a equals $\omega(a)$ and ω_b is used to compute $\omega(b)$. An lcp value is successfully computed if the suffix comparison

was not interrupted or ends at the text end. Then the lcp value lcp[k] and its rank k is appended to L. At the end, L is permuted to be ascending in k and filtered for values lcp[k] in lines 17–18.

Algorithm 3.7: CONSTRUCTLCPTABLE_EXTMEM(s, suftab)

```
   input   : text string s, suffix array suftab
   output  : lcp table lcp
1  n ← |s|, ω_a ← 0, L ← (0,−1)·(n,−1)          // L is a string of pairs (rank,lcp value)
2  A ← ⟨ (k, suftab[k−1], suftab[k]) | k ∈ [0..n) ⟩
3  permute A such that (k,i,j) is moved to j                        // prepare values
4  for j ← 1 to ⌈n/w⌉ do
5      a ← (j−1)·w, b ← min(jw,n), ω_b ← b                   // process window s[a..b)
6      foreach (k,i,j) ∈ A do
7          if k>0 then
8              if ω_a ≤ i and i+l < b then
9                  l ← max(l, a−i)                      // resume interrupted comparison
10                 while (i+l) ∈ [a..b) and j+l < n and s[i+l] = s[j+l] do
11                     l ← l+1
12                 if i+l < b or b = n then       // if comparison was not interrupted
13                     L ← L·(k,l)                      // append (rank,lcp value) to L
14         if i+l ≥ b then ω_b ← min(ω_b, i)
15         if l > 0 then l ← l−1           // skip max(l−1,0) prefix in the next round
16     ω_a ← ω_b
17 permute L such that (i,l) is moved to i               // order lcp values by their rank
18 lcp ← ⟨ l | (i,l) ∈ P ⟩
19 return lcp
```

We implemented the algorithm using the pipelining interface. The current window $s[a..b)$ is loaded into a memory buffer of size w. To minimize the running time, w should be chosen as large as possible.

3.4.4 Extension to multiple sequences

All of the lcp table construction algorithms described above can easily be adapted to multiple sequences. For a given a set $\mathcal{S} = \{s^1, \dots, s^m\}$ of strings of lengths $n^1, \dots, n^m$ and the corresponding generalized suffix array suftab, we define:

$$\phi(i,j) = j + \sum_{k=1}^{i-1} n^k \quad \text{and} \quad n = \sum_{k=1}^{m} n^k. \tag{3.25}$$

As the generalized suffix array stores pairs instead of single integers, its entries cannot directly be used to access suftab^{-1}. Therefore, we adapt Algorithm 3.5 and use ϕ as a unique mapping of suffix start positions onto the interval $[0..n)$ in lines 5,9,10, and 13 in Algorithm 3.8. The second adaptation concerns the lcp comparison in line 11.

Algorithm 3.8: CONSTRUCTLCPTABLE_MULTI($s^1, \dots, s^m$, suftab)

```
input   : multiple text strings s^1, ..., s^m, suffix array suftab
output  : lcp table lcp
1  n ← 0
2  for i ← 1 to m do
3      n^i ← |s^i|, n ← n + n^i
4  for i ← 0 to n − 1 do                                        // invert suffix array
5      suftab^{-1}[φ(suftab[i])] ← i
6  l ← 0, lcp[0] ← −1, lcp[n] ← −1
7  for i ← 1 to m do
8      for j ← 0 to n^i − 1 do
9          if suftab^{-1}[φ(i,j)] ≠ 0 then
10             (a,b) ← suftab[suftab^{-1}[φ(i,j)] − 1]          // determine lex. predecessor
11             while b + l < n^a and j + l < n^i and s^a[b + l] = s^i[j + l] do
12                 l ← l + 1                                    // compute lcp value
13             lcp[suftab^{-1}[φ(i,j)]] ← l
14         if l > 0 then l ← l − 1
15 return lcp
```

By implementing ϕ using an array of length m that stores at position i the partial sum of the first $i-1$ sequence lengths, $\phi(i,j)$ can be determined in constant time and Algorithm 3.8 constructs the lcp table in $\mathcal{O}(n)$ time. Algorithms 3.6 and 3.7 can analogously be adapted without changing their asymptotical running time.

3.5 Construction of the child table

3.5.1 Bottom-up suffix tree traversal

As mentioned in Section 3.1.3, the lcp table alone can be used to traverse the inner nodes of the suffix tree in a bottom-up fashion with an algorithm proposed by Kasai *et al.* [2001]. The corresponding pseudo-code is given in Algorithm 3.9. In linear time the algorithm outputs all ℓ-intervals $[lb..rb)$ that correspond to suffix tree nodes visited in a postorder depth-first search (DFS). Therefore it scans the lcp-table and maintains a stack of growing lcp-table intervals and their minimal lcp value ℓ. Iteratively every lcp value lcp$[i]$ closes intervals with greater lcp values (lines 5–9), extends or if not part of the stack creates a new lcp-interval spanning the closed intervals (lines 9–11). For the proof of correctness we refer the reader to [Kasai *et al.*, 2001].

3.5.2 The linear-time algorithm by Abouelhoda et al.

In [Abouelhoda *et al.*, 2002b] the authors propose two modifications of this bottom-up algorithm to construct the up and down values and the nextℓIndex values. In Algorithm 3.10 we show the combination of both algorithms to directly construct the child

Algorithm 3.9: BOTTOMUPTRAVERSAL(lcp)

```
input  : lcp table lcp
output : lcp-intervals traversed in a postorder dfs
1  n ← |lcp| − 1                                              // n is the text length
2  S ← (−1, 0, 0)         // initialize stack with super-root node (ℓ, lb, rb) = (−1, 0, 0)
3  for i ← 1 to n do
4      lb ← i − 1
5      while lcp[i] < top(S).ℓ do              // close intervals with greater lcp values
6          top(S).rb ← i
7          interval ← pop(S)
8          report(interval)                                    // report closed interval
9          lb ← interval.lb                                  // get the leftmost boundary
10     if lcp[i] > top(S).ℓ then
11         push((lcp[i], lb, i), S)       // create new interval spanning closed intervals
```

table. Instead of ℓ-values and interval boundaries used in Algorithm 3.9, the stack in Algorithm 3.10 only stores ℓ-indices, i.e. each ℓ-interval is represented by a run of its ℓ-indices. According to the space-saving trick described in Section 3.2, up values for ℓ-indices i are stored at position $i-1$ (line 9) and down values are stored only for the last of all ℓ-indices of each interval (lines 6–7).

down **values**

The condition in line 6 is true, iff the two ℓ-values on the top of the stack are different from each other and greater than the current ℓ-value lcp$[i]$. In this case, both elements will be removed and are ℓ-indices from an lcp-interval and its last-child interval. As ℓ-indices of the same interval are stored as an ascending run, the topmost stack entry ($last$) is the first ℓ-index in the last-child interval and the second topmost stack entry (now top(S)) is the last ℓ-index in the parent interval and the left border of the last-child interval. Hence, $last$ is its down value and needs to be stored.

up **values**

In line 8, $last$ is the last ℓ-index removed by the current ℓ-index i or equals -1 if none was removed. In the first case, the last removed ℓ index is the first ℓ-index in the child interval left of i. Hence, $last$ is the up value of i and is stored at position $i-1$.

nextℓIndex **values**

nextℓIndex values are computed similarly. If after the removal of all greater ℓ-values the topmost ℓ-value equals the current one, the topmost ℓ-index is directly preceding i and its nextℓIndex value is set accordingly in line 12.

For the proof of correctness and a more detailed description we refer the reader to [Abouelhoda *et al.*, 2002b]. Although this algorithm as well as Algorithm 3.9 reads the

Algorithm 3.10: ConstructChildTable(lcp)

```
input  : lcp table lcp
output : child table cld
1  n ← |lcp| − 1                                        // n is the text length
2  S ← 0, last ← −1                      // initialize stack with the first ℓ-index
3  for i ← 1 to n do
4      while lcp[i] < lcp[top(S)] do
5          last ← pop(S)
6          if lcp[i] < lcp[top(S)] and lcp[top(S)] ≠ lcp[last] then
7              cld[top(S)] ← last      // top(S) is last ℓ-index → store down value
8      if last ≠ −1 then
9          cld[i − 1] ← last                                    // store up value
10         last ← −1
11     if lcp[i] = lcp[top(S)] then        // is the previous ℓ-index on the stack?
12         cld[top(S)] ← i                         // set its nextℓIndex value to i
13     push(i, S)
14 return cld
```

algorithm	reference	complexity
suffix array		
BWF	BwtWalkFast [Baron and Bresler, 2005; Marschall *et al.*, 2009]	$\mathcal{O}(n \log n)$
BWIP	in-place variant of BwtWalkFast [Marschall *et al.*, 2009]	$\mathcal{O}(n^2)$
DS	deep-shallow sort [Manzini and Ferragina, 2004]	$\mathcal{O}(n \log n)$
MM	prefix doubling algorithm [Manber and Myers, 1993]	$\mathcal{O}(n \log n)$
QSORT	quick sort with lexicographical string comparisons	$\mathcal{O}(n^3)$
SKEW3	our variant of DC3 [Kärkkäinen *et al.*, 2006], see Section 3.3.3	$\mathcal{O}(n)$
SKEW7	SKEW3 extension to the next larger, perfect difference cover	$\mathcal{O}(n)$
lcp table		
KASAI	linear-time lcp construction algorithm [Kasai *et al.*, 2001]	$\mathcal{O}(n)$
KASAIIP	our in-place variant of the Kasai's algorithm, see Section 3.4.2	$\mathcal{O}(n)$
child table		
CHILDTAB	bottom-up construction [Abouelhoda *et al.*, 2002a]	$\mathcal{O}(n)$

Table 3.3: *Enhanced suffix array construction algorithms available in SeqAn.*

lcp table up to position n, it can easily be verified that it leaves cld$[n]$ untouched. Thus, the child table can be stored as a string of length n.

3.5.3 Adaptation to external memory and multiple sequences

Algorithm 3.10 can easily be adapted to external memory as it sequentially reads the lcp table and accesses adjacent elements on a stack. The only random accesses are the write accesses cld$[x] \leftarrow y$. Instead of directly executing them, our implementation collects a sequence of pairs (x, y) which at the end of the algorithm is externally sorted by x and used to sequentially fill cld.

corpus/dataset	reference
Gauntlet corpus	http://compressionratings.com/files/gauntlet_corpus.zip
Schürmann-Stoye corpus	http://bibiserv.techfak.uni-bielefeld.de/download/tools/bpr.html
Manzini-Ferragina corpus	http://people.unipmn.it/manzini/lightweight/corpus/
UCSC genomes	http://hgdownload.cse.ucsc.edu/downloads.html
celegans	concatenated UCSC chromosomes of C. elegans
dmel	concatenated UCSC chromosomes of D. melanogaster
hs	concatenated UCSC human chromosomes
mammals	three UCSC whole genome sequences (human, dog, and mouse)

Table 3.4: *Datasets used for ESA experiments.*

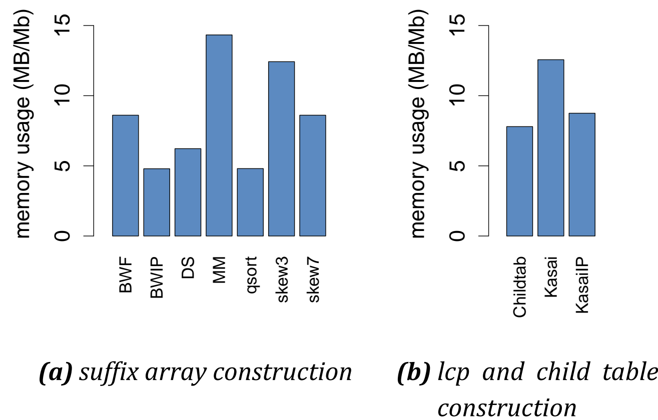
		running time [s/Mb]									
dataset	**size** [Mb]	BWF	BWIP	DS	MM	QSORT	SKEW3	SKEW7	KASAI	KASAIIP	CHILDTAB
abac	0.2	0.0	53.3	121.3	0.3	2417.9	0.1	0.1	0.0	0.0	0.00
paper5x80	1.0	0.0	13.6	0.7	1.1	1587.0	0.2	0.2	0.0	0.1	0.01
test1	2.1	0.1	1726.0	3.7	2.4	–	0.4	0.3	0.0	0.1	0.01
test2	2.1	0.1	1908.5	3.7	2.4	–	0.4	0.3	0.0	0.1	0.01
test3	2.1	0.0	64.9	0.7	2.3	712.2	0.3	0.3	0.0	0.2	0.01
world	2.5	1.4	6.8	0.1	4.2	0.5	0.5	0.4	0.0	0.2	0.02
houston	3.8	0.1	11.9	27.1	1.2	1046.7	0.2	0.3	0.0	0.2	0.02
bible	4.0	1.0	7.2	0.1	5.2	0.5	0.8	0.6	0.1	0.2	0.02
abba	10.5	0.1	0.6	2.5	4.5	–	0.7	0.7	0.0	0.2	0.01
book1x20	15.4	0.1	80.0	5.6	5.2	–	1.0	1.3	0.1	0.2	0.01
Fibonacci	20.0	0.1	0.4	12.5	3.5	–	0.7	0.7	0.1	0.2	0.01
period_1000	20.0	0.1	–	18.6	3.6	–	1.1	1.2	0.0	0.2	0.01
period_20	20.0	0.0	–	50.7	1.1	–	0.3	0.4	0.0	0.2	0.01
period_500000	20.0	0.1	55.9	13.7	5.5	–	1.3	1.4	0.1	0.2	0.01
random	20.0	1.7	5.0	0.2	8.2	0.7	1.1	1.4	0.1	0.3	0.02
howto	39.4	6.0	40.4	0.2	8.2	1.2	1.7	1.6	0.1	0.3	0.02
jdk13c	69.7	1.5	17.2	0.4	7.3	4.6	1.5	1.5	0.1	0.2	0.02
gcc-3.0	86.6	4.5	39.0	0.4	7.7	14.7	1.6	1.5	0.1	0.3	0.02
w3c	104.2	7.5	31.7	0.5	7.7	22.6	1.6	1.6	0.1	0.2	0.02
etext99	105.3	7.1	11.8	0.3	10.3	3.4	1.9	1.9	0.1	0.3	0.02
sprot34	109.6	2.3	33.0	0.2	9.1	1.5	1.7	1.8	0.1	0.3	0.02
reuters	114.7	2.4	–	0.5	9.1	2.9	1.7	1.7	0.1	0.3	0.02
linux-2.4.5	116.3	3.7	27.7	0.2	8.3	2.1	1.7	1.6	0.1	0.3	0.02
rfc	116.4	4.7	2.3	0.2	9.4	1.3	1.7	1.7	0.1	0.3	0.02

Table 3.5: *Construction times for ASCII datasets. We compared different ESA construction algorithms and normalized their running times by the text length (seconds per 1 M characters). Runs that did not finish within 10 h are denoted by dashes (–).*

A special adaptation to multiple sequences is not necessary as the lcp-interval tree depends solely on the lcp table and the text is not required for the construction of the child table.

dataset	size [Mb]	BWF	BWIP	DS	MM	QSORT	SKEW3	SKEW7	KASAI	KASAIIP	CHILDTAB
		running time [s/Mb]									
fss9	2.9	0.1	0.2	2.2	2.3	122.2	0.3	0.3	0.0	0.2	0.01
E_coli	4.6	0.3	0.4	0.1	5.8	0.6	0.8	0.6	0.1	0.2	0.02
4Chlamydophila	4.9	0.2	0.6	0.2	5.4	2.8	0.8	0.6	0.1	0.2	0.02
aaaa	6.0	0.0	0.0	0.0	0.5	–	0.1	0.1	0.0	0.0	0.02
6Streptococci	11.6	0.3	0.8	0.2	6.9	0.9	1.2	0.9	0.1	0.2	0.02
A_thaliana_Chr4	12.1	0.3	0.7	0.2	7.3	0.9	1.2	1.0	0.1	0.3	0.02
fss10	12.1	0.1	0.3	5.6	3.5	–	0.6	0.7	0.1	0.2	0.01
C_elegans_Chr1	14.2	0.3	0.7	0.2	6.7	69.9	1.1	0.9	0.1	0.2	0.02
hs_chr22	34.6	0.4	0.8	0.2	8.1	51.9	1.5	1.2	0.1	0.3	0.02
celegans	101.7	0.4	0.9	0.2	9.9	1.1	1.7	1.4	0.1	0.3	0.02
dmel	122.1	0.5	1.0	0.2	10.6	2.3	1.7	1.4	0.1	0.3	0.02
hs_chrX	145.8	0.4	1.0	0.2	10.7	3.3	1.7	1.4	0.1	0.3	0.02
mm_chrX	169.0	0.4	1.0	0.2	10.9	–	1.7	1.4	0.1	0.4	0.02
mm_chr2	184.3	0.5	1.0	0.2	11.5	–	1.7	1.4	0.1	0.3	0.02
hs_chr2	246.7	0.5	1.0	0.2	12.2	–	1.8	1.5	0.1	0.3	0.02

Table 3.6: *Construction times for DNA datasets. Compare with Table 3.5.*



(a) *suffix array construction* ***(b)*** *lcp and child table construction*

Figure 3.6: *Construction peak memory usage. We compared different ESA construction algorithms and normalized their peak memory usage by the length of the text (megabyte per 1 million text characters). Suffix array algorithms are shown in the left (a) and lcp and child table algorithms in the right plot (b).*

3.6 Applications

3.6.1 Searching the suffix array

The suffix array is a data structure that allows efficient searching of a text for any given pattern. As every substring is a prefix of a suffix, searching a substring p is equivalent

dataset	size [Mb]	running time [h:min:s] SKEW7	KASAI	CHILDTAB	total
internal memory variant					
hs_chrX	145.8	2:40	0:44	0:03	**3:28**
hs_chr2	246.7	6:00	1:23	0:05	**7:29**
hs	3096.5	1:38:09	33:43	1:02	**2:12:55**
external memory variant					
hs_chrX	145.8	1:16	0:30	0:05	**1:52**
hs_chr2	246.7	2:14	1:02	0:14	**3:32**
hs	3096.5	55:58	24:46	5:02	**1:25:47**
multiple sequences					
mammals	8111.1	8:28:52	3:27:18	26:10	**12:22:21**

dataset	size [Mb]	internal memory usage [GB] SKEW7	KASAI	CHILDTAB	max
internal memory variant					
hs_chrX	145.8	1.3	1.2	1.1	**1.3**
hs_chr2	246.7	2.1	2.1	1.9	**2.1**
hs	3096.5	26.0	26.0	23.1	**26.0**
external memory variant					
hs_chrX	145.8	2.9	1.4	1.7	**2.9**
hs_chr2	246.7	4.9	2.2	0.2	**4.9**
hs	3096.5	7.5	2.3	0.2	**7.5**
multiple sequences					
mammals	8111.1	3.5	2.3	0.2	**3.5**

dataset	size [Mb]	external memory usage [GB] SKEW7	KASAI	CHILDTAB	max
external memory variant					
hs_chrX	145.8	2.9	2.7	1.1	**2.9**
hs_chr2	246.7	4.9	4.6	1.8	**4.9**
hs	3096.5	61.4	57.7	23.1	**61.4**
multiple sequences					
mammals	8111.1	323.6	302.2	90.6	**323.6**

Table 3.7: *Construction times and internal and external peak memory usage for large DNA datasets. We compared the internal and external memory ESA construction of 3 single-sequence datasets and the external ESA construction of a multiple-sequence dataset consisting of three mammal genomes.*

to searching suffixes s_i that have a prefix p, whereas i is the begin position of an occurrence of p in s. If p has length m, a substring search in a suffix array can be conducted by a binary search in $\mathcal{O}(m \log n)$ running time. Figure 3.7 shows the pseudo-code of the binary search which is split into two separate searches for the sake of simplicity. The two functions determine the lower and the upper interval bound of the semi-open suffix array interval storing occurrences of the pattern p in the text s. In each iteration the pattern p is compared with the suffix $s_{\mathsf{suftab}[i]}$ beginning with the first characters.

Algorithm 3.11: FINDLOWER(s, p)

```
  input   : text s and pattern p
  output  : minimal l with p ≤_|p| s_suftab[l]
1 l1 ← 0
2 l2 ← |s|
3 while l1 < l2 do
4     i ← ⌊(l1+l2)/2⌋
5     if s_suftab[i] <_|p| p then
6         l1 ← i + 1
7     else
8         l2 ← i
9 return l1
```

Algorithm 3.12: FINDUPPER(s, p)

```
  input   : text s and pattern p
  output  : minimal r with p <_|p| s_suftab[r]
1 r1 ← 0
2 r2 ← |s|
3 while r1 < r2 do
4     i ← ⌊(r1+r2)/2⌋
5     if s_suftab[i] ≤_|p| p then
6         r1 ← i + 1
7     else
8         r2 ← i
9 return r1
```

Figure 3.7: *Binary search on the suffix array.* FINDLOWER *and* FINDUPPER *determine the interval boundaries l and r of the suffix array such that* suftab$[l..r)$ *stores the begin positions of all occurrences of p in s.*

With a simple so-called *mlr-heuristic* [Manber and Myers, 1993] the average running time can be reduced by keeping track of the longest-common prefix lengths between the suffixes at the interval borders of the current search interval and the search pattern p. If for strings $L, M, R \in \Sigma^*$ with $L \leq_{\text{lex}} M \leq_{\text{lex}} R$ holds $l := |\operatorname{lcp}\{p, L\}|$ and $r := |\operatorname{lcp}\{p, R\}|$ then obviously L and R share a common prefix of length $mlr := \min(l, r)$. Obviously M begins with the same prefix shared by L, R and p of length mlr and therefore the first mlr characters can be skipped during the comparison in line 6 of the improved binary searches in Figure 3.8. Although the worst-case running time is still $\mathcal{O}(m \log n)$ the heuristic improves the average running time in practice. With an additional data structure, called lcp-tree, that determines the values $\left|\operatorname{lcp}\left\{s_{\text{suftab}[l_1-1]}, s_{\text{suftab}[i]}\right\}\right|$ and $\left|\operatorname{lcp}\left\{s_{\text{suftab}[i]}, s_{\text{suftab}[l_2]}\right\}\right|$ in $\mathcal{O}(1)$ the running time can be improved to $\mathcal{O}(m + \log n)$ [Manber and Myers, 1993]. However, experiments show that the running time deteriorates in practice due to the overhead in accessing the lcp-tree. We compared the search times of the naive binary search, the binary search with mlr-heuristic, the lcp-tree binary search, and the suffix tree search (following section) on texts over different alphabets. The results in Figure 3.9 show that in all cases the mlr-heuristics outperforms the naive and lcp-tree search. The suffix tree search is faster for small pattern lengths as in contrast to the binary searches its running time does not depend on the length of the text. On large alphabets the suffix tree search deteriorates due to the larger number of child edges which are searched sequentially.

3.6.2 Traversing the suffix tree

With the enhanced suffix array consisting of suffix array, lcp table, and child table, we are now able to top-down traverse the nodes of the suffix tree of a text which actually is a node traversal of the corresponding lcp-interval tree.

Algorithm 3.13: FINDLOWERH(s, p)

input : text s and pattern p
output : minimal l with $p \leq_{|p|} s_{\text{suftab}[l]}$

```
1  l_1 ← 0, k_1 ← 0
2  l_2 ← |s|, k_2 ← 0
3  while l_1 < l_2 do
4      i ← ⌊(l_1 + l_2)/2⌋
5      k ← min{k_1, k_2}
6      if s_{suftab[i]+k} <_{|p|-k} p_k then
7          l_1 ← i + 1
8          k_1 ← k + |lcp{s_{suftab[i]+k}, p_k}|
9      else
10         l_2 ← i
11         k_2 ← k + |lcp{s_{suftab[i]+k}, p_k}|
12 return l_1
```

Algorithm 3.14: FINDUPPERH(s, p)

input : text s and pattern p
output : minimal r with $p <_{|p|} s_{\text{suftab}[r]}$

```
1  r_1 ← 0, k_1 ← 0
2  r_2 ← |s|, k_2 ← 0
3  while r_1 < r_2 do
4      i ← ⌊(r_1 + r_2)/2⌋
5      k ← min{k_1, k_2}
6      if s_{suftab[i]+k} ≤_{|p|-k} p_k then
7          r_1 ← i + 1
8          k_1 ← k + |lcp{s_{suftab[i]+k}, p_k}|
9      else
10         r_2 ← i
11         k_2 ← k + |lcp{s_{suftab[i]+k}, p_k}|
12 return r_1
```

Figure 3.8: *Binary search with mlr-heuristic. FINDLOWERH and FINDUPPERH return the same interval boundaries as FINDLOWER and FINDUPPER. A heuristic determines a lower bound on the lcp length between $s_{\text{suftab}[i]}$ and p and thereby reduces the number of character comparisons on average.*

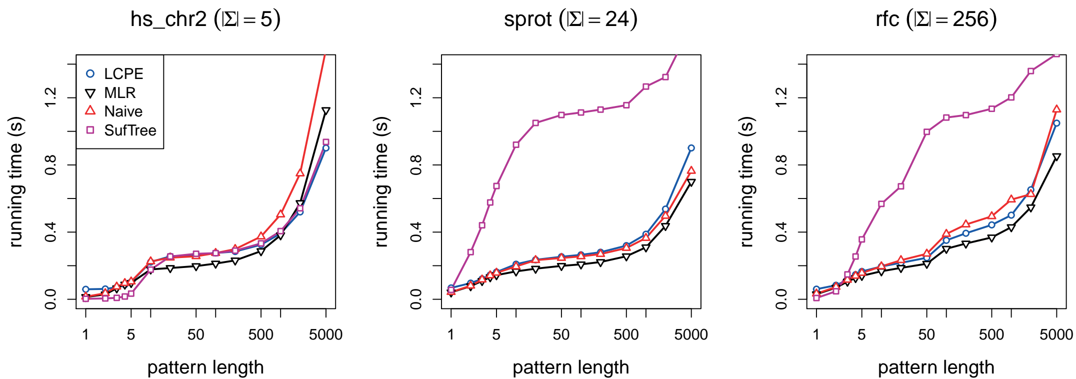


Figure 3.9: *Comparison of the ESA based search algorithms available in SeqAn. We compared the running times required to search 100,000 patterns using the naive binary search (Naive), the binary search with mlr-heuristic (MLR), the binary search with lcp-tree (LCPE), and the suffix tree search (SufTree). As texts we used the first 100 M characters of a DNA, amino acid, and natural language text. Patterns are random substrings of varying length.*

Top-down traversal

We implemented a so-called *top-down iterator* and functions to go to the root node, to go down to the leftmost child, and to go right to the next sibling of the currently visited node, where the children are lexicographically ordered by their edge labels from left to right. The top-down iterator maintains the values *lb* and *rb*, the lcp-interval boundaries of the

Algorithm 3.15: GoRoot($iter$)

```
input     : suffix tree iterator iter
1  n ← |lcp| − 1          // n is the text length
2  iter.lb ← 0
3  iter.rb ← n
4  iter.parentRb ← ⊥
```

Algorithm 3.16: isNextL(i)

```
input     : ℓ-index i
1  j ← cld[i]
2  if i < j and lcp[i] = lcp[j] then
3      return true
4  return false
```

Algorithm 3.17: GoDown($iter$)

```
input     : suffix tree iterator iter
output    : returns true on success
1  if iter.rb − iter.lb ≤ 1 then
2      return false;
3  if iter.rb ≠ iter.parentRb then
4      iter.parentRb ← iter.rb
       // get up value of right boundary
5      iter.rb ← cld[iter.rb − 1]
6  else
       // get down value of left boundary
7      iter.rb ← cld[iter.lb]
8  return true
```

Algorithm 3.18: GoRight($iter$)

```
input     : suffix tree iterator iter
output    : returns true on success
1  if iter.parentRb ∈ {⊥, iter.rb} then
2      return false
3  iter.lb ← iter.rb
4  if isNextL(iter.rb) then
       // iter.rb has a succeeding ℓ-index
5      iter.rb ← cld[iter.rb]
6  else
       // iter.rb is the last ℓ-index
7      iter.rb ← iter.parentRb
8  return true
```

currently visited node, and $parentRb$ the right boundary of the parent node. It starts in the root of the lcp-interval tree which is the interval $[lb..rb) = [0..n)$. For the root node, we initialize $parentRb$ with $\perp$, see Algorithm 3.15. The intervals in the lcp-interval tree are distinct from each other and two iterators can be compared by comparing their boundary pairs. For leaf nodes hold $rb - lb = 1$.

When moving the iterator to the first child of the current node, the left boundary remains the same whereas the smallest ℓ-index in ℓ-indices(lb, rb) becomes the new right boundary rb and $parentRb$ the former value of rb. If the current node is not the last child of its parent ($rb \neq parentRb$), the smallest ℓ-index in ℓ-indices(lb, rb) is the up value of rb stored at $\mathsf{cld}[rb-1]$, otherwise it is the down value of lb stored only in this case at $\mathsf{cld}[lb]$. The corresponding pseudo-code is given in Algorithm 3.17. Moving the iterator to the next sibling is possible iff $rb \notin \{\perp, parentRb\}$, i.e. for all but the last sibling. Then lb becomes the former rb and rb becomes, if existent, its next ℓ-index or $parentRb$, otherwise (see Algorithm 3.18).

Besides the GoDown function that moves an iterator along the leftmost edge to the first child, we also implemented variants to go down to a child at the end of an edge starting with a certain character or to go down along the path of string characters. All the traversal functions return a boolean which is true, iff the iterator could be successfully moved.

Random traversal

If only the 3 tables of the enhanced suffix array are given, it is not possible to move a top-down iterator upwards the tree in $\mathcal{O}(1)$ time. It is however possible to recover the state the iterator had in the parent node by manually maintaing a stack of top-down it-

Algorithm 3.19: GONEXT_PRE(it)

```
input : suffix tree iterator it
  // try go down, if not try go right
1 if not goDown(it) and not goRight(it) then
2     while goUp(it) and not goRight(it) do
          // go up until we can go right
3 if IsRoot(it) then
      // end when entering the root node
4     (it.lb, it.rb) ← (−1, −1)
5     return false
6 return true
```

Algorithm 3.20: GONEXT_POST(it)

```
input : suffix tree iterator it
1 if goRight(it) then
2     while goDown(it) do
          // try go right and down to leaf
3 else                              // otherwise go up
4     if not goUp(it) then
          // end after halting in root
5         (it.lb, it.rb) ← (−1, −1)
6         return false
7 return true
```

erator copies. To provide a more convenient interface we implemented the subclass *top-down history iterator* which extends the top-down iterator by a stack. The stack stores the lcp-interval boundaries of nodes on the path to the root. We adapted the GOROOT and GODOWN functions to clear the stack and to push the current interval boundary pair (lb, rb) onto the stack before going down. The new function GOUP simply replaces (lb, rb) by the topmost pair and removes it from stack. We also need to adapt $parentRb$, which is set to the new topmost rb value on stack.

Depth-first search

While the two iterators above require 3 tables of the enhanced suffix array, we have seen in Section 3.5.1 that only 2 tables are required to conduct a postorder depth-first search (DFS). Algorithm 3.9 can be executed to report all nodes. However, in some applications it is more appropriate to have an iterator that can be moved node-by-node. Therefore, we implemented a light-weight *bottom-up iterator* which maintains lb and rb, the current node boundaries, and provides functions GOBEGIN and GONEXT. Like a coroutine [Knuth, 1997], GONEXT resumes and suspends the execution of Algorithm 3.9 between two calls of report($interval$) and instead of reporting the interval it sets lb and rb accordingly. GOBEGIN executes Algorithm 3.9 up to the first call of report($interval$). The last node the iterator halts in is the root node. If GONEXT is called again, it sets the iterator in a defined end-state and indicates that by returning false.

We provide the same depth-first search interface for the top-down history iterator but implemented GONEXT by means of GODOWN, GORIGHT, and GOUP to traverse all nodes in the order of either a postorder or a preorder depth-first search. The pseudo-codes of both DFS variants are shown in Algorithms 3.19 and 3.20. In contrast to postorder DFS, in a preorder DFS each node is traversed before its children.

Some applications require to traverse nodes only up to a certain tree or string depth or have a different criteria to skip a whole subtree in the traversal. To meet this requirement, we implemented GONEXTRIGHT and GONEXTUP. GONEXTRIGHT skips the subtree of the current node and proceeds with the next sibling by omitting GODOWN in line 1 of Algorithm 3.19. Analogously GONEXTUP skips the subtree and all right siblings of the current node.

3.6.3 Accessing the suffix tree

All of the suffix tree iterators described above store a pair of boundaries of the current lcp-interval $[lb..rb)$. We are not only able to traverse the lcp-interval tree but also to access all information about the corresponding suffix tree, given only the suffix array, the lcp table, and the node boundary pairs.

In the following, we are going to determine the concatenation string of the current node, i.e. the concatenation of characters on the path from the root node to the node the iterator points at. The sentinel character $ at the end of each leaf edge should be omitted as it is not part of the text and especially for multiple sequences would bloat the alphabet. If v is the suffix tree node that corresponds to the lcp-interval $[lb..rb)$, the concatenation string without sentinel equals the longest-common prefix ω of the ℓ-interval $[lb..rb)$, see Lemma 3.1. Thus, ω especially is the ℓ-prefix of the lexicographically smallest suffix of the ℓ-interval and it holds $\omega = s_{\mathsf{suftab}[lb]}[0..\ell)$. It remains to determine ℓ. If $rb - lb = 1$, v is a leaf and ℓ equals the suffix length $|s_{\mathsf{suftab}[lb]}| = n - \mathsf{suftab}[lb]$. For $rb - lb > 1$, v is an inner node and ℓ is the lcp value of any, especially the smallest index in ℓ-indices(lb, rb). In the previous section, we described how to determine the smallest ℓ-index (used in Algorithm 3.17). The function repLength (Algorithm 3.21) computes ℓ using a similar approach. It only differs in the way of testing whether the current node is the last of its siblings, as *iter.parentRb* is not available for the bottom-up iterator. In line 3, the child table entry at position *iter.lb* is read. It either contains a nextℓIndex-value if the current node is the first or an inner sibling or a down-value if the node is the last sibling. In the latter case, the entry is less than *iter.rb*. If it is greater than or equal to *iter.rb*, it contains a nextℓIndex-value, the current node is not the last sibling and the right boundary is an ℓ-index with an up-value at position $iter.rb - 1$. The concatenation string of a node is returned by representative (Algorithm 3.22).

Algorithm 3.21: repLength(*iter*)

input : suffix tree iterator *iter*
output : returns the length of the concatenation string

```
1  if iter.rb − iter.lb = 1 then
2      return |s| − suftab[iter.lb];              // determine suffix length
3  i ← cld[iter.lb]                    // try to get down value of left boundary
4  if (iter.rb ≤ i) then                          // was it a nextℓIndex value?
5      i ← cld[iter.rb − 1]                       // get up value of right boundary
6  return lcp[i]                       // value of the ℓ-interval [iter.lb..iter.rb)
```

To only determine the label of the edge from the parent to the current node, we need to compute the ℓ-value of the parent interval of $[lb..rb)$ in the interval tree. This could be easily accomplished for a top-down history iterator by going up one node and calling repLength. However, there is another way that works for every tree iterator and only requires lb and rb. Assume $[lb..rb)$ is not the root node and $[lb'..rb')$ its parent interval. In Section 3.1.3, we have seen that the child interval boundaries are contained in the set ℓ-indices$(lb', rb') \cup \{lb', rb'\}$ and at least one boundary of every interval is an ℓ-index

Algorithm 3.22: REPRESENTATIVE($iter$)

input : suffix tree iterator $iter$
output : returns the concatenation string, also called representative

```
1  i ← suftab[iter.lb]                          // start of the first suffix in ℓ-interval
2  return s[i..i + REPLENGTH(iter))             // return its prefix of length ℓ
```

of the ℓ-interval $[lb'..rb')$. Thus, either both boundaries are ℓ-indices or only one is an ℓ-index and then the other's lcp value must be less than ℓ. In either case the maximal lcp value of the boundaries equals ℓ. For the case that $[lb..rb) = [0..n)$ is the root interval, we set $\ell = 0$ as the root has no parent edge. Thus, it holds $\ell = \max\{\mathsf{lcp}[lb], \mathsf{lcp}[rb], 0\}$ and the parent edge label is the suffix of the concatenation string starting at position ℓ, compare with Algorithm 3.23.

Algorithm 3.23: PARENTEDGELABEL($iter$)

input : suffix tree iterator $iter$
output : returns label of the edge between parent and current node

```
1  l ← max{lcp[iter.lb], lcp[iter.rb], 0}      // length of the parent's representative
2  t ← REPRESENTATIVE(iter)
3  return t[l..|t|)                             // cut the first l characters
```

The occurrences of a string t in the text s are the start positions of suffixes of s beginning with t. Hence, if t is the concatenation string of a suffix tree node, its occurrences can be determined by traversing the leaves in the node's subtree. Given the enhanced suffix array, the set of suffix start positions can directly be obtained, as for a node $[lb..rb)$ the substring $\mathsf{suftab}[lb..rb)$ contains the start positions of its concatenation string. Algorithm 3.24 shows the corresponding pseudo-code.

Algorithm 3.24: GETOCCURRENCES($iter$)

input : suffix tree iterator $iter$
output : reports the occurrences of the node's concatenation string

```
1  for i ← iter.lb to iter.rb − 1 do
2      print "occurrence at position " suftab[i]
```

3.6.4 Repeat search

Besides the above-mentioned iterators, we implemented special purpose iterators for finding maximal repeats, supermaximal repeats, and maximal unique matches. A repeat is a substring that occurs at least twice in a text. It is called maximal, if the substring cannot be extended to the left or right such that it still has the same number of occurrences in the text. The following definitions will more precisely formalize the term repeat.

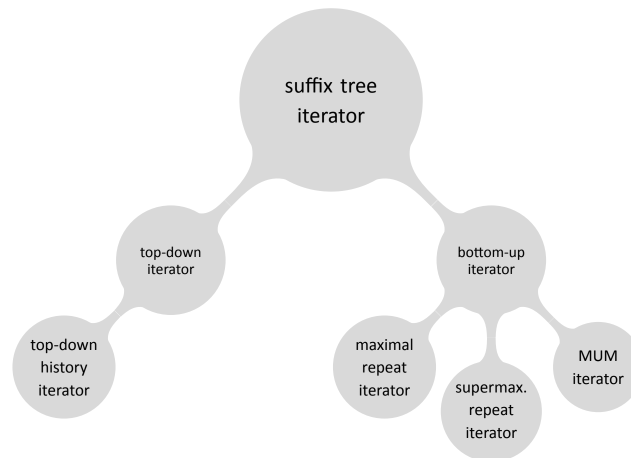


Figure 3.10: *Class hierarchy of suffix tree iterators. The superclass of all iterators is the* suffix tree iterator *which maintains lb and rb, the boundaries of the current lcp-interval, and provides functions to access the corresponding suffix tree node as described in Section 3.6.3.*

Definition 3.6 (repeated pair)**.** For a given text s of length n, the triple (p_1, p_2, l) is called a *repeated pair*, iff $p_1, p_2 \in [0..n - l]$, with $p_1 \neq p_2$ and the two substrings $s[p_1..p_1 + l)$ and $s[p_2..p_2 + l)$ are equal.

Definition 3.7 (maximal repeated pair)**.** A repeated pair (p_1, p_2, l) is *right maximal* if $(p_1, p_2, l + 1)$ is not a repeated pair. It is *left maximal* if $(p_1 - 1, p_2 - 1, l)$ is not a repeated pair. A *maximal repeated pair* is left and right maximal.

Given a text s. A string α is called a *maximal repeat*, iff there is at least one maximal repeated pair $(p_1, p_2, |\alpha|)$ with $\alpha = s[p_1..p_1 + |\alpha|) = s[p_2..p_2 + |\alpha|)$. For example, the text $s =$ `xabcyabcwabcyz` contains the two maximal repeats `abc`, with maximal repeated pairs $(1, 5, 3)$ and $(5, 9, 3)$, and `abcy` with the only maximal repeated pair $(1, 9, 4)$, see Figure 3.11a.

Definition 3.8 (supermaximal repeat)**.** A supermaximal repeat is a maximal repeat which is not a substring of another maximal repeat.

As mentioned above, the text $s =$ `xabcyabcwabcyz` contains the two maximal repeats `abc` and `abcy`. As `abcy` is a superstring of `abc`, it is the only contained supermaximal repeat, see Figure 3.11b.

Definition 3.9 (maximal unique match)**.** For given strings $s^1, s^2, \dots, s^m$, a unique match is a string that occurs exactly once in every s^i, with $i \in [1..m]$. A maximal unique match (MUM) is a unique match that is not a substring of another unique match.

One important observation is that for every maximal repeat or maximal unique match α there is an inner suffix tree node $\overline{\alpha}$. This is due to the right-maximality of α which implies that different characters or sentinels follow the α prefix in two distinct suffixes. Vice

$$s = \begin{cases} \text{xabcyabcwabcyz} \\ \text{xabcyabcwabcyz} \\ \text{xabcyabcwabcyz} \end{cases} \qquad s = \text{xabcyabcwabcyz} \qquad \begin{array}{l} s_1 = \text{axyzbcxyzcxyza} \\ s_2 = \text{bxyzaxyzaxyzbc} \\ s_3 = \text{baxyzaxyzbxyzb} \end{array}$$

***(a)** maximal repeats* ***(b)** supermaximal repeat* ***(c)** MUM*

Figure 3.11: *Repeat examples. The text* `xabcyabcwabcyz` *contains 3 maximal repeated pairs of 2 repeats (a) and only one supermaximal repeat (b). The 3 strings shown in (c) contain only one MUM.*

versa, for every inner node there is a repeated pair that cannot be extended to the right. For the exact proof, we refer the reader to Chapter 7.12.1 in [Gusfield, 1997]. As a consequence of this observation, maximal and supermaximal repeats as well as MUMs can be found by traversing the nodes of the suffix tree. We implemented 3 special purpose iterators as subclasses of the bottom-up iterator that only halt at nodes that are maximal or supermaximal repeats or MUMs. The following paragraphs describe their functionality in more detail. We assume that the alphabet size is fixed.

Searching maximal repeats in linear-time

For a given text s and a minimum length n_0, the *maximal repeat problem* is to find all maximal repeats α with $|\alpha| \geq n_0$ and all corresponding maximal pairs. According to the observation above, the right-maximality is given for every inner node v, i.e. every pair of suffixes s_{p_1} and s_{p_2} from different subtrees of v form a right maximal repeated pair $(p_1, p_2, |concat(v)|)$. The pair is also left maximal, if p_1 or p_2 equals 0 or the characters left of the suffixes differ. If at least one such pair exists for a node v, $concat(v)$ is a maximal repeat.

In Section 7.12.3 of [Gusfield, 1997], the author proposed a suffix tree algorithm to efficiently enumerate all maximal pairs in $\mathcal{O}(n|\Sigma| + k)$ time, where k is the number of maximal repeated pairs. This algorithm was later adapted to enhanced suffix arrays by Abouelhoda *et al.* [2002a]. Fundamental to both algorithms is to traverse the suffix tree from bottom up and for every tree node to partition the sets of suffixes in the subtree according to their preceding character. To well-define the preceding character, we define \$ to precede the suffix starting at position 0. For a tree node $[i..j)$ and a character $x \in \Sigma \cup \{\$\}$, the partition of start positions of suffixes preceded by character x in the subtree of $[i..j)$ is:

$$\mathcal{P}_{[i..j)}(x) = \begin{cases} \{0 \mid 0 \in \text{suftab}[i..j)\}, & \text{if } x = \$ \\ \{p \in \text{suftab}[i..j) \mid p \neq 0 \wedge s[p-1] = x\}, & \text{else.} \end{cases} \tag{3.26}$$

Let v be an inner suffix tree node that corresponds to the lcp-interval $[i..j)$ with child intervals $[l_0..l_1)$, $[l_1..l_2)$, ..., $[l_{m-1}..l_m)$, where $l_0 = i$ and $l_m = j$ holds. The set of maximal repeated pairs for $concat(v)$ is the union of Cartesian products of partitions from different

subtrees (right-maximality) and different preceding characters (left-maximality):

$$\mathcal{R}_{[i..j)} = \bigcup_{\substack{a,b\in[0..m),\\ a<b}} \bigcup_{\substack{x,y\in\Sigma\cup\{\$\},\\ x\neq y}} \mathcal{P}_{[l_a..l_{a+1})}(x) \times \mathcal{P}_{[l_b..l_{b+1})}(y) \times \big\{\big|concat(v)\big|\big\}. \quad (3.27)$$

If the set $\mathcal{R}_{[i..j)}$ is empty, the concatenation string $concat(v)$ is not a maximal repeat. Otherwise, it contains all maximal repeated pairs of the maximal repeat $concat(v)$.

The algorithm described in [Gusfield, 1997; Abouelhoda *et al.*, 2002a] computes the sets $\mathcal{P}_{[i..j)}(x)$ for every node $[i..j)$ in the lcp-interval tree from bottom up. It begins in the leaves $[k..k+1)$, where $\mathcal{P}_{[k..k+1)}(x)$ is empty for all but one character $x \in \Sigma \cup \{\$\}$. It is non-empty and equals the singleton $\{k\}$ for the character that precedes the suffix s_k. Whenever a child node $[l_b..l_{b+1})$ is visited the last time during the postorder DFS, its sets $\mathcal{P}_{[l_b..l_{b+1})}(\cdot)$ are joined to the sets of its parent node $[i..j)$. At the moment, between leaving the child node and appending its sets, the parent $[i..j)$ stores the union of sets of all left siblings of $[l_0..l_1)$, ..., $[l_{b-1}..l_b)$ which equals $\mathcal{P}_{[l_0..l_b)}(\cdot)$. The Cartesian products of the sets $\mathcal{P}_{[l_0..l_b)}(x)$ and $\mathcal{P}_{[l_b..l_{b+1})}(y)$, with $x \neq y$, constitute maximal pairs and are output for every child $[l_b..l_{b+1})$, with $b \in [1..m)$. It becomes clear that the algorithm is correct, after equivalently rewriting equation 3.27:

$$\mathcal{R}_{[i..j)} = \bigcup_{b\in[1..m)} \bigcup_{\substack{x,y\in\Sigma\cup\{\$\},\\ x\neq y}} \mathcal{P}_{[l_0..l_b)}(x) \times \mathcal{P}_{[l_b..l_{b+1})}(y) \times \big\{\big|concat(v)\big|\big\}. \quad (3.28)$$

We implemented the described algorithm as a specialized bottom-up iterator. The iterator is extended by a stack that stores the position sets for every $x \in \Sigma \cup \{\$\}$ for the current tree node and all of its ancestors. Sets are represented as linked lists and thus can be joined in constant time. They are no longer used after they were joined to the parent node. As a consequence, all active sets are disjoint and the linked lists can be stored in a single string P of length n over $[0..n)$, where the sets $\{17, 42, 23\}$ and $\{10, 20\}$ are represented by:

$$P[17] = 42, \quad P[42] = 23, \quad P[23] = 17,$$
$$P[10] = 20, \quad P[20] = 10$$

and joined by switching their first links to $P[17] = 20$, $P[10] = 42$. Instead of directly storing positions in the sets, our implementation stores the indices of the corresponding suffix array entries. This enables us to seamlessly extend the approach to multiple sequences. The bottom-up iterator halts in every node v, where $concat(v)$ is a maximal repeat of minimum length $n_0 \leq \big|concat(v)\big|$. More precisely it might halt multiple times in v, whenever it leaves a child node whose position sets, in conjunction with the sets of left siblings, contribute maximal repeated pairs. The user may proceed with the next maximal repeat or with a second iterator enumerate these maximal repeated pairs. Deciding whether a child contributes maximal repeated pairs and joining its position sets can be done in $\mathcal{O}(|\Sigma|)$ time. The time required for enumerating these maximal repeated pairs is proportional to the number of pairs.

Theorem 3.1. *The time to enumerate all maximal repeats of a text of length n is $\mathcal{O}(n|\Sigma|)$. If the text contains overall k maximal repeated pairs of minimal length n_0, the overall time to output them is $\mathcal{O}(n|\Sigma| + k)$.*

We extend the repeated pair definition (Definition 3.6) straightforward to multiple sequences as well as the dependent maximal and supermaximal repeat definitions.

Definition 3.10 (generalized repeated pair)**.** Given strings $s^1, \dots, s^m$ of lengths $n^1, \dots, n^m$, the triple $((i_1, j_1), (i_2, j_2), l)$ is called a *(generalized) repeated pair*, iff $(i_1, j_1), (i_2, j_2) \in \bigcup_{i \in [1..m]} \{i\} \times [0..n^i - l]$, with $(i_1, j_1) \neq (i_2, j_2)$ and the two substrings $s^{i_1}[j_1..j_1 + l)$ and $s^{i_2}[j_2..j_2 + l)$ are equal.

The suffix tree algorithm described above as well as our implemented iterator can without a change be applied to the generalized suffix tree to solve the generalized maximal repeat problem in $\mathcal{O}(n|\Sigma| + k)$ time, where $n = \sum_{i \in [1..m]} n^i$ is the overall text length.

Searching supermaximal repeats in linear-time

For a given text s and a minimum length n_0, the *supermaximal repeat problem* is to find all supermaximal repeats α with $|\alpha| \geq n_0$. A supermaximal repeat is a maximal repeat that is not a substring of a larger maximal repeat. Whereas one maximal repeated pair suffices for being a maximal repeat, all possible pairs of text occurrences of a supermaximal repeat must form maximal repeated pairs. Otherwise a pair of occurrences could be extended to the left or right to form a larger maximal repeated pair of a superstring.

Theorem 3.2. *A substring of a text is a supermaximal repeat, iff it occurs at least twice and all its occurrences have pairwise distinct preceding characters and pairwise distinct succeeding characters (if existent).*

Considering the suffix tree, supermaximal repeats are inner nodes whose children are leaves (criterion 1) and the characters preceding the leaf suffixes are pairwise distinct (criterion 2). Testing a node for being leaf can be done in constant time. As every child node is tested at most once and the time for enumerating the children of a node is proportional to their number, the overall time for testing the first criterion is $\mathcal{O}(n)$. Deciding whether all characters preceding the occurrences of a single node are pairwise distinct takes $\mathcal{O}(|\Sigma|)$ time, deciding it for all nodes consequently takes $\mathcal{O}(n|\Sigma|)$. The $\mathcal{O}(|\Sigma|)$ time per node is required to erase a boolean vector of size $\mathcal{O}(|\Sigma|)$ and to check for every occurrence if the preceding character was marked and otherwise mark it.

For large alphabets, a simple trick avoids erasing the vector and reduces the time to $\mathcal{O}(o)$ per node, where o is the number of occurrences. Instead of using a vector of booleans, we store for each character a unique identifier of the last visited criterion-1-node with occurrences preceded by that character. This way, the vector needs to be erased only once per traversal instead of once per node and two occurrences of the same node preceded by the same character can be detected by the same node identifier. As the vector is updated only for the occurrences of nodes fulfilling criterion 1, i.e. all children are leaves, the total number of updates is not greater than the total number of leaves. Hence, the overall running time to test criterion 2 is $\mathcal{O}(n + |\Sigma|)$.

Theorem 3.3. *The supermaximal repeat problem for a text of length n can be solved in $\mathcal{O}(n + |\Sigma|)$ time.*

While our above-described approach can be applied to *any* suffix tree index, Abouelhoda *et al.* [2004] proposed an algorithm that solves the supermaximal repeat problem using the suffix array and the lcp table of a given text. Although both have the same asymptotical time consumption, their approach is twice as fast in practice. Fundamental is the observation that nodes whose children are leaves are ℓ-intervals where every element in the interval is an ℓ-index, i.e. the ℓ-interval is a local maximum in the lcp table. In a linear scan over the lcp table, all intervals $[i..j)$ can be determined where $\mathsf{lcp}[i] < \mathsf{lcp}[i] = \mathsf{lcp}[i+1] = \ldots = \mathsf{lcp}[j-1] > \mathsf{lcp}[j]$ holds. These intervals $[i..j)$ are ℓ-intervals that fulfill the first supermaximality criterion. The second criterion can be tested as described above by comparing the preceding characters of suffixes starting at positions $\mathsf{suftab}[i], \mathsf{suftab}[i+1], \ldots, \mathsf{suftab}[j-1]$.

For both variants, the generic and the specializiation for enhanced suffix arrays, we implemented a supermaximal repeat iterator which, just like the maximal repeat iterator, can be applied to multiple sequences as well.

Searching MUMs in linear-time

Given multiple strings $s^1, s^2, \ldots, s^m$ and a minimum length n_0, the *MUM problem* is to find all MUMs α with $|\alpha| \geq n_0$. A unique match is a substring that occurs exactly once in every string, i.e. the number of occurrences equals m and there is no string s^i containing two occurrences. A *unique match* α is not maximal, iff it is contained in a longer unique match. As the longer unique match must have the same number of occurrences, all occurrences of α would be preceded or succeeded by the same character.

Theorem 3.4. *A unique match is a MUM, iff it has two occurrences preceded by different characters and two occurrences succeeded by different characters.*

From the latter follows, that MUMS are inner nodes of the suffix tree. They can be found by traversing the suffix tree while skipping all nodes with more or less than m occurrences. For every remaining node it can be tested in $\mathcal{O}(m)$ time whether the m occurrences are preceded by the same character or contained twice in a string s_i. If neither the former nor the latter holds, the node is a MUM and can be reported. We implemented the MUM iterator as a subclass of the bottom-up iterator, where we overloaded goBegin and goNext to only halt in MUMs with a minimal length n_0. The time spent in every node is bound linear in the number of children. Thus the overall time for reporting all MUMs is linear in the number of nodes.

Theorem 3.5. *For multiple strings of overall text length n, the MUM problem can be solved in $\mathcal{O}(n)$ time.*

CHAPTER 4

Lazy Suffix Tree

In the previous chapter, we proposed repeat search algorithms which construct and traverse the whole suffix tree of a text. Some applications, however, require to traverse only an upper part or a single path of the tree. In such cases another index, described in the following, becomes more appropriate.

A *lazy suffix tree* [Giegerich *et al.*, 2003] is a suffix tree whose nodes are created on demand, i.e. when they are visited the first time in a top-down traversal. The suffix tree construction is *deferred* to the traversal and driven by it. The term *deferred data structuring* was first introduced by Karp *et al.* [1987]. It denotes a concept where data structures are query-driven constructed on demand. Ching *et al.* [1990] distinguish between *static* and *dynamic* deferred data structures, depending on whether the underlying dataset must remain constant or permits dynamic changes, e.g. insertions or removals of elements. As the lazy suffix tree assumes a constant text, it falls in the first category.

Depending on the usage scenario, using a lazy suffix tree can significantly improve on the overall running time and memory consumption compared to an enhanced suffix array. In [Weese and Schulz, 2008; Schulz *et al.*, 2008a; Weese *et al.*, 2013] we propose different applications of a lazy suffix tree that outperform competitive algorithms that use suffix trees or enhanced suffix arrays. In Chapter 7, we describe in detail an application of the lazy suffix tree to frequency string mining, where we exploit a property of the on-demand construction for the efficient computation of substring frequencies in databases, and analyze its performance on different real-world datasets.

Giegerich *et al.* introduced the first lazy suffix tree data structure that utilizes the *write-only, top-down algorithm* [Giegerich and Kurtz, 1995] for the on-demand node expansion. We first describe this algorithm and the original lazy suffix tree data structure. Then, we propose our own lazy suffix tree variant that supports multiple sequences and provides the same suffix tree interface as the enhanced suffix array (described in Section 3.6.2). At the end of this chapter, we propose different applications that benefit from the on-demand tree construction.

4.1 The *wotd* algorithm

The *wotd* (write-only, top-down) algorithm was first proposed by Giegerich and Kurtz [1995] as a purely functional suffix tree construction algorithm. In a follow-up paper, Giegerich *et al.* [2003] introduced a data structure to represent the partially constructed suffix tree of a single string and restated their algorithm in an imperative language. The

basic idea of the wotd algorithm is to determine the children of a branching suffix tree node by partitioning the set of corresponding suffixes by the character following the longest common prefix. Beginning with only the root node it recursively expands a directed tree step-by-step up to the entire suffix tree.

We consider a given non-empty text s of length n and a rooted, directed tree T that in every state of the algorithm is a subgraph of the suffix tree including its root, in the following referred to as *partial suffix tree*. Let R be a function that maps any string $\alpha \in \Sigma^*$ to the set of suffixes of $s\$$ that begin with α:

$$R(\alpha) := \left\{ \alpha\beta \;\middle|\; \alpha\beta \text{ is a suffix of } s\$ \right\} \setminus \{\$\}. \tag{4.1}$$

Given a branching suffix tree node $\overline{\alpha}$, $R(\alpha)$ contains the concatenation strings of the leaves below $\overline{\alpha}$. The children of $\overline{\alpha}$ can be determined as follows: Divide $R(\alpha)$ into non-empty groups $R(\alpha c_1), \dots, R(\alpha c_m)$ of suffixes, where character $c_i \in \Sigma \cup \{\$\}$ follows the common α-prefix. Let $\alpha c_i \beta_i$ be the longest-common prefix of $R(\alpha c_i)$, which for singleton groups equals the only contained suffix. For non-singleton groups, $\overline{\alpha c_i \beta_i}$ is a branching node in the suffix tree, as there are two suffixes of $s\$$ that differ in the character following their common prefix $\alpha c_i \beta_i$. Singleton groups contain suffixes of $s\$$ which correspond to leaves $\overline{\alpha c_i \beta_i}$ in the suffix tree. As every suffix with prefix αc_i also has a prefix $\alpha c_i \beta_i$, there is no branching node between $\overline{\alpha}$ and $\overline{\alpha c_i \beta_i}$. Hence every $\overline{\alpha c_i \beta_i}$ can be inserted as a child of $\overline{\alpha}$ in T, which remains a partial suffix tree. This procedure, called *node expansion*, is recursively repeated for every newly inserted branching node; Algorithm 4.1 shows the corresponding pseudo-code. The wotd-algorithm begins with T consisting of only the root node and expands it and all its descendants, see Algorithm 4.2.

Algorithm 4.1: WOTDEAGER($T, \overline{\alpha}$)

```
  input   : partially constructed suffix tree T and node ᾱ

1 divide R(α) into subsets R(αc) of suffixes where character c follows the α-prefix
2 foreach c ∈ Σ ∪ {$} and R(αc) ≠ ∅ do
3     αcβ ← lcp R(αc)
4     if |R(αc)| = 1 then                                   // leaf node
5         add leaf αcβ (overlined) as a child of ᾱ in T
6     else                                                  // branching node
7         add inner node αcβ (overlined) as a child of ᾱ in T
8         wotdEager(T, αcβ (overlined))                     // recurse into child subtree
```

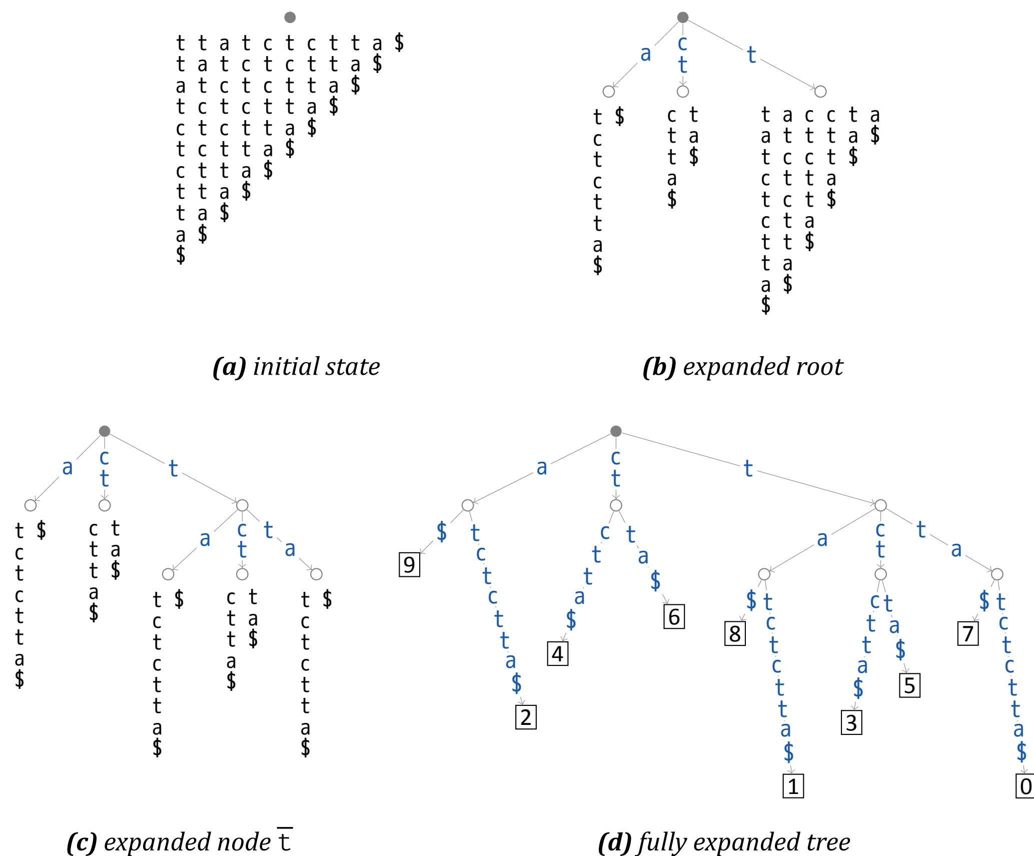
4.2 Lazy construction and representation

A key property of the wotd-algorithm is that it constructs the suffix tree top-down and nodes from disjunctive subtrees can be expanded independently and in arbitrary order. That makes it possible to step-by-step expand single nodes instead of entire subtrees and allows turning the suffix tree construction into a lazy, on-demand construction. Such a

Algorithm 4.2: CREATESUFFIXTREE(s)

input : text string s over the alphabet Σ
output : suffix tree of s

1 create tree T that consists of only the root node $\overline{\epsilon}$
2 WOTDEAGER($T, \overline{\epsilon}$)
3 return T



***(a)** initial state* ***(b)** expanded root*

***(c)** expanded node* $\overline{\mathtt{t}}$ ***(d)** fully expanded tree*

***Figure 4.1:** Different states of the lazy suffix tree for s = `ttatctctta`. Below each unexpanded node $\overline{\alpha}$ the remaining suffixes $R(\alpha)$ are shown without their common α-prefix. In the beginning (a), the lazy suffix tree consists of only the unexpanded root node. (b) shows the result of the root node expansion and (c) the expansion of node* $\overline{\mathtt{t}}$. *The fully expanded suffix tree is shown in (d).*

lazy suffix tree requires a method to expand a suffix tree node and a data structure to represent a partial suffix tree whose nodes are either in *expanded* or *unexpanded* state. Further, it requires $R(\alpha)$ for the expansion of nodes $\overline{\alpha}$ and needs to provide the corre-

sponding set of suffix start positions for all (even expanded) nodes $\overline{\alpha}$:

$$l(\alpha) := \{\, i \in [0..n) \;\big|\; \exists_{\beta \in \Sigma^*}\, s_i\$ = \alpha\beta \,\} \tag{4.2}$$

to determine the text occurrences of a pattern. Giegerich *et al.* proposed a lazy suffix tree that meets all of these requirements. However, their approach has two drawbacks: the children of inner nodes are not lexicographically ordered as in the enhanced suffix array based suffix tree and the tree cannot directly be generalized to multiple sequences. We introduce a new data structure that overcomes these limitations and present their and our approaches in the next two sections.

4.2.1 The original data structure

As mentioned above, in the original lazy suffix tree [Giegerich *et al.*, 2003] the children $\overline{\alpha\beta}$ of an inner node $\overline{\alpha}$ are not in lexicographically order, instead they are ordered increasingly by $\min l(\alpha\beta)$, i.e. decreasingly by the length of the longest suffix in $R(\alpha\beta)$. This order is well defined as the children $\overline{\alpha\beta_1}, \overline{\alpha\beta_2}, \dots, \overline{\alpha\beta_m}$ of $\overline{\alpha}$ partition the set $l(\alpha)$ into non-empty, disjoint sets $l(\alpha\beta_1), l(\alpha\beta_2), \dots, l(\alpha\beta_m)$.

We first describe how to represent edge labels. Consider an edge from the expanded node $\overline{\alpha}$ to a child $\overline{\alpha\beta}$. As $l(\alpha\beta)$ is the set of occurrence begin positions of $\alpha\beta$, it holds that $\beta = s[\min l(\alpha\beta) + |\alpha|..\min l(\alpha\beta) + |\alpha\beta|]$. Let lp be a function on tree nodes defined as:

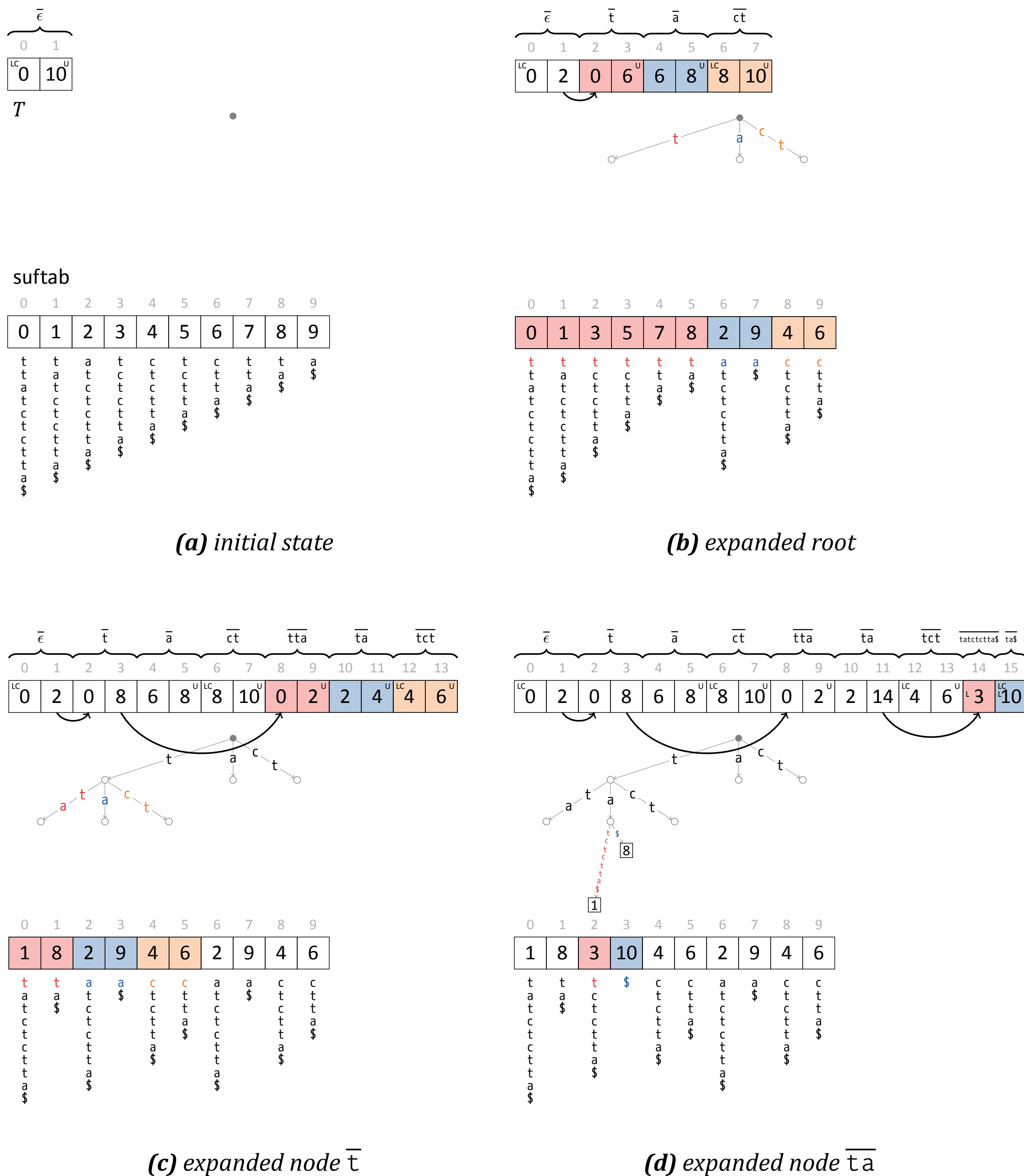
$$lp(\overline{\alpha\beta}) := \min l(\alpha\beta) + |\alpha|, \quad \text{where } \overline{\alpha} \text{ is parent of } \overline{\alpha\beta}. \tag{4.3}$$

We then can substitute $\beta = s[lp(\overline{\alpha\beta})..lp(\overline{\alpha\beta}) + |\beta|)$. For now, assume that the tree representation stores lp-values for all children of an expanded node. It remains to show how to determine $|\beta|$. In case $\overline{\alpha\beta}$ is a leaf in the suffix tree, $\alpha\beta$ is a suffix of $s\$$ and it holds $|\beta| = n + 1 - lp(\overline{\alpha\beta})$. Otherwise, assume that $\overline{\alpha\beta}$ is expanded and let $\overline{\alpha\beta\gamma_1}$ be the first child of $\overline{\alpha\beta}$. By definition of the child order it holds $\min l(\alpha\beta) = \min l(\alpha\beta\gamma_1)$ and hence $|\beta| = lp(\overline{\alpha\beta\gamma_1}) - lp(\overline{\alpha\beta})$. If $\overline{\alpha\beta}$ is unexpanded, $|\beta|$ can be computed via $|\beta| = |\operatorname{lcp} R(\alpha\beta)| - |\alpha|$.

The nodes of the partial suffix tree are stored in a string T of integers, where the children of a node are stored in a contiguous block and in the same order as in the tree. An expanded inner node $\overline{\alpha}$ is represented by two adjacent entries, $lp(\overline{\alpha})$ and $firstchild(\overline{\alpha})$. The latter refers to the beginning of the block of child nodes in T. A leaf is represented by a single entry in T the value $lp(\overline{\alpha})$. To distinguish between inner nodes and leaves, a *leaf bit* (L) is split off the first entry. The last child of a node is marked by a *last-child bit* (LC) in the first entry.

Node expansion

Unexpanded nodes are marked by an *unexpanded bit* (U) in the second entry. To expand a node $\overline{\alpha}$, the suffixes $R(\alpha)$ are partitioned according to their character at position $|\alpha|$. To this end, we store the corresponding suffix start positions in an additional integer string suftab of length n initialized with $0, 1, \dots, n-1$. In T the two reserved entries of every



***(a)** initial state*

***(b)** expanded root*

***(c)** expanded node* $\overline{\texttt{t}}$

***(d)** expanded node* $\overline{\texttt{ta}}$

Figure 4.2: *Different states of the lazy suffix tree for* $s = \texttt{ttatctctta}$ *and how they are represented by the original data structure. Below each unexpanded node* $\overline{\alpha}$ *the remaining suffixes* $R(\alpha)$ *are shown without their common* α*-prefix. In the beginning (a), the lazy suffix tree consists of only the unexpanded root node. (b) shows the result of the root node expansion, (c) and (d) the expansions of nodes* $\overline{\texttt{t}}$ *and* $\overline{\texttt{ta}}$

unexpanded node store boundaries i, j of substrings of suftab such that the following invariant holds. The intervals $[i..j)$ are disjoint subsets of $[0..n)$ and for an unexpanded node $\overline{\alpha\beta}$, suftab$[i..j)$ contains the values $l(\alpha\beta) + |\alpha|$ in increasing order. Hence the *lp*-value of $\overline{\alpha\beta}$ equals suftab$[i]$ and is therefore also available for unexpanded nodes. In the beginning, the partial suffix tree consists of only the root node represented by two entries $0, n$ in T with *leaf* and *unexpanded bits* set, see Figure 4.2a. Before expanding $\overline{\alpha\beta}$, the length of $|\beta|$ is unknown and must be determined by computing the lcp value lcp$\{s_k \mid k \in$ suftab$[i..j)\}$. This can be done by step-wise comparing all characters $s[\text{suftab}[i..j) + l]$ for $l = 1, 2, \ldots$ for equality. Then $|\beta|$ equals the smallest value l for which $a, b \in [i..j)$ exist with $s[\text{suftab}[a] + l] \neq s[\text{suftab}[b] + l]$. The values suftab$[i..j)$ are then increased by $|\beta|$ and stably rearranged into subintervals G_x of the same character x, such that for $x \in \Sigma$ holds $\forall_{k \in G_x} s[\text{suftab}[k]] = x$ and $\dot{\bigcup}_{x \in \Sigma \cup \{\$\}} G_x = [i..j)$. The groups correspond to the children of $\overline{\alpha\beta}$ and are appended to T in *lp*-order, i.e. increasingly by the value $\min_{k \in G_x}$ suftab$[k]$. For each singleton group, a single *lp*-entry with a set *leaf bit* is stored. The remaining groups are branching nodes whose subinterval boundaries are stored and marked with a set *unexpanded bit*. The last group is marked by setting the *last-child bit*. Finally, the *unexpanded bit* of the parent node is cleared and the two interval boundaries are replaced by $lp(\overline{\alpha\beta})$ and *firstchild*$(\overline{\alpha\beta})$, the position of the first child group appended to T.

Example 4.1. As an example, Figure 4.2 shows different states of the lazy suffix tree of $s =$ `ttatctctta` during the search of the pattern $p =$ `ttat`. Initially empty, the lazy suffix tree is expanded node-by-node along the whole search path. The subfigures (a)–(d) show the contents of T and suftab and the corresponding partial suffix trees. The different colors represent the different suffix groups computed when expanding a node. It can easily be seen that the groups are decreasingly sorted by the length of the longest suffix and that suftab is not a permuted suffix array.

The theoretical running time for constructing the whole suffix tree is $\mathcal{O}(n^2 + |\Sigma|)$ for the worst case and $\mathcal{O}(n \log_{|\Sigma|} n + |\Sigma|)$ on average [Giegerich and Kurtz, 1995]. In practice, the algorithm shows almost a linear running time and benefits from its good cache locality during the recursive descent [Giegerich *et al.*, 2003]. Giegerich *et al.* use a modified counting sort [Cormen *et al.*, 2001] that avoids iterations over the alphabet and reuses the counter array to group m suffixes in $\mathcal{O}(m)$ instead of $\mathcal{O}(|\Sigma| + m)$ time. As a result, the size of the alphabet is an addend instead of a factor in the overall running time and suffixes are stably assigned to groups with increasing *lp*-values instead of increasing group characters.

4.2.2 Our data structure

As explained above, Giegerich *et al.* use a modified counting sort with the effect that the outgoing edges of their lazy suffix tree are not in lexicographical order. However, some applications require a lexicographical order, e.g. to speed up the search for an outgoing edge from $\mathcal{O}(|\Sigma|)$ to $\mathcal{O}(\log |\Sigma|)$[1] or to search common edge labels between two suffix trees,

[1] A binary search on the outgoing edges requires an extra bit to distinguish the both entries of an inner node from the single entry of a leaf in T.

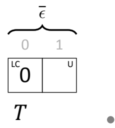
e.g. in a multiple exact pattern search described in Section 4.3.3. As a remedy, we first implemented a variant of the original lazy suffix tree where the suffixes are sorted by the character following the longest common prefix using an unmodified counting sort. However, for the edge label determination the group containing the longest suffix of all groups must be moved to front and become the first child in T. This variant makes it possible to enumerate the lexicographically ordered children of a node in linear time and to search an outgoing edge in logarithmic time.

In the following, we propose a second, more elegant variant of the lazy suffix tree where children can be stored in arbitrary and especially lexicographical order. Our data structure is applicable to multiple sequences and represents exactly the same generalized suffix tree as the enhanced suffix array, i.e. every tree node $\overline{\alpha}$ corresponds to the same substring of suftab and is stored in the same order as in the enhanced suffix array. The number of occurrences can be determined in $\mathcal{O}(1)$ time, whereas in the original data structure a DFS traversal is required.

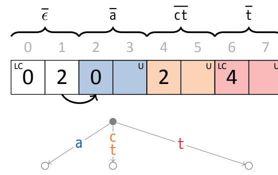
Again, our lazy suffix tree data structure consists of the two strings T and suftab and nodes require the same number of entries in T. The second entry of every expanded node is the *firstchild* pointer and extra bits are used to mark leaves, last children and unexpanded nodes. Every tree node $\overline{\alpha}$ corresponds to an interval $[i..j)$ such that $\mathsf{suftab}[i..j)$ now stores $l(\alpha)$, the set of positions where α occurs in the text. In contrast to the original lazy suffix tree [Giegerich *et al.*, 2003] which stores lp-values, our data structure stores lcp values, i.e. the length $|\alpha| = T[\mathit{firstchild}(\overline{\alpha})]$ is available for every expanded node $\overline{\alpha}$. For unexpanded nodes the lcp value can be computed as described in the previous section. Consider an edge from an expanded node $\overline{\alpha}$ to a child $\overline{\alpha\beta}$ and let $\mathsf{suftab}[i..j)$ be the substring that stores $l(\alpha\beta)$. The edge label β then can be determined via $\beta = s[\mathsf{suftab}[i] + |\alpha|..\mathsf{suftab}[i] + |\alpha\beta|)$.

Node expansion

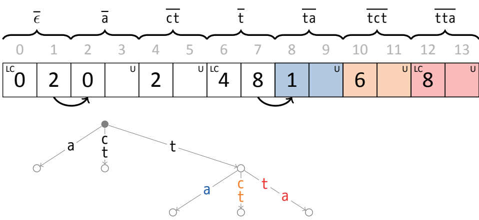
Consider an unexpanded node $\overline{\alpha}$ whose text occurrences are stored in $\mathsf{suftab}[i..j)$ in ascending order. To expand $\overline{\alpha}$, the elements $k \in \mathsf{suftab}[i..j)$ are stably sorted by $s\,[k + |\alpha|]$ using counting sort. The result is a partition $[l_0..l_1), [l_1..l_2), \dots, [l_{m-1}..l_m)$ with $l_0 = i$ and $l_m = j$, such that $\mathsf{suftab}[l_0..l_1), \mathsf{suftab}[l_1..l_2), \dots, \mathsf{suftab}[l_{m-1}..l_m)$ store the occurrences of the substrings $\alpha x_1 < \alpha x_2 < \dots < \alpha x_m$ in the text $s\$$ in increasing order, with $x_i \in \Sigma \cup \{\$\}$. Each interval stores the start positions of suffixes $R(\alpha x_i)$ whose longest common prefix $\alpha\beta_i = \mathrm{lcp}\, R(\alpha x_i)$ corresponds to a child node $\overline{\alpha\beta_i}$ of $\overline{\alpha}$ in the suffix tree and is inserted into our lazy suffix tree in the same order. Singleton intervals $[l_i..l_i + 1)$ correspond to suffix tree leaves. They occupy a single entry in T with value l_i and the *leaf bit* set. All other intervals $[l_i..l_{i+1})$ correspond to inner suffix tree nodes and occupy two entries with the *unexpanded bit* set. One entry is the left boundary l_i, the other is reserved for the $\mathit{firstchild}(\overline{\alpha\beta_i})$ pointer which is used if $\overline{\alpha\beta_i}$ is in expanded state. The right boundary needs not to be stored as it equals the left boundary of the following sibling or the right boundary of the parent node if $i = m - 1$. After all child nodes are inserted into T, the *last-child bit* is set for $\overline{\alpha\beta_m}$, the $\mathit{firstchild}(\overline{\alpha})$-pointer is adapted and length $|\alpha|$ is stored in place of the left boundary of the first child, which is expendable as it equals the left



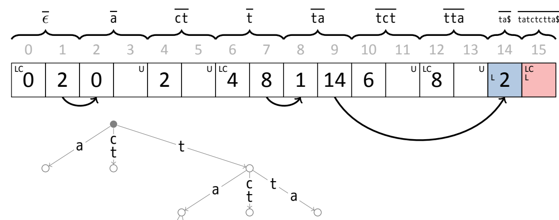
***(a)** initial state*



***(b)** expanded root*



***(c)** expanded node* $\overline{\texttt{t}}$



***(d)** expanded node* $\overline{\texttt{ta}}$

***Figure 4.3:** Different states of the lazy suffix tree and how they are represented by our data structure.*

boundary of the parent node.

Example 4.2. Figure 4.3 shows how the different states of the lazy suffix tree in Figure 4.2 are represented by our data structure. The same pattern $p = \texttt{ttat}$ is searched in the lazy suffix tree of $s = \texttt{ttatctctta}$. In contrast to the original lazy suffix tree [Giegerich *et al.*, 2003], in our data structure groups are sorted lexicographically and suftab is a permuted suffix array. Moreover, for every node $\overline{\alpha}$ there is a corresponding substring suftab$[i..j)$ that stores the occurrences of α in the text. Some of the entries in T are empty. They are reserved for *firstchild* pointers of yet unexpanded nodes, e.g. entry 1 initially empty and after expansion refers to the first child stored at position 2. For an expanded node $\overline{\alpha}$, $T[\mathit{firstchild}(\overline{\alpha})]$ stores the length $|\alpha|$, e.g. entry $T[14]$ stores 2 the length of the concatenation string of node $\overline{\texttt{ta}}$.

The whole lazy suffix tree can be constructed in $\mathcal{O}(n^2 + |\Sigma|n)$ time in the worst and $\mathcal{O}(n \log_{|\Sigma|} n + |\Sigma|n)$ time in the average case. The additional $\mathcal{O}(|\Sigma|n)$ running time is a result of using the original counting sort which sorts m suffixes by their first character in $\mathcal{O}(|\Sigma| + m)$ time.

4.2.3 Extension to multiple sequences

Our approach, described in the previous subsection, can easily be extended to multiple sequences. Given a set $\mathcal{S} = \{s^1, \dots, s^m\}$ of sequences of lengths $n^1, \dots, n^m$, a suffix is represented by a pair of integers (i, j), with $i \in [1..m]$ and $j \in [0..n^i)$. Hence we change suftab to be a string of pairs of length $n = \Sigma_{i \in [1..m]} n^i$ and initialize it with:

$$\mathsf{suftab} := (1,0)(1,1)\dots(1,n^1-1)(2,0)\dots(2,n^2-1)(3,0)\dots(m,n^m-1). \qquad (4.4)$$

T needs not to be changed, as all of its entries solely store prefix lengths, interval boundaries or child pointers. In order to construct a lazy suffix tree, whose nodes have the same order as in the ESA based suffix tree, the sentinel relation $\$^1 < \dots < \$^m < \min \Sigma$ must be retained. In order to achieve this without introducing extra alphabet characters in the implementation, we modified counting sort to use an extra bucket in front of all other buckets that represents all sentinels. When sorting the suffixes $R(\alpha)$ by their character at position $|\alpha|$, this sentinel bucket contains all pairs (i, j) with $j + |\alpha| = n^i$. As counting sort is stable, these pairs will have the same relation as in the initialization, i.e. in suftab a suffix $\alpha\$^i$ will be stored left of a suffix $\alpha\$^j$ with $i < j$. At last these suffixes are appended to T as leaves below $\overline{\alpha}$. They are appended in the same order as they occur in suftab and left of the remaining buckets.

4.3 Applications

In the following, we show that our lazy suffix tree is a complete replacement for the enhanced suffix array as it creates the same suffix tree and provides the same interface for traversing and accessing it. In general, lazy suffix trees are well suited for applications

where either only the upper parts of the suffix tree are traversed, e.g. if only substrings up to a certain length should be examined, or only certain paths to the leaves are traversed, e.g. searching patterns. For bottom-up traversals the enhanced suffix array is a better solution. In the following, we show different examples where a lazy suffix tree is well suited, and show in chapter 7 a new approach to frequency based string mining that exploits another property of our lazy suffix tree.

4.3.1 Traversing and accessing the lazy suffix tree

To make the lazy suffix tree easily accessible, we implemented a *top-down* and a *top-down history* suffix tree iterator for our data structure. Both iterators provide exactly the same functionality as the *top-down* and a *top-down history iterators* introduced in Section 3.6.2. The latter is also simply a subclass of the first extended by a stack that permits to go up the tree. *Top-down iterators* start at the root node and can arbitrarily go down and go right. If they are moved down from a node in unexpanded state the node is automatically expanded, which makes them a complete replacement for enhanced suffix array iterators. We also implemented variants that work on the original lazy suffix tree for the sake of completeness, but refer the reader to [Giegerich *et al.*, 2003] for details on how they are implemented.

A *top-down iterator* consists of member variables $node$, lb, rb, and $parentRb$. Let $\overline{\alpha}$ be the node the iterator points at. The position of the first of the at most two corresponding entries in T is stored by $node$. Analogously to the enhanced suffix array iterators, lb and rb are the boundaries such that suftab$[lb..rb)$ contains the start positions of suffixes that have a prefix α (our data structure) or the start positions of the remaining suffixes (original data structure). The right boundary of the parent node is stored in $parentRb$. Algorithm 4.3 shows how the iterator is initialized to point at the root node. As described above, 3 bits are required to signal whether a node is a leaf (L), in unexpanded state (U) or the last child (LC). We therefore use the 2 most significant bits of the first entry in T to store the *last-child* and *leaf bits* and store the *unexpanded bit* in the second entry. The remaining bits in T are used to store integers. In the following, we only describe the details of the functions goDown, goRight, and repLength for our lazy suffix tree variant. All other functions have been implemented as described in Sections 3.6.2 and 3.6.3.

Algorithm 4.6 shows how to move the iterator to the next sibling, which is only possible for all but the rightmost sibling, i.e. for $rb \neq parentRb$. The siblings are alphabetically arranged in T and in suftab. Hence, the sibling's left boundary in suftab equals the right boundary of the current node. Depending on whether the current node is a leaf or a branching node it occupies one or two entries in T. Consequently, $node$ must be increased accordingly. Finally, the right boundary is updated and becomes $lb + 1$ if the sibling is a leaf, $parentRb$ if the sibling is the last child, or the left boundary of the following sibling which is stored in $T[node + 2]$ otherwise.

How to move the iterator to the leftmost child of the current node is shown in Algorithm 4.5. First, the *leaf bit* is verified to be cleared, otherwise the node would have no child. A set *unexpanded bit* in the second entry signals that the node must be expanded first, which is done automatically. Now, the $node$ variable can be updated with the value

Algorithm 4.3: GoRoot($iter$)

```
input   : lazy suffix tree iterator iter
1  iter.lb ← 0
2  iter.rb ← |suftab|
3  iter.parentRb ← |suftab|
4  iter.node ← 0                // position in T
5  LC ← U ← w − 1              // w is the number of
6  L ← w − 2                   // bits of an entry in T
```

Algorithm 4.4: updateRb($iter$)

```
input   : lazy suffix tree iterator iter
1  if T[iter.node] & 2^LC ≠ 0 then
2      iter.rb ← iter.parentRb
3  else if T[iter.node] & 2^L ≠ 0 then
4      iter.rb ← iter.lb + 1
5  else
6      iter.rb ← T[iter.node + 2] & (2^L − 1)
```

Algorithm 4.5: goDown($iter$)

```
input   : lazy suffix tree iterator iter
output  : returns true on success
1  if T[iter.node] & 2^L ≠ 0 then
2      return false             // leaf bit is set
3  if T[iter.node + 1] & 2^U ≠ 0 then
4      expandNode(iter.node);
5  iter.node ← T[iter.node + 1] & (2^L − 1)
6  tmp ← iter.rb
7  updateRb(iter)
8  iter.parentRb ← tmp
9  return true
```

Algorithm 4.6: goRight($iter$)

```
input   : lazy suffix tree iterator iter
output  : returns true on success
1  if iter.rb = iter.parentRb then
2      return false
3  iter.lb ← iter.rb
4  if T[iter.node] & 2^L ≠ 0 then
5      iter.node ← iter.node + 1
6  else
7      iter.node ← iter.node + 2
8  updateRb(iter)
9  return true
```

of the node's second entry, which refers to the first entry of the leftmost child node in T. The right boundary rb is updated as in goRight and $parentRb$ gets its former value.

In order to determine the length of the concatenation string (without sentinel) for the current node, we implemented the function repLength as shown in Algorithm 4.7. It first examines whether the current node is a leaf and, if so, returns the length of the suffix. Otherwise, it either computes the length, as described in Section 4.2.2, if the node is unexpanded, or returns the length stored at the first child of the current node.

With the lazy suffix tree adaptation of repLength, the concatenation string and its occurrences in the text can be determined by representative (Algorithm 3.22) and getOccurrences (Algorithm 3.24) as described in Section 3.6.3 on page 57.

Algorithm 4.7: repLength($iter$)

```
input   : lazy suffix tree iterator iter
output  : returns the length of the concatenation string (without sentinel)
1  if T[iter.node] & 2^L ≠ 0 then
2      return n − suftab[iter.lb]
3  else if T[iter.node + 1] & 2^U ≠ 0 then
4      return computeLcp(iter.node)
5  else
6      return T[T[iter.node + 1]] & (2^L − 1)
```

4.3.2 Radix trees

The linear-time construction algorithms of suffix array and lcp table described in the previous chapter save redundant character comparisons by reusing rank or lcp information of neighboring suffixes. This is only possible if *all* suffixes are included in the computation. In contrast to this, the wotd algorithm does not exploit the fact that the strings to sort are overlapping suffixes of the same text. The suffix tree is constructed by solely comparing string characters. That makes it possible not only to expand nodes in arbitrary order but also to construct the radix tree of arbitrary, not necessarily all suffixes.

For a given set of strings, a radix tree, also known as patricia trie [Morrison, 1968], is a tree whose paths to the leaves represent the given strings. This is in contrast to a suffix tree which for a given strings represents *all* of their suffixes. A suffix tree therefore is a patricia tree of all suffixes.

Definition 4.1 (radix tree). The radix tree $RT(\mathcal{S})$ of a set of strings $\mathcal{S} = \{s^1, \dots, s^m\} \subseteq \Psi^*$ is a rooted tree whose edges are labeled with strings over $\Sigma := \Psi \cup \{\$^1, \dots, \$^m\}$, where $\$^j$ is a sentinel character with $\$^j \notin \Psi$ and $\$^1 < \dots < \$^m < \min \Psi$. The radix tree fulfills the following properties:

1. Each internal node is *branching*, i.e. it has at least two children.
2. For branching nodes the labels of outgoing edges begin with distinct characters.
3. The radix tree has m leaves numbered from 1 to m. The concatenation of edge labels from the root to leaf i yields the string $s^i\$^i$.

To create the radix tree of strings $s^1, \dots, s^m$ with the wotd algorithm, it suffices to change the way suftab is initialized. Instead of storing all the suffixes, we chose only the largest suffixes, i.e. the strings itself. Thus suftab must be a string of length m initialized with the pairs:

$$\mathsf{suftab} := (1,0)(2,0)\dots(m,0). \tag{4.5}$$

Because of the similarities between radix and suffix trees, the remaining parts of the algorithm can be kept unchanged and the radix tree can in the same way be accessed via suffix tree iterators.

4.3.3 Multiple exact pattern search

Given a text and a corresponding suffix tree, a single pattern p can be searched in the text by descending the tree along the path of edges labeled with the pattern sequence. If this is not possible, the text has no occurrence of p. Otherwise the descent ends in the topmost node v for which $p \preceq concat(v)$ holds. The occurrences of p can be determined by enumerating all leaves in the subtree of v.

A multiple pattern search is typically conducted by either searching each pattern separately in the suffix tree of the text [Weiner, 1973] or streaming the text against a keyword tree of patterns [Aho and Corasick, 1975], which is an uncompacted radix tree where edge labels are single characters and inner nodes may be non-branching.

Algorithm 4.8: EXACTMULTIRECURSION($iterA, iterB, i$)

```
   input   : iterator iterA of pattern radix tree
   input   : iterator iterB of text suffix tree
   input   : length i of compared prefixes
   output  : all text occurrences of a pattern
   // Part I: Go down a path that is not branching in both trees
1  α ← REPRESENTATIVE(iterA)
2  β ← REPRESENTATIVE(iterB)
3  repeat
4      j ← min{|α|, |β|}
5      if α[i..j) ≠ β[i..j) then return
6      if j < |β| then                                  // end of edge in radix tree?
7          if ISLEAF(iterA) then                        // patterns are leaves
8              print "pattern " α " found at: " GETOCCURRENCES(iterB)
9              return
10         if not GODOWN(iterA, β[j]) then return       // follow the suffix tree path
11         α ← REPRESENTATIVE(iterA)
12     else if j < |α| then                             // end of edge in suffix tree?
13         if not GODOWN(iterB, α[j]) then return       // follow the radix tree path
14         β ← REPRESENTATIVE(iterB)
15     i ← j
16 until |α| = |β|
   // Part II: Recursively go down paths branching in both trees
17 if not GODOWN(iterA) then
18     print "pattern " α " found at: " GETOCCURRENCES(iterB)
19     return
20 while REPLENGTH(iterA) = i do
21     print "pattern " REPRESENTATIVE(iterA) " found at: " GETOCCURRENCES(iterB)
22     if not GORIGHT(iterA) then return
23 GODOWN(iterB)
24 while REPLENGTH(iterB) = i do
25     if not GORIGHT(iterB) then return
26 while true do                                        // find pairs with equal edge beginnings
27     α ← REPRESENTATIVE(iterA)
28     β ← REPRESENTATIVE(iterB)
29     if α[i] < β[i] then
30         if not GORIGHT(iterA) then return
31     else if α[i] > β[i] then
32         if not GORIGHT(iterB) then return
33     else
34         EXACTMULTIRECURSION(iterA, iterB, i)         // recurse
35         if not (GORIGHT(iterA) and GORIGHT(iterB)) then return
```

In the following, we propose an algorithm that combines both approaches and top-down traverses the suffix tree of the text and the radix tree of the patterns in parallel. It recursively searches common paths in both trees and whenever a radix tree leaf is reached, a pattern was found whose occurrences are represented by the suffixes below the same path in the suffix tree. Beginning in the root nodes of both trees, the labels of edges to child nodes $(\overline{\alpha}, \overline{\beta})$ beginning with the same character are compared character-wise. If the lengths of the concatenation strings differ, e.g. $|\alpha| < |\beta|$, only the $|\alpha|$-prefix of β can be compared and if successful the comparison continues with the pair $(\overline{\alpha'}, \overline{\beta})$, where $\overline{\alpha'}$ is the child of $\overline{\alpha}$ whose edge begins with $\beta[|\alpha|]$. The recursion stops if such an edge does not exist or a mismatch occurs. Whenever the whole concatenation string of a radix leaf $\overline{\alpha}$ is compared successfully, the pattern α is found with occurrences given by the suffixes below $\overline{\beta}$.

Algorithm 4.8 shows the pseudo-code of our approach. It is called with radix and suffix tree iterators $iterA$ and $iterB$ pointing at the root nodes and $i = 0$ at the beginning. In the first part (lines 1–16) edge labels are compared, whereas the second part (lines 17–35) searches for pairs of children with the same edge label beginnings. This is possible by visiting each child at most once as the children are sorted by the first character of their edge label. The while-loops in lines 20–22 and 24–25 handle sentinel edges which are implemented to have an empty edge label instead of a $ character and appear left of all non-sentinel edges.

4.3.4 Approximate pattern search

In Section 2.5 on page 19, we described a DP algorithm that sequentially scans a text of length n to search a pattern p with up to k errors in $\mathcal{O}(kn)$ time on average. In typical applications, such as searching sequenced reads in a reference genome, the text is much larger than the pattern and even the linear search time becomes prohibitive. In the following, we propose a simple recursive search algorithm that descends the suffix tree of the text and solves the in $\mathcal{O}\big((2 \cdot |\Sigma| \cdot |p|)^k\big)$ time [Navarro and Baeza-Yates, 2000].

Searching a pattern with errors in a suffix tree requires to tolerate mismatches while descending along the path of pattern characters from the root towards the leaves. That means whenever a pattern character is compared with an edge character, a mismatch only reduces the remaining number of tolerated errors, see Algorithms 4.9 and 4.10 for the corresponding pseudo-code. Branching nodes must be left via the edge beginning with the current pattern character and, if there are errors remaining, also via all other edges. Approximate matches have been found if the end of the pattern has been reached without exceeding the number of tolerated errors.

The algorithmic idea of Algorithm 4.9 can be combined with the multiple pattern search algorithm of the previous section to approximately search multiple patterns in a text. We again use a radix tree of patterns and a suffix tree of the text and traverse both in parallel, starting at the root nodes. During the recursion, both concatenation strings are compared character-wise while recording the number of mismatches. When the end of one or both strings is reached, the search recurses into all children of the nodes or the Cartesian product of children sets of both nodes. The recursion ends if more mismatches

Algorithm 4.9: APPROXIMATERECURSION($pattern, iter, i, e$)

```
input  : search pattern, suffix tree iterator
input  : length i of compared prefixes of pattern and concatenation string
input  : remaining number of tolerated errors e
output : all text occurrences within tolerated Hamming distance

1  if e = 0 then
2      if goDown (iter, pattern[i..|pattern|]) then
3          print "pattern found at: " getOccurrences(iter)
4  else
5      while i < |pattern| and i < repLength(iter) do
6          if pattern[i] ≠ representative(iter)[i] then          // on mismatch ...
7              if e = 0 then                                 // reduce the tolerated errors
8                  return
9              else
10                 e ← e − 1
11         i ← i + 1
12     if i = |pattern| then
13         print "pattern found at: " getOccurrences(iter)
14     else
15         if not goDown(iter) then return
16         repeat                                            // at branching nodes
17             approximateRecursion(pattern, iter, i, e)    // try all outgoing edges
18         until not goRight(iter)
```

Algorithm 4.10: APPROXIMATEPATTERNSEARCH($pattern, errors$)

```
input  : pattern and number of tolerated Hamming errors
output : all approximate matches

1  create iterator iter of the suffix tree of the text
2  goRoot(iter)
3  approximateRecursion(pattern, iter, 0, errors)
```

occurred than tolerated or a leaf in either tree is reached. If it is a leaf in the radix tree a pattern has been found. Algorithm 4.11 shows the corresponding pseudo-code of this algorithm. The multiple exact pattern search algorithm of the previous section is used in line 2 as an optimization if no more errors are tolerated. The repeat-loops in lines 24 and 26 enumerate the children of the current nodes $\overline{\alpha}$ or $\overline{\beta}$ depending on whether the end of α or β has been reached in the comparison.

To evaluate the practical running time of the multiple approximate search algorithm, we searched 100,000 substrings in DNA, protein, and natural language texts of length 100 million characters while varying the allowed number of mismatches and the length of the substrings. The results are shown in Figure 4.4.

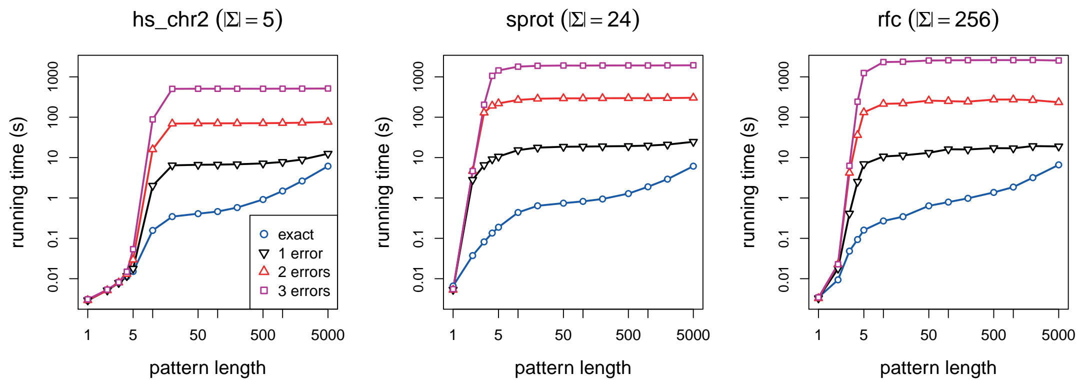


Figure 4.4: *Running times required to search 100,000 patterns with a varying number of tolerated errors in the first 100 M characters of a DNA, amino acid, and natural language text. We compared the exact (Section 4.3.3) and approximate (Section 4.3.4) recursive algorithms that search the radix tree of the patterns in the suffix tree (enhanced suffix array) of the text. Patterns are random substrings of varying length.*

It can be seen that the number of errors has the greatest influence on running time which increases by an order of magnitude for every additional error. The search time on large alphabets is higher than on small alphabets due to a greater out-degree of suffix tree nodes.

In [Siragusa *et al.*, 2013a,b], we demonstrate the applicability of the above-mentioned exact and approximate multiple backtracking approaches to the read mapping problem. In that work, we search exact or approximate occurrences of non-overlapping seeds of the reads in the reference sequence and extend them up to a given error rate.

The approximate search can also be extended to edit distance. Instead of comparing the edge labels of both trees character-wise, they need to be aligned recursively with a modified DP algorithm [Needleman and Wunsch, 1970] that updates a DP matrix which for a pair of tree nodes reflects the pairwise alignment of both concatenation strings. For more details, we refer the reader to [Navarro and Baeza-Yates, 2000].

Algorithm 4.11: APPROXIMATEMULTIRECURSION($iterA, iterB, i, e$)

```
   input  : iterator iterA of pattern radix tree
   input  : iterator iterB of text suffix tree
   input  : length i of compared prefixes
   input  : remaining number of tolerated errors e
   output : all text occurrences within tolerated Hamming distance

 1 if e = 0 then
 2     EXACTMULTIRECURSION(iterA, iterB, i)    // no errors left, use Algorithm 4.8
 3 else
 4     α ← REPRESENTATIVE(iterA)
 5     β ← REPRESENTATIVE(iterB)
 6     while i < |α| and i < |β| do
 7         if α[i] ≠ β[i] then                          // on mismatch ...
 8             if e = 0 then                        // reduce the tolerated errors
 9                 return
10             else
11                 e ← e − 1
12         i ← i + 1
13     if i = |α| then
14         if ISLEAF(iterA) then
15             print "pattern " α " found at: " GETOCCURRENCES(iterB)
16             return
17         GODOWN(iterA)
18     if i = |β| then
19         if not GODOWN(iterB) then return
20     repeat
21         iterB′ ← iterB
22         repeat
23             APPROXIMATEMULTIRECURSION(iterA, iterB, i, e)
24         until i ≠ |β| or not GORIGHT(iterB)
25         iterB ← iterB′
26     until i ≠ |α| or not GORIGHT(iterA)
```

Algorithm 4.12: APPROXIMATEMULTIPATTERNSEARCH($patterns, errors$)

```
   input  : multiple patterns and number of tolerated Hamming errors
   output : all approximate matches

 1 create pattern radix tree and tree iterator iterA
 2 create iterator iterB of the suffix tree of the text
 3 GOROOT(iterA), GOROOT(iterB)
 4 APPROXIMATERECURSION(iterA, iterB, 0, errors)
```

CHAPTER 5

q-gram Index

The two indices described in the previous chapters can be seen as alternative implementations of a suffix tree. However, many applications do not require the whole suffix tree functionality and instead only need to search patterns up to a certain length. For such case, a q-gram index may be a much faster alternative. We first define the term q-gram and how to construct and access a q-gram index. Finally, we explain how to use q-gram indices in q-gram counting filters to accelerate approximate string matching. We will later use one of the filters in our read mapper proposed in Chapter 6.

5.1 Definitions

A *q-gram* is a sequence of q characters over an alphabet Σ. The substring $s[i..i+q)$ of a given string s is called the *ungapped* or *contiguous* q-gram at position i in s. A generalization of contiguous q-grams are gapped q-grams [Burkhardt and Kärkkäinen, 2003] which are (non-contiguous) subsequences of s with length q and a certain shape.

Definition 5.1 (shape). A *shape* Q is a set of non-negative integers $\{i_1, i_2, \dots, i_q\}$ with $0 = i_1 < i_2 < \dots < i_q$. The *span* of Q is $\text{span}(Q) = i_q + 1$ and the *weight* of Q is the set cardinality $|Q| = q$.

The definition above includes ungapped q-grams as a special case where $Q = [0..q)$. Considering a given shape $Q = \{i_1, \dots, i_q\}$ the subsequence $s[i+i_1]s[i+i_2] \dots s[i+i_q]$ is called *(gapped)* q-gram at position i in s. In the following, we only consider q-grams that are true subsequences, i.e. $0 \le i \le n - \text{span}(Q)$. As a shortcut we denote the shape Q by a string over {-, #} of length $\text{span}(Q)$ where the character # only occurs at positions $p \in Q$, e.g. the shape $Q = \{0, 1, 4, 6\}$ corresponds to `##--#-#`. For example, the 3-grams of shape `##-#` in the string `gttca` are `gtc` and `tta`.

Consider the function $\text{rank} : \Sigma \to [0..|\Sigma|)$ that maps each character to its rank in the alphabet Σ, i.e. for each $a, b \in \Sigma$ holds $a < b \Rightarrow \text{rank}(a) < \text{rank}(b)$. We now want to extend rank to arbitrary q-grams and define the *q-gram code* to be the rank of the q-gram t in the set of all possible q-grams Σ^q in lexicographical order. The following function $\text{code} : \Sigma^q \to [0..|\Sigma|^m)$ computes the q-gram code value:

$$\text{code}(t) = \sum_{i=0}^{q-1} \text{rank}(t[i]) \cdot |\Sigma|^{q-1-i}. \tag{5.1}$$

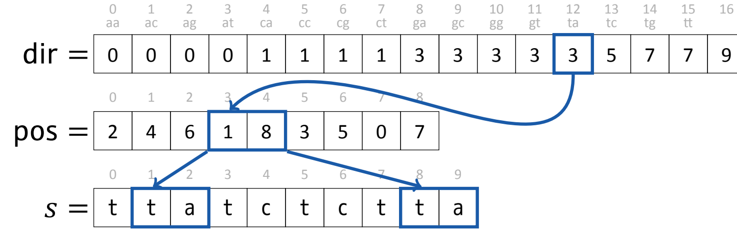


Figure 5.1: *Ungapped 2-gram index of* ttatctctta*. To look up all text occurrences of the 2-gram* ta*, we first determine its code value* code(ta) = 12*. The directory table* dir *stores at position 12 and 13 begin and end position of the substring* pos[3..5)*. This substring contains 1 and 8, the begin positions of all occurrences in the text.*

5.2 The direct addressing q-gram index

A (direct addressing) q-gram index is a data structure that permits looking up all text occurrences of a q-gram in a time linear to the number of occurrences.

For a string s of length n, the direct addressing q-gram index consists of two tables, the position table pos and the directory table dir. The position table stores all positions of gapped q-grams ordered by increasing code values.

Definition 5.2 (position table)**.** Given a shape $Q = \{i_1, \dots, i_q\}$ and a string s of length n, the position table pos of s is a string of length $n - \mathrm{span}(Q) + 1$ over the alphabet $[0..n - \mathrm{span}(Q)]$. For every $i, j \in [0..n - \mathrm{span}(Q)]$ holds:

$$i < j \Rightarrow \mathrm{code}(s[\mathrm{pos}[i]+i_1] \dots s[\mathrm{pos}[i]+i_q]) \leq \mathrm{code}(s[\mathrm{pos}[j]+i_1] \dots s[\mathrm{pos}[j]+i_q])\,. \quad (5.2)$$

As a consequence, the occurrences of each q-gram are stored in a contiguous interval in pos; this interval is called *q-gram bucket*. The directory table stores for each q-gram the start of its bucket which equals the number of occurrences of q-grams having a less code value.

Definition 5.3 (directory table)**.** Given a shape $Q = \{i_1, \dots, i_q\}$ and a string s of length n, the directory table dir of s is a string of length $|\Sigma|^q + 1$ over the alphabet $[0..n - \mathrm{span}(Q)]$. For every $i \in [0..|\Sigma|^q]$ holds:

$$\mathrm{dir}[i] = \left|\left\{ j \in [0..n - \mathrm{span}(Q)] \;\middle|\; \mathrm{code}(s[j + i_1] \dots s[j + i_q]) < i \right\}\right|. \qquad (5.3)$$

To determine the occurrences of a q-gram t we first need to compute the code $c := \mathrm{code}(t)$ in $\mathcal{O}(q)$ time. By definition, $m = \mathrm{dir}[c + 1] - \mathrm{dir}[c]$ equals the number of occurrences and $\mathrm{pos}[\mathrm{dir}[c]..\mathrm{dir}[c + 1])$ is the interval of occurrences. Thus, the m occurrences of any q-gram can be retrieved in optimal $\mathcal{O}(q + m)$ time.

5.3 Construction

The position table of a q-gram index for ungapped q-grams can be considered as a suffix array that is partially sorted. It contains all but the last $q - 1$ suffixes of the text sorted

by the first q characters. Hence, the position table can be constructed from a suffix array as it is proposed by Burkhardt *et al.* [1999]. Recording the lowest suffix ranks of runs of suffixes beginning with the same ungapped q-gram yields the directory table in a sequential scan over the position table. However, in practice constructing a suffix array is much more expensive in terms of running time and memory consumption than constructing only a partially sorted suffix array or constructing the q-gram index using non-comparison based sorting algorithms, e.g. counting sort or radix sort [Cormen *et al.*, 2001].

5.3.1 Counting sort algorithm

In SeqAn we implemented an adapted counting sort to construct the q-gram index in $\mathcal{O}(n + |\Sigma|^q)$ time and memory. Its pseudo-code is given in Algorithm 5.1. Note that our implementation uses the $(c+1)$-th bucket for counting (line 6) and increasing the target position (line 14) of q-grams with code value c. In this way, at the end of the algorithm $\text{dir}[c+1]$ contains the end position of the c-th bucket which equals the desired begin position of the $(c+1)$-th bucket.

Algorithm 5.1: CONSTRUCTQGRAMINDEX(s, Q)

input : text string s over the alphabet Σ, shape Q
output : position table pos, directory table dir

```
1  Let {i_1, ..., i_q} ← Q such that i_1 < ... < i_q
2  for j ← 0 to |Σ|^q do                                   // initialization
3      dir[j] ← 0
4  for j ← 0 to |s| − span(Q) do                           // count q-grams
5      c ← code(s[j + i_1]s[j + i_2] ... s[j + i_q])
6      dir[c + 1] ← dir[c + 1] + 1
7  sum ← max(0, |s| − span(Q) + 1)
8  for j ← |Σ|^q downto 0 do                               // cumulative sum
9      sum ← sum − dir[j]
10     dir[j] ← sum
11 for j ← 0 to |s| − span(Q) do                           // fill position table
12     c ← code(s[j + i_1]s[j + i_2] ... s[j + i_q])
13     pos[dir[c + 1]] ← j
14     dir[c + 1] ← dir[c + 1] + 1
15 return (pos, dir)
```

5.3.2 Extension to multiple sequences

Analogously to the definition of the generalized suffix array, the q-gram index of multiple sequences $s^1, s^2, \dots, s^m$ differs from the single sequence index only in that pos contains position pairs. The pair (i, j) with $i \in [1..m]$ and $j \in \left[0..|s^i| - \text{span}(Q)\right]$ represents a

gapped q-gram at position j in s^i. The staightforward extension of Algorithm 5.1 to multiple sequences is shown in Algorithm 5.2. Instead of scanning a single sequence, it counts q-grams and fills their begin positions into buckets in consecutive scans of the sequences.

Algorithm 5.2: CONSTRUCTQGRAMINDEX_MULTI($s^1, \dots, s^m, Q$)

```
   input   : multiple text strings s^1,...,s^m over the alphabet Σ, shape Q
   output  : position table pos, directory table dir

1  Let {i_1,...,i_q} ← Q such that i_1 < ... < i_q
2  for j ← 0 to |Σ|^q do                                   // initialization
3      dir[j] ← 0
4  for i ← 1 to m do                                       // count q-grams
5      for j ← 0 to |s^i| − span(Q) do
6          c ← code(s^i[j + i_1]s^i[j + i_2] ... s^i[j + i_q])
7          dir[c + 1] ← dir[c + 1] + 1
8  sum ← Σ_{i=1}^{m} max(0, |s^i| − span(Q) + 1)
9  for j ← |Σ|^q downto 0 do                               // cumulative sum
10     sum ← sum −dir[j]
11     dir[j] ← sum
12 for i ← 1 to m do                                       // fill position table
13     for j ← 0 to |s^i| − span(Q) do
14         c ← code(s^i[j + i_1]s^i[j + i_2] ... s^i[j + i_q])
15         pos[dir[c + 1]] ← (i, j)
16         dir[c + 1] ← dir[c + 1] + 1
17 return (pos, dir)
```

5.3.3 Adaptation to external memory

For large texts we provide an external memory variant of Algorithm 5.1. Algorithm 5.3 first scans the text and externally sorts pairs of code values and positions of all overlapping q-grams by increasing code values. The corresponding positions of the sorted array A are used to sequentially fill pos. The begin position of each bucket c is determined during a scan over A and sequentially written to dir while counting the number of q-grams with code values less than c.

5.4 The open addressing q-gram index

As described above, the directory table of the direct addressing q-gram index is a string of length $|\Sigma|^q + 1$ and the whole index consumes $\Theta(|\Sigma|^q + n)$ memory. Thus, for growing values of q the available memory rapidly becomes a limiting factor, e.g. for a DNA alphabet with $|\Sigma| = 4$, $q = 16$, and 4 byte per entry the directory requires 16 GB of memory. For alphabets with $|\Sigma| = 256$ the same amount of memory is required for $q = 4$. In fact,

Algorithm 5.3: CONSTRUCTQGRAMINDEX_EXTMEM(s, Q)

```
input   : text string s over the alphabet Σ, shape Q
output  : position table pos, directory table dir
                              // sort q-grams and store positions in pos
1  A ← ⟨(code(s[j + i_1]s[j + i_2] ... s[j + i_q]), j) | j ∈ [0..|s| − span(Q)]⟩
2  sort A by the first component
3  pos ← ⟨last component of a : a ∈ A⟩
                   // scan pos for buckets and store their begin positions in dir
4  dir ← ⟨⟩
5  b ← 0
6  c_last ← −1
7  foreach (c, j) ∈ A do
8      if c_last < c then
9          for i ← c_last + 1 to c do
10             append b to dir
11         c_last ← c
12     b ← b + 1
13 for i ← c_last + 1 to |Σ|^q do
14     append b to dir
```

the directory is only required to retrieve the position table interval of every *non-empty* q-gram bucket and to determine whether a bucket is empty. However, the number of non-empty buckets is not only bound by the number of possible different q-grams $|\Sigma|^q$ but also by the number of overlapping q-grams $n - \text{span}(Q) + 1$.

In the following, we propose the *open addressing q-gram index* with a memory consumption of $O(\alpha^{-1}n)$, for a fixed load factor α with $0 < \alpha \leq 1$. Instead of addressing entries in dir directly by q-gram code values, it uses an open addressing scheme [Cormen *et al.*, 2001] to map the q-gram codes (keys) of non-empty buckets to entries in dir (values). The load factor determines the maximal ratio between non-empty buckets and available entries of the q-gram directory and provides a trade-off between number of collisions and memory consumption.

In addition to the directory table dir, the open addressing index uses a string C of length $\lfloor \alpha^{-1}n \rfloor$ over the alphabet $[-1..|\Sigma|^q)$, the so-called *code table*. A pseudo-random hash function hash : $[0..|\Sigma|^q) \to [0..\lfloor \alpha^{-1}n \rfloor)$ maps a q-gram code c to an entry in dir. Our index implementation can be used with arbitrary hash functions and by default maps the q-gram code c to its CRC32 checksum using the SSE4.2 CPU instruction _mm_crc32_u64 [Intel, 2011].

As hash may map different values of c to the same position i (collision), $C[i]$ stores the code that currently occupies the entry dir$[i]$ or equals -1 if empty. Whenever a collision with a different code occurs, the entries $C\left[(i + j) \bmod \lfloor \alpha^{-1}n \rfloor\right]$ for $j = 1^2, 2^2, 3^2, \ldots$ are probed for being empty or containing the correct code. Compared to linear probing this

Algorithm 5.4: GETBKT(c, C)

```
input   : code value c, code table C
output  : position of the corresponding
          entry in dir
1 if |C| = 0 then return c
2 i ← hash(c) mod |C|
3 d ← 0      // search with quadratic probing
4 while C[i] ≠ c and C[i] ≠ −1 do
5     i ← (i + 2d + 1) mod |C|
6     d ← d + 1
7 return i
```

Algorithm 5.5: REQBKT(c, C)

```
input   : code value c, code table C
output  : position of the corresponding
          entry in dir
1 if |C| = 0 then return c
2 i ← GetBkt(c, C)
  // occupy bucket and return its position
3 C[i] ← c
4 return i
```

quadratic probe sequence prevents primary clustering of buckets [Cormen *et al.*, 2001]. The buckets in pos need to be arranged in the same order as their code values appear in C. In this way, dir$[i]$ stores the begin and dir$[i+1]$ the end position of the bucket in pos.

Algorithm 5.4 shows how to retrieve for a q-gram code c the position i of the entry in dir that stores the corresponding bucket begin position. Algorithm 5.5 returns the same position i or occupies a new entry if it not yet exists. The first lines of both algorithms optionally switch to the behavior of the direct addressing index if it is more memory efficient ($|\Sigma|^q < 2\alpha^{-1}n$). In that case C is empty as it is not needed. To construct the open addressing index we modified Algorithm 5.1 (and 5.2) by inserting $c \leftarrow$ REQBKT(c, C) between lines 5 and 6 (6 and 7) and $c \leftarrow$ GETBKT(c, C) between lines 12 and 13 (14 and 15). We made the same modification to Algorithm 5.6 by inserting $c \leftarrow$ GETBKT(c, C) between lines 1 and 2. Algorithm 5.3 can be modified as well if C and dir are stored in internal memory, otherwise an external hash table [Jensen and Pagh, 2008] would be preferable.

We compared the running time and memory required to construct direct addressing and open addressing q-gram indices of different DNA texts for different values of q. The results are shown in Figure 5.2. The construction times and memory consumptions are identical for $q \leq 8$ as the open addressing scheme is disabled since it would consume more memory than the direct addressing index. For $q = 14, 15$ the switch occurs at text lengths around 50 Mb or 200 Mb. It can be seen that the open addressing index is at least half as fast as the direct addressing index and for $q = 16$ on texts smaller than 100 Mb even faster.

5.5 Applications

The most common and simplest application of q-gram indices is to search the occurrences of a q-gram in a text. For a given q-gram t with code value $c = \text{code}(t)$, the begin positions of all text occurrences are stored in a bucket pos$[l..r)$ with $l = \text{dir}[c]$ and $r = \text{dir}[c+1]$ for the direct addressing index or $l = \text{dir}[\text{GETBKT}(c, C)]$ and $r = \text{dir}[\text{GETBKT}(c, C) + 1]$ for the open addressing index. Figure 5.1 shows how to look up the occurrences of the ungapped 2-gram `ta` in a direct addressing index of the text `ttatctctta` over the alphabet $\Sigma = \{\text{a}, \text{c}, \text{g}, \text{t}\}$. The general pseudo-code for searching a direct addressing index is given in Algorithm 5.6.

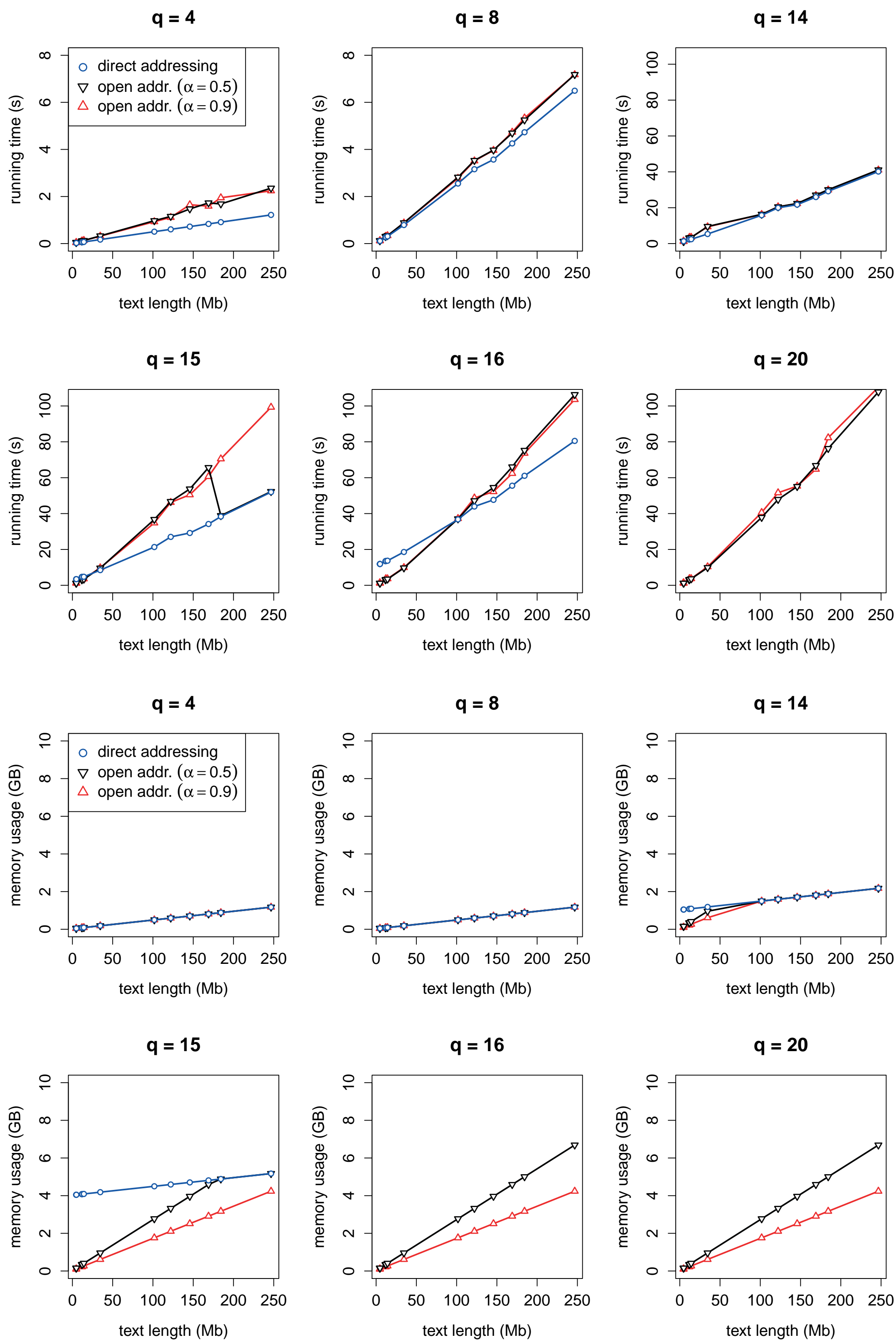


Figure 5.2: *Construction time (top) and memory consumption (bottom) of direct addressing and open addressing q-gram indices for different values of q and load factors α. We used the DNA datasets described in Table 3.6 on page 51.*

The q-gram index of multiple sequences can and has been successfully used to efficiently determine the number of common q-gram between pairs of sequences, e.g. in [Göke *et al.*, 2012] to replace costly pairwise alignments by q-gram based similarity measures or to build a guide-tree in a progressive sequence alignment as we proposed in [Rausch *et al.*, 2008].

In the following, we introduce two *filters* based on q-gram counting that accelerate local alignment algorithms by skipping parts of the text that contain no local matches. In Section 6.4.1 we propose a variant of the second filter that we specialized for the approximate matching problem and utilize in our read mapping tool RazerS.

Algorithm 5.6: GETOCCURRENCES(t, pos, dir)

```
input  : q-gram t, position table pos, directory table dir
output : text occurrences of t
1 c ← code(t)
2 occs ← ⟨⟩
3 for j ← dir[c] to dir[c + 1] − 1 do
4     append pos[j] to occs
5 return occs
```

5.5.1 q-gram counting filters for approximate matching

In Chapter 2.5 we introduced the approximate string matching problem which can be solved with the proposed dynamic programming algorithms or the recursive suffix tree search described in Chapter 4.3.4. However, the bilinear running time of the DP algorithm makes it infeasible to millions of patterns and texts of billions of characters. The recursive suffix tree search with a running time exponential in the number of errors is applicable to only a small numbers of errors, e.g. 1 or 2. A remedy to this situation are *filters* which make it possible to reduce the search space and thus the overall running time of approximate matching algorithms by orders of magnitude.

A *lossless* filter is an algorithm that speeds up the pattern search by discarding large parts of the text that are guaranteed *not* to contain an approximate match. The remaining parts, called *candidate regions*, are then examined using an approximate pattern matching algorithm. Commonly used filters are either *seed based* or based on *q-gram counting*. Single or multiple seed based filters define candidate regions to share a single or multiple gapped q-grams with the pattern. Filters based on q-gram counting, which we consider in the following, require a candidate region to have a certain number of q-grams in common with the pattern. Fundamental to most of the q-gram counting filters is the *q-gram lemma* which determines this number such that the filter is lossless:

Lemma 5.1 (q-gram lemma [Owolabi and McGregor, 1988; Jokinen and Ukkonen, 1991])**.** *Let $a, b \in \Sigma^n$ be two strings with Hamming distance k, then a and b share at least $t = n - \mathrm{span}(Q) + 1 - k \cdot q$ common q-grams.*

Proof. First assume $k = 0$, then a and b are equal and have $t = n - \text{span}(Q) + 1$ q-grams in common. As all q-grams have the same shape and each q-grams covers q characters, every introduced mismatch is covered by at most q formerly common q-grams. Hence, overall not more than kq q-grams can be destroyed by k errors and at least $t = n - \text{span}(Q) + 1 - k \cdot q$ common q-grams remain. ■

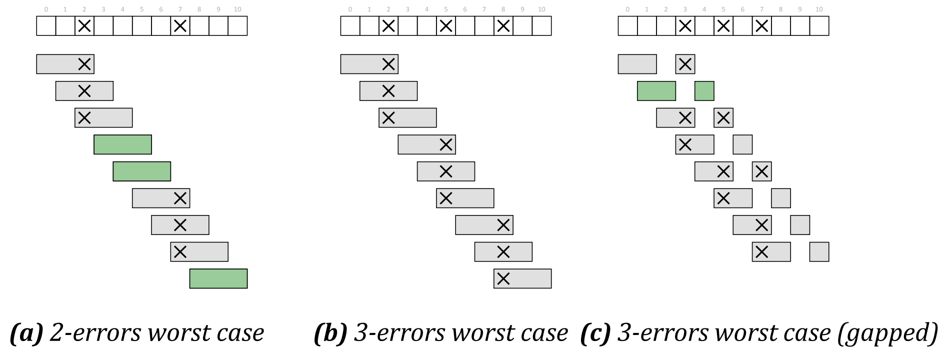
The threshold t is the minimal number of q-grams that two strings within Hamming distance k share. This lemma can also be applied to edit distance if n is the length of the larger sequence. The term common q-gram was intentionally not defined precisely as the lemma holds for different alternative definitions [Burkhardt and Kärkkäinen, 2003]. One way is to count the number of overlapping q-grams of one string that occurs at least once in the other. The q-gram lemma is strict for ungapped shapes and the worst case can be constructed by placing errors such that each error destroys q q-grams, e.g. at positions $q - 1, 2q - 1, 3q - 1, \ldots$, see Figures 5.3a and 5.3b. For gapped shapes the lemma gives a lower bound for the exact threshold and a strict closed formula has not been found yet. However, the exact threshold can be computed using dynamic programming as proposed by Burkhardt and Kärkkäinen [2003].

Although the threshold given by the lemma is lower for gapped shapes compared to ungapped shapes with the same weight, gapped shapes may yield a higher exact threshold. Even more, gapped shapes may increase the filtration specificity due to a higher *minimal coverage* [Burkhardt and Kärkkäinen, 2003]. For a shape Q and a threshold t the minimal coverage is the minimal number of characters that must match to observe t common q-grams. The higher the minimal coverage, the less likely random hits are and the more specific the filter is. However, most of the advantages of gapped shapes are limited to Hamming distance. Under edit distance gapped shapes have the same threshold as ungapped shapes with the same span. As q-gram gaps are only *immune* to mismatches, additional counting methods are required to also tolerate indels [Burkhardt and Kärkkäinen, 2002].

Example 5.1. Considering strings of length $n = 11$ with Hamming distance $k = 3$. The q-gram lemma for ungapped 3-grams gives an exact threshold of $t = 11 - 3 + 1 - 3 \cdot 3 = 0$, see Figure 5.3b. For the gapped shape ##-# a lower bound for the threshold would be $t = -1$. However, the exact threshold $t = 1$ is even higher as in every mismatch pattern at least one gapped 3-gram remains uncovered by mismatches. A worst case is shown in Figure 5.3c.

QUASAR

The first filter algorithm that utilizes the q-gram lemma was QUASAR [Burkhardt *et al.*, 1999]. It searches a pattern of length m with at most k edit errors in a text and uses the observation that the maximal length of a match is $m + k$. Considering the dot plot between pattern and text, the common q-grams between pattern and match must reside in a $m \times (m+k)$ rectangle. In order to detect $m \times (m+k)$ rectangles with a sufficiently high number of common q-grams, QUASAR virtually partitions the text into blocks of length



(a) 2-errors worst case *(b) 3-errors worst case* *(c) 3-errors worst case (gapped)*

Figure 5.3: *Mismatch patterns of length 11. (a) and (b) show worst cases of the q-gram lemma where a maximal number of ungapped 3-grams is destroyed. For 3 mismatches, a gapped shape (c) should be preferred as it yields a higher threshold.*

$2b$, with $b \geq m + k$, and additionally uses a second partitioning where blocks are shifted by b (see Figure 5.4a). In this way, blocks overlap by at least $m + k$ characters and each match completely resides in at least one block. A q-gram index[1] built over the text is used to look up the text occurrences of pattern q-grams. Each q-gram of the pattern increases the counters of blocks that contain at least one occurrence. At the end, only blocks whose counters are greater or equal t are output as candidates for containing a pattern match and are further examined using an approximate string matching algorithm. QUASAR can also be used to find local similarities by sliding a window of length w over the pattern while updating the q-gram counters.

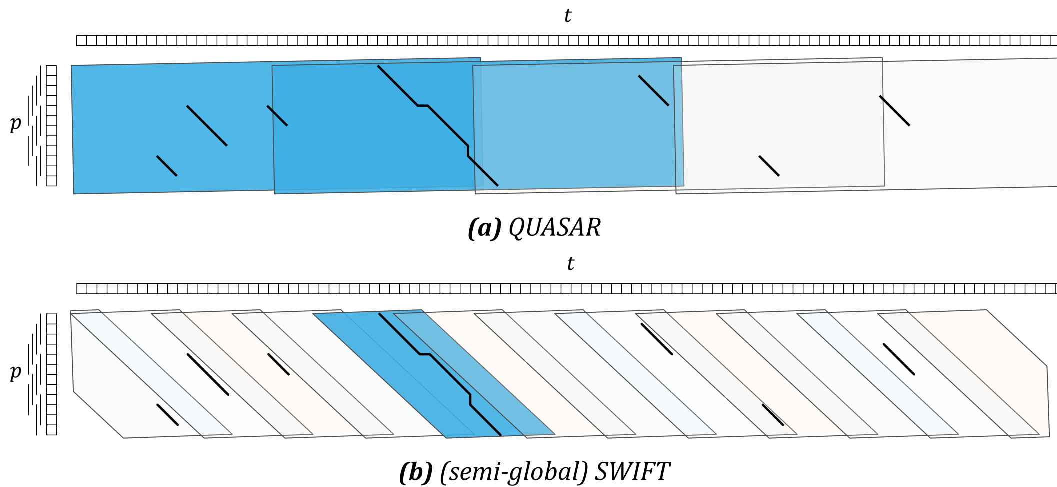
SWIFT

A direct successor of QUASAR is SWIFT [Rasmussen *et al.*, 2006], a filter for local alignments. In contrast to QUASAR which uses an absolute error threshold k and a fixed window length, it permits to specify an error rate ε and a minimum match length n_0 which is more appropriate since the length of a local alignment is not known in advance. An ε-match of a pattern p is a match of a substring $p' \prec p$ with at most $\lfloor \varepsilon |p'| \rfloor$ errors. SWIFT has a higher filtration specificity, as it counts common q-grams in partial dot plot parallelograms instead of rectangles, and guarantees to detect an overlapping parallelogram for each ε-match.

We specialized the original algorithm to find approximate (semi-global) matches instead of local matches and count common q-grams in full, not partial parallelograms (see Figure 5.4b). A more detailed description of our variant can be found in Section 6.4.1.

In the following, we describe the fundamental idea of the original SWIFT algorithm. Considering the dot plot between pattern and text, every common q-gram, called q-*hit*, corresponds to a diagonal stretch of matches with length q. Obviously, all q-hits of an alignment with k errors can cover at most $k + 1$ different diagonals in the dot plot. Using the q-gram lemma, Rasmussen *et al.* proved that for any given ε and n_0 there exist w, q, e

[1] QUASAR actually uses a suffix array. However, a q-gram index can be constructed much faster and is also applicable to gapped shapes.



***(a)** QUASAR*

***(b)** (semi-global) SWIFT*

Figure 5.4: *Filters based on q-gram counting. (a) shows how QUASAR partitions the dot plot into rectangles[a] such that each match fits into at least one of them. Our semi-global variant of SWIFT (b) counts q-grams in overlapping parallelograms of less area such that the trace of each match is covered by at least one. The example shows, that SWIFT is a more specific filter and for the true match reports a much smaller candidate region (blue area) compared to QUASAR.*

[a] Rectangles and parallelograms are intentionally rotated to visualize their overlap.

and τ such that every ε-match contains τ q-hits that reside in a $w \times e$ parallelogram. A $w \times e$ parallelogram is the intersection of $e + 1$ consecutive diagonals and $w + 1$ consecutive columns in the dot plot.

To detect $w \times e$ parallelograms with τ q-hits in the dot plot, the SWIFT algorithm slides from left to right over the pattern and searches each pattern q-gram in a q-gram index of the text. Found q-hits are counted in bins of $\Delta + e$ consecutive diagonals whose first diagonal is a multiple of Δ. As adjacent bins share e diagonals, every $w \times e$ parallelogram is contained in one bin. Every bin contains a q-hit counter and represents the parallelogram with columns bounded by the leftmost and rightmost contained q-hit. If a q-hit is found that is at most $w - q$ columns apart from the rightmost q-hit, the parallelogram is extended. Otherwise it is closed and a new one starting at the found q-hit is opened as the two hits cannot be part of the same $w \times e$ parallelogram. A closed parallelogram whose bin counter has reached τ is output as a *SWIFT hit* and needs to be verified whether it contains a part of an ε-match.

The ε-match verification is a non-trivial part and not covered by the work of Rasmussen *et al.* However, we proposed an algorithm that uses SWIFT for filtration and a new verification strategy to detect and report all ε-matches in [Kehr *et al.*, 2011]. To verify SWIFT hits, we first examine the candidate parallelogram as to whether it contains an ε-match using a local alignment algorithm with an appropriate scoring scheme. If it contains an ε-match, we extend the alignment, if possible, using an X-drop strategy [Zhang *et al.*, 1998] and report a maximal ε-match without X-drop. In a last phase we remove du-

plicate or overlapping matches which are the result of ε-matches that contain multiple shorter ε-submatches. For more details, we refer the reader to [Kehr *et al.*, 2011].

Part III

APPLICATIONS

Chapter 6

Read Mapping

In this chapter, we propose RazerS [Weese *et al.*, 2009, 2012], a versatile read mapper that allows to align millions of sequenced reads of arbitrary length under Hamming or edit distance. Our algorithm supports shared-memory parallelism and utilizes the q-gram index and q-gram based filters as proposed in Chapter 5, and a fast bit-parallel DP algorithm for approximate string search. We first describe the existing read mapping tools and their characteristics. Thereafter, we state a formal definition of the read mapping problem to solve and introduce the novel algorithmic ideas of RazerS. As such, we present an automatic parametrization approach that, given a user-defined loss rate, guarantees not to lose more reads than specified. This enables the user to run our tool either lossless or with a controlled loss rate at higher speeds and provides a seamless tradeoff between sensitivity and running time. At the end of this chapter, we extensively evaluate our approach on different real-world datasets in comparison with other state-of-the-art read mappers.

6.1 Related work

In the last years a variety of tools was designed and developed specifically for the purpose of mapping short reads. In Table 6.1 we compare the algorithmic key features of a subset of mappers, i.e. Bowtie [Langmead *et al.*, 2009], Bowtie 2 [Langmead and Salzberg, 2012], BWA [Li and Durbin, 2009], Eland [Cox, 2006], Hobbes [Ahmadi *et al.*, 2012], Maq [Li *et al.*, 2008a], mrFAST [Alkan *et al.*, 2009], SeqMap [Jiang and Wong, 2008], SHRiMP 2 [David *et al.*, 2011], Soap [Li *et al.*, 2008b], Soap 2 [Li *et al.*, 2009b], Zoom [Lin *et al.*, 2008], and RazerS [Weese *et al.*, 2009, 2012]. For a thorough algorithmic comparison we refer the reader to [Li and Homer, 2010; Fonseca *et al.*, 2012].

All existing read mapping approaches are based on a substring index. Depending on the type of index, they can be divided into two classes: 1) q-gram based or 2) prefix-trie based read mappers.

q-gram based read mappers. The q-gram based approaches use a two-step strategy. First, a filtration algorithm reduces the search space by filtering regions that cannot contain a match. This includes building a q-gram index, either on the set of reads, the reference sequence, or both to efficiently find common q-grams. The remaining *candidate regions* are then examined for true matches in a second, more time-consuming verification

		q-gram based									prefix trie based			
		Eland	Maq	Soap	SeqMap	Zoom	SHRiMP 2	RazerS	Hobbes	mrFAST	Bowtie	Bowtie 2	BWA	Soap 2
index	index of reads	•	•	-	•	•	-	•	-	•	-	-	-	-
	index of reference	-	-	•	-	-	•	-	•	•	•	•	•	•
filtering	seed-based	•	•	•	•	•	-	•	•	•		•		
	q-gram counting	-	-	-	-	-	•	•	-	-		-		
	supports indels	-	-	-	•	-	•	•	•	•		•		
	full sensitivity mode	•[a]	•[b]	-[c]	•[d]	•	-[e]	•	•	•		-		
	controllable loss rate	-	-	-	-	-	-	•	-	-		-		
mapping	supports indels	-	-	-	•	•[f]	•	•	•	•	-	•	•	-
	max. read length [bp]	32	127	60	-	240	-	-	100	1000	-	-	-	-
	max. error	2	3	2	5	-	-	-	-	-	-	-	-	2
	multiple (suboptimal) hits	-	-	-	•	•	•	•	•	•	•	•	•	-

Table 6.1: *Read mapping tools and their characteristics.*

[a] full sensitive up to 2 mismatches
[b] full sensitive up to 3 mismatches
[c] depends on settings, no switch to guarantees full sensitivity
[d] full sensitive up to 5 errors
[e] no help for parameter choice, default will be lossy for most settings
[f] limited to one gap

step. The filtration algorithms again can be divided into *seed-based* or *q-gram counting* filters.

Most of the seed-based filters are based on the pigeonhole principle. It states that, given two sequences within distance k, any partition of the first sequence into $k+1$ parts contains one part that can be found without errors in the other sequence [Navarro and Raffinot, 2002]. The shorter this *seed*, the more likely it is to encounter random matches, and therefore the lower the specificity of the filter. This strategy is thus rather limited to a small number of errors. To increase the seed length and the filtration specificity, Eland was the first to extend this strategy and divides one sequence into $k+2$ parts. Now at least two of such parts will occur in the other sequence. These two parts retain their relative positions as long as no indels occur in between. Eland, Maq, and Soap make use of this observation but are therefore limited to Hamming distance. Furthermore, Eland and Soap always use a 4-segment partition and Maq at most a 5-segment partition and can therefore not guarantee full sensitivity for $k > 2$ or $k > 3$, respectively. SeqMap extends the two-seed pigeonhole strategy to edit distance and searches the two parts varying the gap length by $-k, \dots, k$ nucleotides. Not based on the pigeonhole lemma but also seed-based filters, where a common (gapped) q-gram triggers a verification, are BLAT [Kent, 2002], PatternHunter [Ma *et al.*, 2002], PatternHunter II [Li *et al.*, 2003], and Zoom.

The second class of filters is based on q-gram counting and the minimal number t of

common q-grams every k-error match must have (see Lemma 5.1 on page 90). To find match candidates, SHRiMP 2 scans the reference and for every read of length m reports dot plot rectangles of size $m \times (m + k)$ with at least t common q-grams as candidates. For SHRiMP 2, however, q and t must be set manually and the default configuration is not guaranteed to be lossless.

Verification methods encompass approximate matching algorithms for Hamming or edit distance [Sellers, 1980], local-alignment algorithms [Smith and Waterman, 1981], e.g. in SHRiMP 2, or alignment algorithms that minimize the sum of base-call qualities at mismatching bases, e.g. in Maq. In current implementations one has to carefully distinguish whether both steps, the filtration and the verification step, are adequate for the distance chosen (Hamming or edit distance). Some implementations, e.g. Maq, verify matches using base-call qualities but filter the candidate regions using a fixed Hamming or edit distance.

Some q-gram based read mappers primarily target sensitivity and support to output all (including suboptimal) matches of a read up to a certain distance. This set of so-called *all-mappers* includes Hobbes, SeqMap, SHRiMP 2, Hobbes, mrFAST, Zoom, and RazerS.

Prefix-trie based mappers. Another approach to read mapping utilizes the *Burrows-Wheeler transform* [Burrows and Wheeler, 1994] which allows for the compression of a text and its suffix array suftab. Ferragina *et al.* [2004] complemented the functionality of this data structure so that one can count and locate all exact text occurrences of a pattern p without prior uncompressing either of the two tables. This self-index, also called *FM index*, uses a backward search to determine the interval $\mathtt{suftab}[l..r)$ of occurrences in $\mathcal{O}(|p|)$ time. Lam *et al.* [2008] extended this idea to find all approximate pattern occurrences. For each read, their approach recursively descends a prefix trie, i.e. the suffix trie of the reversed text, analogously to Algorithm 4.9 on page 79.

This idea of backward backtracking combined with heuristics to prune the search space is used by Bowtie, Bowtie 2, BWA, and Soap 2. Whereas most of the approaches search *whole* reads and recursively enumerate mismatches (Bowtie, Soap 2) or also indels (BWA), Bowtie 2 uses the FM index to search exact seeds and verifies candidates with an approximate matching algorithm, as described for seed-based filters above. In general, the backtracking approach is not limited to prefix-tries and was successfully adapted to suffix trees in [Hoffmann *et al.*, 2009].

Recursive approaches are usually designed for the fast search of one or a few locations where reads map with low error rates. These search algorithms are mostly based on heuristics and optimized for speed instead of sensitivity. As they aim at directly finding the “best” location for mapping a read, they are called *best-mappers*. If a read has multiple mapping locations, the best location is randomly chosen either error-based or quality-based, where the first strategy minimizes the absolute number of errors and the second prefers alignments where mismatches correspond to low base-call qualities. We will show that both strategies have limitations in the presence of repeats or SNPs and that best-mappers are not applicable to reliably detect sequence variations (Section 6.10.5).

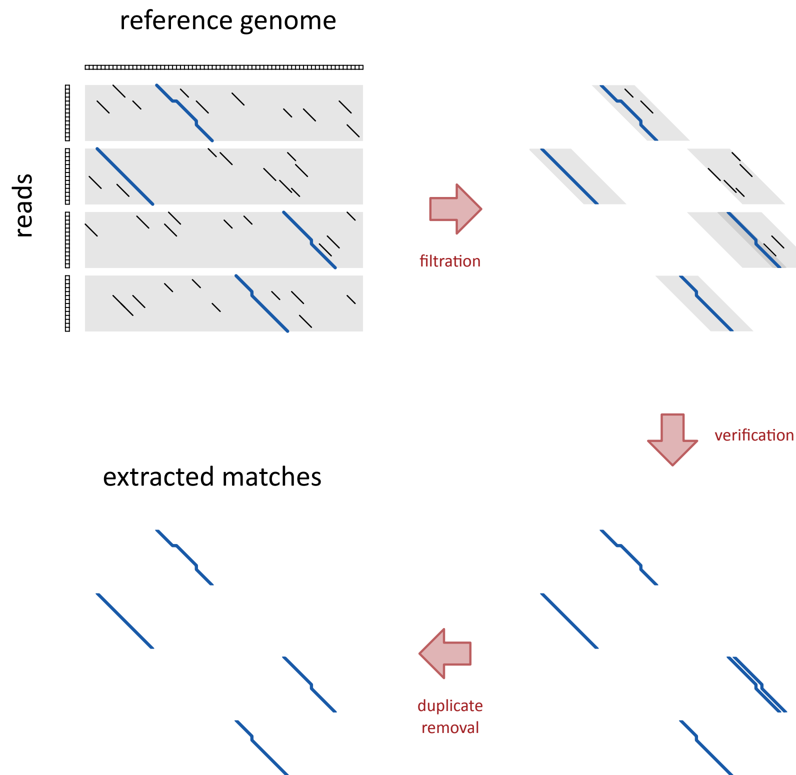


Figure 6.1: Match extraction in RazerS. The reference is scanned by a q-gram based filter which outputs candidate parallelograms that possibly contain a match. To verify whether the parallelograms contain true matches, they are searched with an approximate string matching algorithm. At last, duplicate matches, a result of overlapping parallelograms, are removed.

6.2 The RazerS algorithm

Some of the common problems inherent in many existing read mappers are that they use complicated rules to chose the best match, heuristics that prohibit full-sensitivity, or filters that must be parametrized manually. As a consequence, a clear definition of their solution space is often impossible and they are hard to adapt to a specific biological problem.

RazerS is a versatile q-gram based all-mapper that distinguishes itself in several respects from existing algorithms. First, it can be easily parametrized and maps reads using edit or Hamming distance (in filtration and verification phase without any restrictions). Second, given a user-defined sensitivity (possibly 100 % making the mapper full sensitive), we integrated an algorithm that automatically parametrizes the mapper such that the chosen sensitivity will be exceeded in expectation. Finally, our implementation can map millions of reads of any length with an arbitrary number of errors and is currently the fastest in reporting all hits for typical read lengths and loss rates. It supports paired-end mapping, the SAM output format [Li *et al.*, 2009a], shared-memory parallelism, and requires no prebuilt index.

RazerS consists of four modules: *parameter chooser*, *filter*, *verifier*, and *match processor*. Based on the user-specific read mapping settings (e.g. minimal sensitivity, error rate, given read lengths) the *parameter chooser* selects and parameterizes one of the two available filters such that the required minimal sensitivity is exceeded in expectation and the overall mapping time is minimal. Both strands of the reference genome are divided into windows of fixed length. The selected *filter* constructs a q-gram index of the reads and scans each window to collect candidate regions that possibly contain a read match. After a window has been processed, a Hamming or edit distance *verifier* examines the collected candidate regions as to whether they contain a true match. In regular intervals, the *match processor* searches the recorded matches for duplicates, which are artifacts by the filtration method. It also adapts the filters to be more stringent, if the number of matches per read is limited and enough distinct matches have been found.

Figure 6.1 depicts the steps required to extract true matches. We describe the details of each step in the following.

6.3 Definitions

The *general read mapping problem*, which we consider in the following, can be formalized as follows:

Definition 6.1 (read mapping problem)**.** Given a set of read sequences $\mathcal{R} \subset \Sigma^*$, a reference sequence $G \in \Sigma^*$, and an error rate $\varepsilon \in \mathbb{R}$, for every read $r \in \mathcal{R}$ find all substrings g of G that are within distance $\lfloor \varepsilon |r| \rfloor$ to r. The occurrences of g in G are called *(read) matches*.

Common distance measures are Hamming distance or edit distance, see Section 2.4. An error rate ε instead of an absolute number of errors takes account to the fact that read lengths may vary. For the sake of simplicity, we assume in the following that all reads have the same length m and the tolerated distance is $k = \lfloor \varepsilon |r| \rfloor$. However, all the proposed approaches are applicable to and were implemented for varying read lengths.

In the following sections, we consider transcripts and first need to generalize the q-gram lemma to introduce a more precise definition of common q-grams, the *q-matches*.

Lemma 6.1 (generalized q-gram lemma)**.** *Given an edit transcript T between two strings $s, t \in \Sigma^*$ with $k = \|T\|_E$ errors. Then T contains at least $t = |T| - k(q+1) + 1$ occurrences of the substring $\mathtt{M}^q$, called q-matches.*

Proof. For $k = 0$ holds $T = \mathtt{M}^{|T|}$ and T contains $t = |T| - q + 1$ q-matches. Every additional error x either replaces a former match $\mathtt{M}$ in T, if $x \in \{\mathtt{R}, \mathtt{D}\}$, or is inserted into T, if $x = \mathtt{I}$. In the first case, the replaced $\mathtt{M}$ can be part of at most q former q-matches. In the second case, the inserted $\mathtt{I}$ extends T by one character and is contained in at most $q - 1$ former q-matches. ∎

In particular, the above lemma holds for Hamming transcripts as they are special edit transcripts. The original q-gram lemma (Lemma 5.1 on page 90) is a direct consequence

of Lemma 6.1 as $|T|$ is always greater than or equal to the length of the larger string. Due to the correspondence of transcripts and alignments (Definition 2.12 on page 18), a q-match is a diagonal stretch of length q in an alignment trace and hence corresponds to a diagonal stretch of q matching bases in the dot plot, the so-called *q-hit*.

Definition 6.2 (q-hit)**.** Two strings $s, t \in \Sigma^*$ contain a so-called q-hit (i, j), if they share a common q-gram $s[i..i + q) = t[j..j + q)$.

The q-gram filters, which we propose in the following, search the dot plot between reads and reference for q-hits to find match candidates.

6.4 Filtration

To make RazerS applicable to a broad spectrum of use cases, we implemented two fast filtration algorithms: (1) SWIFT, a q-gram counting filter, and (2) the seed-based pigeonhole filter. Both differ in filtration specificity and processing speed. In Section 6.10.1 we will empirically compare their specificities and overall mapping times.

6.4.1 SWIFT filter

To find potential match regions of reads in the reference genome we adapted the SWIFT algorithm, described in Section 5.5.1, to search semi-global instead of local matches. As a semi-global match aligns the *whole* read to a genomic substring, we need to consider dot plot parallelograms that span the whole read. Hence, we can omit to open or close parallelograms as it is done by the original SWIFT filter for local alignments.

The generalized q-gram lemma (Lemma 6.1) defines a threshold t, such that every read alignment with at most k errors contains at least t q-matches. They correspond to at least t q-hits that reside in a dot plot parallelogram of at most $d = k + 1$ consecutive diagonals, as the alignment contains at most k insertions or deletions. In case of Hamming distance there is a single diagonal ($d = 1$) completely covering all q-hits. Hence it is sufficient to count q-hits in each possible $|r| \times d$ parallelogram, i.e. the intersection of $|r|$ consecutive rows and d consecutive diagonals in the dot plot. To reduce processing overhead, we count them in larger $|r| \times w$ parallelograms, where $w > d$, that overlap by $d - 1$ diagonals as every $|r| \times d$ parallelogram is contained in one $|r| \times w$ parallelogram, see Figure 6.2.

As an optimization the width w is chosen such that parallelograms start at multiples of a power of 2. In this way, the parallelogram counter can be efficiently determined by bit-shifting the coordinate of the q-hit diagonal.

We construct a q-gram index of all overlapping read q-grams and search common q-hits in a linear scan over the reference sequence G. During the scan only a limited number of counters is needed per read at the same time. As every $|r| \times w$ parallelogram spans at most $|r| + w - 1$ letters of G, we re-use parallelogram counters after $|r| - w - q$ scan steps. Each q-hit increases the counters of the parallelograms it is covered by. Whenever a counter reaches the threshold t, the parallelogram is recorded as a candidate region.

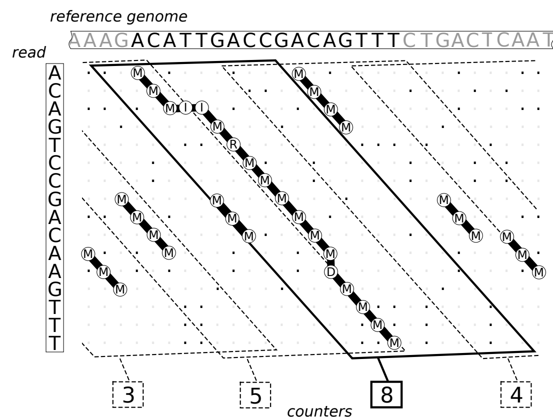


Figure 6.2: *Example for q-gram counting in parallelograms[a]. The dot plot between a read and a genome is covered by parallelograms that count common 3-grams, span $w = 12$ diagonals, and overlap by $k = 4$ diagonals. The marked parallelogram contains a 4-error read match and counts eight 3-hits that correspond to the seven 3-matches in the transcript and one random 3-hit.*

[a] Parallelograms are intentionally rotated to visualize their overlap.

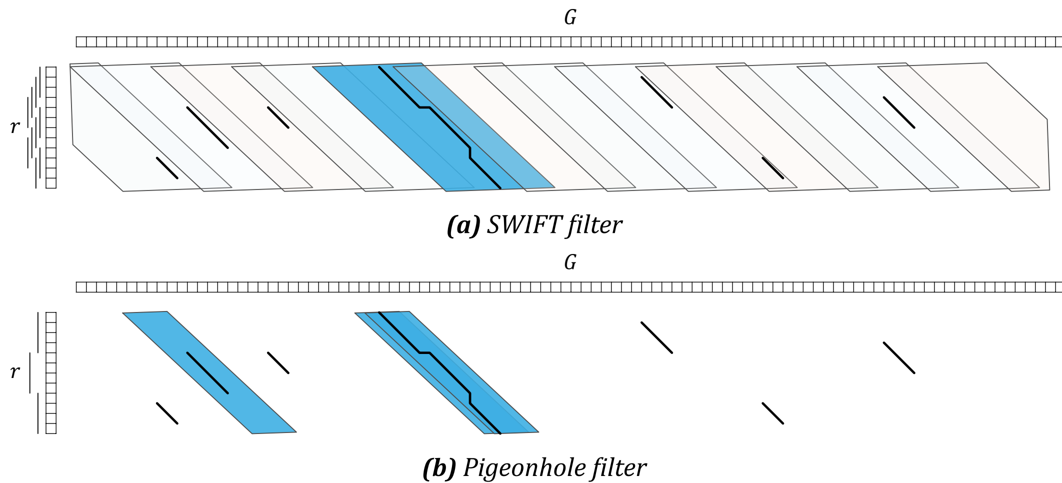
We extended the filter to be applicable to gapped q-grams as well, given a shape Q and an appropriate threshold t. Burkhardt and Kärkkäinen [2003] proposed how to compute optimal thresholds for full-sensitivity Hamming filters. In Section 6.5.1, we introduce a more generic approach that computes the sensitivity of Hamming and edit distance filters for *arbitrary* thresholds.

6.4.2 Pigeonhole filter

The second filter is based on the pigeonhole principle, which states that if a read is cut into $k + 1$ pieces, then in every approximate match of the read with at most k errors at least one piece occurs without error [Baeza-Yates and Navarro, 1999]. If all reads have the same length m, they are cut into $\lfloor \varepsilon m + 1 \rfloor$ pieces of length $q = \left\lfloor \frac{m}{\lfloor \varepsilon m + 1 \rfloor} \right\rfloor$, where ε is the tolerated error rate.

For reads of arbitrary lengths the minimal q is chosen to build a q-gram index over the pieces of the reads. These pieces are then searched in a linear scan of the reference sequence. For every exact match, the dot plot parallelogram, consisting of the diagonals that are at most k diagonals apart of the matching piece, is considered as a candidate region for a match within the tolerated edit distance. In Hamming distance mode, only the diagonals that cover matching pieces are considered as candidate regions. The candidate parallelograms of all matching pieces are recorded and verified in the further verification step.

As the pigeonhole filter requires no counters and searches only *non-overlapping* q-grams of the reads in the reference, it requires less processing time and a smaller index on the expense of more verifications compared to the SWIFT filter.



(a) *SWIFT filter*

(b) *Pigeonhole filter*

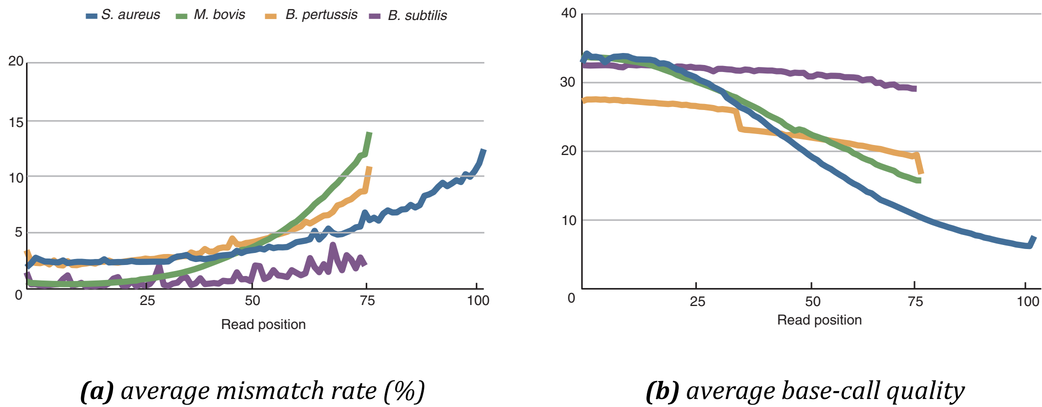
Figure 6.3: *Filters used in RazerS. The SWIFT filter (a) counts common q-grams in overlapping parallelograms. Parallelograms with a sufficiently high number of common q-grams are reported as candidate regions. The pigeonhole filter (b) divides the read into non-overlapping q-grams (seeds) and reports for each occurrence in the reference G a surrounding parallelogram as candidate region.*

6.5 Lossy filtration and prediction of sensitivity

Both filters are fully sensitive if parameterized as described above, i.e. every occurrence of a read within the tolerated edit or Hamming distance will be detected as a candidate region and positively verified in the verification step. However, the filter parametrization depends on a number of worst cases where all errors are almost equidistantly distributed, which is small compared to all possible error distributions. Further, some sequencing technologies show an accumulation of sequencing errors towards the 3’ end of the read (Sanger and Illumina) or clustered errors at SNPs (SOLiD) or homopolymer runs (Roche/454), which lowers the worst-case probabilities compared to independent and identically distributed errors. As an example, Figure 6.4a shows the positional error profile observed after mapping the reads of typical Illumina runs to their reference genomes. A strong correlation between observed and predicted errors becomes apparent when comparing Figure 6.4a with Figure 6.4b, which shows the average base-call quality values.

In this section, we propose modifications that make both filters more stringent and reduce the filtration time, the number of recorded candidate regions, and hence the overall mapping time. As the q-gram and pigeonhole lemmas are strict, our improvements can only be achieved at the expense of sensitivity. To obtain control over the loss of a filter, we additionally devise methods to compute the sensitivity of both filters that are in general applicable to any filter based on q-gram counting or the pigeonhole principle.

The sensitivity of a filter is the probability that a randomly chosen *true* match (r, g) is detected by the filter as a match candidate. Existing sensitivity estimation approaches assume errors to be generated by a Markov process [Herms and Rahmann, 2008] or are



***(a)** average mismatch rate (%)*

***(b)** average base-call quality*

***Figure 6.4:** Plots of average mismatch ratio (a) and base-call quality (b) along the cycles of typical Illumina runs. Plots by Nakamura* et al. *[2011].*

limited to uniform error distribution [Li *et al.*, 2003]. Our methods efficiently estimate the filtration sensitivity under any position-dependent error distribution, e.g. as observed in Sanger or Illumina sequencing technologies [Dohm *et al.*, 2008; Nakamura *et al.*, 2011].

We consider positional error probabilities p_i^{R}, i.e. the probability that in a random match of a read there is a mismatch at nucleotide position i. As errors we consider base miscalls and SNPs, and before mapping we compute an average error profile over all reads based on their base-call quality values and a user-specific mutation rate. For an average Phred quality [Ewing and Green, 1998] value q_i at position i in the reads, the base-miscall probability χ_i can be computed as follows:

$$\chi_i \quad = \quad 10^{\frac{-q_i}{10}}\,. \tag{6.1}$$

With a given mutation rate μ, we compute the probability p_i^{R} to observe a mismatch at position i as:

$$p_i^{\mathrm{R}} = 1 - (1 - \chi_i) \cdot (1 - \mu)\,. \tag{6.2}$$

6.5.1 Sensitivity calculation of q-gram counting filters

A (q, t)-*counting filter* is a filtering algorithm that detects any pair (r, g) for which a transcript T from r to g exists that contains at least t q-matches. In this section, we devise DP algorithms that efficiently compute the sensitivity of any (q, t)-counting filter and, especially, the SWIFT filter used in RazerS. Knowing the filtration sensitivity enables us to increase q or t and reduce random hits while guaranteeing a required minimal sensitivity.

Hamming distance sensitivity. We first consider the read mapping problem for reads of length m and matches with a Hamming distance of at most k. To determine a lower bound for the sensitivity of a (q, t)-counting filter we could enumerate all Hamming transcripts with up to k replacements and sum up the occurrence probabilities of those tran-

scripts having at least t substrings $\mathtt{M}^q$. However, as there are $\sum_{i=0}^{k}\binom{m}{i}$ different transcripts, a full enumeration takes $\Omega((\frac{m}{k})^k)$ time and is not feasible for large reads or high error rates, e.g. of length $m = 200$ with $k = 20$ errors. We have developed a dynamic programming approach which is significantly faster by using a recurrence similar to the threshold calculation in [Burkhardt and Kärkkäinen, 2003].

Given a position-dependent error distribution $p_i^{\mathtt{R}}$. Then the occurrence probability of a Hamming transcript T over the alphabet $\Phi = \{\mathtt{M}, \mathtt{R}\}$ is the product of the individual occurrence probabilities of transcript characters $p(T) = \prod_{i=0}^{|T|-1} p_i^{T[i]}$, with $p_i^{\mathtt{M}} = 1 - p_i^{\mathtt{R}}$. We calculate the sensitivities for matches with e errors for each $e \leq k$ separately. Let $S(m, e, t)$ be the sum of occurrence probabilities of transcripts of length m, having e errors, and at least t q-matches. The sensitivity of a (q, t)-counting filter to detect e-error matches is at least:

$$P\big(T \text{ contains} \geq t\ q\text{-matches} \,\big|\, \|T\|_{\mathrm{E}} = e\big) = \frac{S(m,e,t)}{S(m,e,0)}\,. \tag{6.3}$$

We will see how to calculate $S(m, e, t)$ using a DP algorithm. Let $p(T, j) = \prod_{i=0}^{|T|-1} p_{i+j}^{T[i]}$ be the occurrence probability of sub-transcript T to occur after j letters of a read. We define $R(i, e, t, T_2)$ the sum of occurrence probabilities of transcripts $T_1 \in \Phi^i$, s.t. T_1 has e errors and the concatenation T_1T_2 contains at least t substrings $\mathtt{M}^q$. By definition of S, the following holds:

$$S(m,e,t) = \sum_{T \in \Phi^q,\, \|T\|_{\mathrm{E}} \leq e} R\left(m-q, e-\|T\|_{\mathrm{E}}, t, T\right) \cdot p(T, m-q)\,. \tag{6.4}$$

The sum goes over all transcript ends T of length q with at most e errors. The right factor is the probability of T occurring at the end of a random transcript of length m. The left factor is the occurrence probability sum over all transcript beginnings, s.t. the concatenation of beginning and end is a transcript of length m with e errors, and at least t q-matches. With the following lemma a DP algorithm can be devised to determine R and therefore the sensitivities $S(m, e, t)$ for all $e = 0, \ldots, k$ and $t = 1, \ldots, t_{\max}$ in $O(m \cdot k \cdot t_{\max} \cdot 2^q)$ time.

Lemma 6.2. *Let $i, q \in \mathbb{N}$; $e, t \in \mathbb{Z}$; $T \in \{\mathtt{M}, \mathtt{R}\}^q$. R can be calculated using the following recurrence:*

$$R(0,e,t,T) = \begin{cases} 1, & \text{if } e = 0, t \leq \delta(T) \\ 0, & \text{else,} \end{cases} \tag{6.5}$$

$$\begin{aligned} R(i,e,t,T) = {} & p_{i-1}^{\mathtt{M}} \cdot R(i-1, e\phantom{{}-1}, t-\delta(T), \operatorname{shift}(\mathtt{M},T)) \\ & + p_{i-1}^{\mathtt{R}} \cdot R(i-1, e-1, t-\delta(T), \operatorname{shift}(\mathtt{R},T))\,, \end{aligned} \tag{6.6}$$

with

$$\operatorname{shift}(x,T) = xT\big[0..|T|-1\big)\,, \tag{6.7}$$

$$\delta(T) = \begin{cases} 1, & \text{if } T = \mathtt{M}^q \\ 0, & \text{else.} \end{cases} \tag{6.8}$$

Proof. See Appendix A.2. ∎

Extension to gapped shapes. Given a shape $Q = \{i_1, i_2, \dots, i_q\}$ and a Hamming transcript T, we call the gapped q-gram $T[i + i_1]T[i + i_2] \dots T[i + i_q] = \mathtt{M}^{|Q|}$ a Q-match at position i. A *(Q,t)-filter* is an algorithm that detects any pair (r, g) for which a transcript T from r to g with at least t Q-matches exists. To extend the sensitivity calculation to gapped shapes Q, the transcript T in Lemma 6.2 must be resized to cover the whole shape and the matching criterion in $\delta(T)$ must be adapted. This can be done by replacing q in (6.4) by span(Q) and $\delta(T)$ in (6.5) and (6.6) by:

$$\delta(T) = \begin{cases} 1, & \text{if } T[i_1]T[i_2] \dots T[i_q] = \mathtt{M}^{|Q|} \\ 0, & \text{else.} \end{cases} \tag{6.9}$$

For two strings of length span(Q) with the Hamming transcript $T \in \Phi^{\mathrm{span}(Q)}$, $\delta(T)$ returns 1 iff they share their sole Q-gram. A lemma similar to Lemma 6.2 can be proven analogously.

Edit distance sensitivity. We now consider the read mapping problem under edit distance and propose a DP algorithm that computes the sensitivity of a q-gram filter to detect a randomly chosen true match (r, g) with edit distance $d(r, g) \leq k$ as a potential match. Again, we consider all reads $r \in \mathcal{R}$ to be of equal length m and reduce randomly chosen true matches (r, g) to randomly chosen edit transcripts from r to g with $\Phi = \{\mathtt{M}, \mathtt{R}, \mathtt{D}, \mathtt{I}\}$. We therefore assume a given error distribution that associates each nucleotide position i in a read with positional error probabilities $p_i^{\mathtt{R}}, p_i^{\mathtt{D}}, p_i^{\mathtt{I}}$, where $p_i^{\mathtt{R}}$ and $p_i^{\mathtt{D}}$ are the probabilities that the nucleotide $r[i]$ is replaced or deleted, and $p_i^{\mathtt{I}}$ the probability that a single nucleotide is inserted after the nucleotide $r[i]$ in g. For any transcript T from read to genome we define $\|T\|_{\mathtt{R}} = \big|\{i \mid T[i] \in \{\mathtt{M}, \mathtt{R}, \mathtt{D}\}\}\big|$, the number of read characters affected by T. Finally, we assume the following occurrence probability of an edit transcript T:

$$p(T) = \prod_{i=0}^{|T|-1} p_{\|T[0..i)\|_{\mathtt{R}}}^{T[i]}, \tag{6.10}$$

with $p_i^{\mathtt{M}} = 1 - p_i^{\mathtt{R}} - p_i^{\mathtt{D}} - p_i^{\mathtt{I}}$. We define the set $\Phi(i) = \{T \mid T \in \Phi^*, \|T\|_{\mathtt{R}} = i\}$ of transcripts from reads of length i. In the following, we omit to enumerate transcripts beginning or ending with $\mathtt{I}$, as these transcripts can always be shortened resulting in a match with less errors. Similar to (6.4) the occurrence probability sum $S(m, e, t)$ of edit transcripts from reads of length m, with e errors, and at least t substrings $\mathtt{M}^q$ can be written as:

$$S(m, e, t) = \sum_{\substack{T \in \Phi(q),\ \|T\|_{\mathtt{E}} \leq e, \\ T[|T|-1] \neq \mathtt{I}}} R\left(m - q, e - \|T\|_{\mathtt{E}}, t, T\right) \cdot p(T, m - q), \tag{6.11}$$

where $R(i, e, t, T_2)$ is the occurrence probability sum of transcripts $T_1 \in \Phi(i)$ with e errors, s.t. $T_1[1] \neq \mathtt{I}$ and $T_1 T_2$ contains at least t substrings $\mathtt{M}^q$. $p(T, j) = \prod_{i=0}^{|T|-1} p_{\|T[0..i)\|_{\mathtt{R}}+j}^{T[i]}$ is the occurrence probability of sub-transcript T to occur after j letters of a read.

Lemma 6.3. *Let $e, i, q \in \mathbb{N}$; $t \in \mathbb{Z}$; $T \in \Phi(q)$. R can be calculated using the following recurrence:*

$$R(0,e,t,T) = \begin{cases} 1, & \text{if } e = 0,\ t \le \delta(T),\ T[0] \ne \mathtt{I} \\ 0, & \text{else,} \end{cases} \tag{6.12}$$

$$R(i,-1,t,T) = 0, \tag{6.13}$$

$$\begin{aligned} R(i,e,t,T) = &\quad p^{\mathtt{M}}_{i-1} \cdot R(i-1, e, t-\delta(T), \mathrm{shift}(\mathtt{M},T)) \\ &+ p^{\mathtt{R}}_{i-1} \cdot R(i-1, e-1, t-\delta(T), \mathrm{shift}(\mathtt{R},T)) \\ &+ p^{\mathtt{D}}_{i-1} \cdot R(i-1, e-1, t-\delta(T), \mathrm{shift}(\mathtt{D},T)) \\ &+ p^{\mathtt{I}}_{i-1} \cdot R(i, e-1, t, \mathtt{I}T), \end{aligned} \tag{6.14}$$

with

$$\mathrm{shift}(x,T) = xT\big[0..\max\{i \in [0..|T|-1) \mid T[i] \ne \mathtt{I}\}\big], \tag{6.15}$$

$$\delta(T) = \begin{cases} 1, & \text{if } T \text{ contains } \mathtt{M}^q \\ 0, & \text{else.} \end{cases} \tag{6.16}$$

Proof. See Appendix A.2. ■

Accordingly, the sensitivities $S(m,e,t)$ for all $e = 0,\dots,k$ and $t = 1,\dots,t_{\max}$ can be determined in $\mathcal{O}(m \cdot k \cdot t_{\max} \cdot 4^{q+k})$ time.

Algorithm 6.1: $\delta(T)$ for gapped shapes

```
   input  : transcript T, shape Q
   output : 0, if T destroys Q-gram; 1, else
1  r ← 0, g ← 0
2  j ← min{i | T[i] ≠ I}
3  for i ← j to |T| − 1 do
4      if T[i] ≠ D then
5          g ← g + 1
6      if T[i] ≠ I then
7          if r ∈ Q and (T[i] ≠ M or r ≠ g) then
8              return 0
9          r ← r + 1
10 return 1
```

Extension to gapped shapes. Although “don’t care” positions of gapped shapes are not immune to indels, we extend the edit distance sensitivity calculation to gapped shapes for the sake of completeness. To calculate R and S for a gapped shape Q, every q in (6.11) and Lemma 6.3 must be replaced by span(Q). Algorithm 6.1 can be used to detect whether a common Q-gram is retained or destroyed by a transcript affecting *span*(Q) read characters.

6.5.2 Sensitivity calculation of pigeonhole filters

A lossless pigeonhole filter divides a read into at least $k + 1$ fragments and uses them as seeds to detect all k-error matches. As fragments we use the first $k + 1$ non-overlapping q-grams of the reads where q is chosen as large as possible. In expectation, every read q-gram has $n/4^q$ occurrences in a genome of length n. To reduce the number of random candidates and to reduce the overall running time, we increase q and allow the seeds to overlap, see Figure 6.5. However, with overlapping seeds some matches will be missed by the filter, e.g. if every odd seed overlap contains an error.

With a (q, Δ)-seed filter we denote a filter that uses all q-grams starting at multiples of Δ in the read as seeds, with $q/2 \leq \Delta \leq q$, such that adjacent q-grams overlap by $q - \Delta$ characters. To compute the sensitivity of such a filter, we consider Hamming transcripts between a read of length m and all of its true matches. Again, the sensitivity for matches with $e = 0, 1, \ldots, k$ errors is the sum of occurrence probabilities of e-error transcripts that are detected by the filter divided by the probability that an e-error transcript occurs. Instead of enumerating all possible e-error Hamming transcripts we devise a DP algorithm that virtually splits the transcript into segments at q-gram boundaries $\Delta, q, 2\Delta, \Delta + q, \ldots, (k + 1)\Delta, k\Delta + q$ and denote the first $2(k + 1)$ segments from left to right as $x_0, y_0, x_1, y_1 \ldots, x_k, y_k$, see Figure 6.6. Our approach is analogously applicable to edit distance as insertions or deletions behave like mismatches in relation to destroyed seeds.

We first compute the probability $P\big(\|T[i..j)\|_{\mathrm{E}} = e\big)$ that a random Hamming transcript T contains e errors in the segment $T[i..j)$ given positional error probabilities p_i^{R} and their

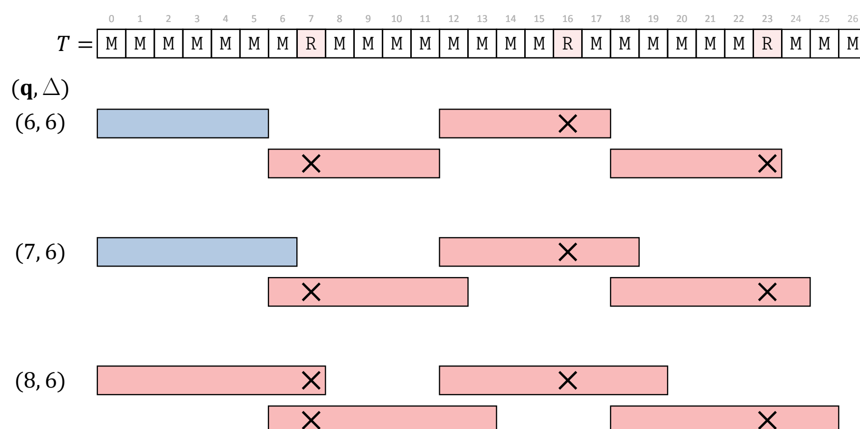


Figure 6.5: *Examples for (q, Δ)-seed filters. The upper is lossless for up to 3 mismatches and based on the pigeonhole principle. The second uses 7-grams that overlap by one position and still recognizes the match, whereas the third uses 8-grams and misses the match.*

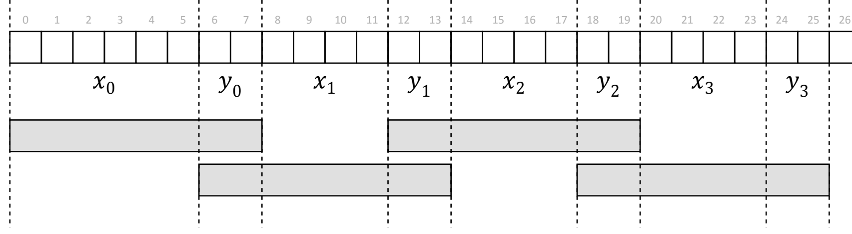


Figure 6.6: *A (q, Δ)-seed filter, with $q = 8$ and $\Delta = 6$, for searching matches with up to $k = 3$ errors (seed i consists of segments y_{i-1}, x_i, and y_i, except for $i = 0$).*

complementary probabilities $p_i^{\mathrm{M}} = 1 - p_i^{\mathrm{R}}$:

$$P\big(\|T[i..i)\|_{\mathrm{E}} = e\big) = \begin{cases} 1, & \text{if } e = 0 \\ 0, & \text{else,} \end{cases} \tag{6.17}$$

$$P\big(\|T[i..i+1)\|_{\mathrm{E}} = e\big) = \begin{cases} p_i^{\mathrm{M}}, & \text{if } e = 0 \\ p_i^{\mathrm{R}}, & \text{if } e = 1 \\ 0, & \text{else,} \end{cases} \tag{6.18}$$

$$P\big(\|T[i..j)\|_{\mathrm{E}} = e\big) = p_{j-1}^{\mathrm{M}} \cdot P\big(\|T[i..j-1)\|_{\mathrm{E}} = e\big) + p_{j-1}^{\mathrm{R}} \cdot P\big(\|T[i..j-1)\|_{\mathrm{E}} = e-1\big). \tag{6.19}$$

Let $L(i, e, y)$ be the probability of the event that the first $i + 1$ seeds contain overall e errors, each at least one error, and y_i contains y errors. Let X_i and Y_i be random variables for the number of errors in the segments x_i and y_i, then L can recursively be computed as follows:

$$L(0, e, y) = \begin{cases} 0, & \text{for } e = 0 \\ P(X_0 = e - y) \cdot P(Y_0 = y), & \text{else,} \end{cases} \tag{6.20}$$

$$L(i, e, y) = \sum_{s=1}^{e} \sum_{y'=0}^{s-y} L(i-1, e-s+y', y') \cdot P(X_i = s - y - y') \cdot P(Y_i = y). \tag{6.21}$$

The probability that all seeds are destroyed with overall e errors is:

$$L_{\mathrm{all}}(e) = \sum_{y=0}^{e} \sum_{x=0}^{e} L(k, e-x, y) \cdot P(\|T[k\Delta + q \mathrel{..} n)\|_{\mathrm{E}} = x), \tag{6.22}$$

and thus the sensitivity of the (q, Δ)-seed filter for matches with at most k errors is:

$$S(q, \Delta, k) = 1 - \sum_{e=0}^{k} \frac{L_{\mathrm{all}}(e)}{P(\|T\|_{\mathrm{E}} = e)}. \tag{6.23}$$

6.5.3 Choosing filtration parameters

Now that we are able to compute the sensitivity of a filter parametrization, we want to automatically choose filtration parameters, such that (1) a certain sensitivity level is achieved and (2) the running time of the mapping procedure is minimized.

SWIFT filter. Due to the huge parameter space of thresholds t and possible gapped shapes Q, we have precomputed the sensitivities using a selection of different shapes and thresholds for all read lengths from 24 to 100 bp and error rates up to 10 %. As an error distribution we assume a typical Illumina error profile [Dohm *et al.*, 2008]. Additionally, all parameter combinations were used to run RazerS on simulated data, yielding a rough estimate for the corresponding relative mapping times. Parameters for reads longer than 100 bp are extrapolated from parameters of precomputed shorter reads with the same error rate. Given a user-defined minimum sensitivity, suitable filtration parameters are chosen from the precomputed tables such that the anticipated running time is minimized.

If preferred, the parameter tables can be precomputed based on a machine-specific error distribution and user-defined shapes. This error distribution can be obtained in two ways. (1) *Quality based probabilities:* transform the average base-call quality value for each position into a probability value. (2) *A posteriori probabilities:* map a small subset of reads and determine the position-dependent error frequency. Given an error distribution the parameters for reads of length 50 bp can, for instance, be calculated within 10 minutes.

Pigeonhole filter. In contrast to the SWIFT filter, the small parameter space of the pigeonhole filter allows for the filter to be adjusted based on the quality-based probabilities of the read set at hand. Before starting the mapping, RazerS estimates the sensitivities of different filter settings and maximizes the seed length q as it has the greatest influence on the overall running time. Beginning with the lossless setting $q = \Delta = \lfloor m/(k+1) \rfloor$, it step-wise increases q as long as the estimated sensitivity is higher than required, q does not exceed the maximal seed length of 31, and not more than two seeds overlap ($q \leq 2\Delta$). The corresponding step sizes $\Delta = \lfloor (m-q)/k \rfloor$ are chosen such that each read contains $k+1$ overlapping seeds.

6.6 Verification

The result of the filtration step is a set of candidate regions and corresponding reads. A candidate region is a parallelogram in the dot plot that might contain the alignment trace of a match. Depending on the considered string metric it is verified by one of the approximate matching algorithms explained in the following.

6.6.1 Hamming distance verification

In Hamming mode, a match covers solely one dot plot diagonal. Hence, the candidate parallelogram can be verified by scanning each diagonal while counting the number of mismatches between read and reference sequence. A diagonal can be skipped as soon as the counter exceeds the number of tolerated errors. Otherwise, a match has been found.

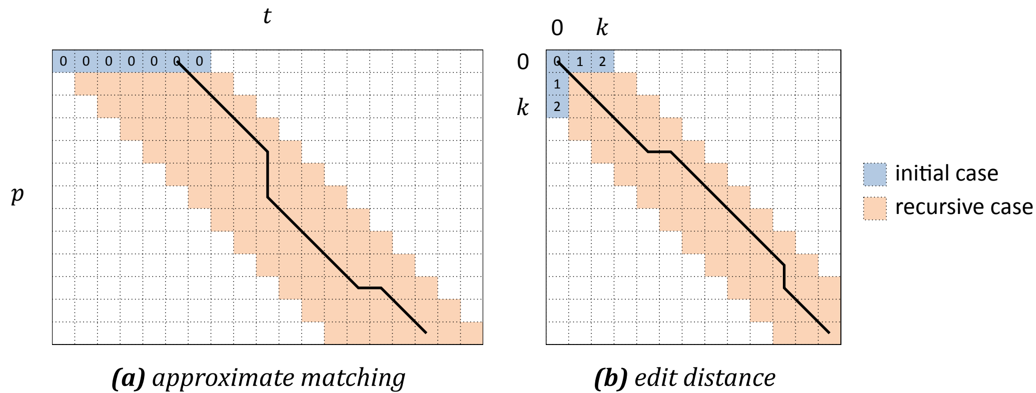
6.6.2 Edit distance verification

To verify an edit distance match candidate, the reference substring covered by the candidate region could be searched with one of the approximate pattern matching algorithms explained in Section 2.5. However, none of these algorithms take account of the parallelogramic shape of the candidate region but instead verify the whole surrounding dot plot rectangle. To take advantage of knowing the shape of the candidate region, we implemented a banded version of Myers' [1999] bit-vector algorithm as it was proposed in [Hyyrö, 2003] with some improvements.

Myers' bit-vector algorithm. The original algorithm by Myers can be used for approximate pattern matching in a dot plot rectangle. For each position in the reference, it determines the minimal number of errors a match ending there. The underlying idea is the same as in the DP algorithm for approximate pattern matching by Sellers [1980], but the implementation is much more efficient as it encodes a whole DP column in two bit-vectors and computes the adjacent column in a constant number of 12 logical and 3 arithmetical operations. For reads up to length 64 bp, CPU registers can directly be used. For longer reads, bit-vectors and operations must be emulated using multiple words where only words affecting a possible match need to be updated [Ukkonen, 1985]. However, the additional processing overhead results in a performance drop for reads of length 65 bp and longer.

Banded variant by Hyyrö. Hyyrö [2003] proposed a variant of Myers' algorithm that only computes DP cells that are covered by a parallelogram. Hence, only the columns of the parallelogram need to be encoded by bit-vectors, which makes it applicable to parallelograms with the width of up to 63 cells without the need for bit-vector emulation. However, the banded variant as proposed in [Hyyrö, 2003] still requires to precompute bitmasks of $|\Sigma| \times m$ bits for each read of length m and does not support clipped parallelograms. Clipping of parallelograms is, however, necessary to find the begin position of a match in the reference and to verify parallelograms that cross the beginning or end of the reference sequence.

Our banded algorithm. We devise a banded variant of Myers' algorithm that supports clipped parallelograms, requires no preprocessing information at all, and uses online-computed pattern bitmasks of $|\Sigma| \times w$ instead of $|\Sigma| \times m$ bits. Before going into algorithmic details, we sketch the outline of the edit distance verification. In contrast to Hamming distance verification, where the difference between begin and end position of every match equals the read length, Myers' algorithm reports only the ends of matches. More precisely, it determines the minimal number of errors for a fixed end position and a free begin position. To determine a corresponding begin position we search the read backwards with a fixed end position. As edit distance scores mismatches and indels equally, there can be multiple best match beginnings. We choose the largest best match to optionally shrink it later using an alignment algorithm for affine gap costs [Gotoh, 1982] where



***(a)** approximate matching*

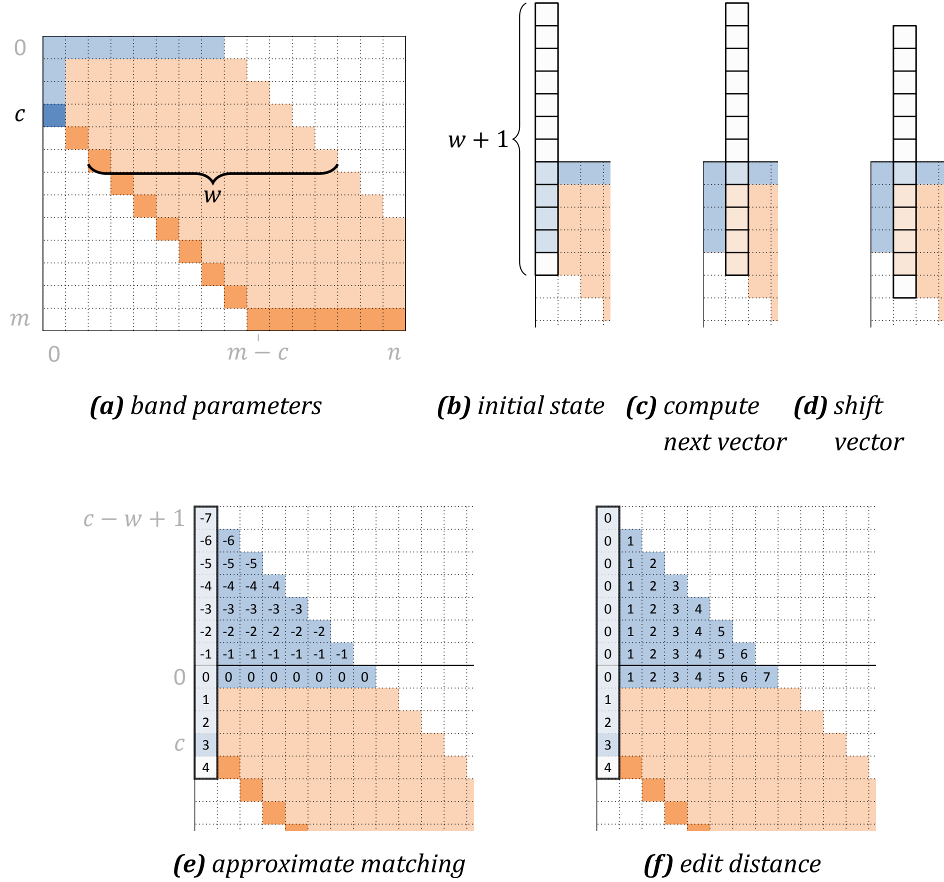
***(b)** edit distance*

***Figure 6.7:** Applications of the banded alignment algorithm are: (a) approximate pattern matching in a band of the DP matrix, e.g. a candidate region, or (b) edit distance computation between two sequences tolerating at most k errors.*

gaps are penalized slightly more than mismatches and gap opening costs are higher than extension costs.

In the following, we propose the algorithmic details of our banded algorithm. It is not only applicable to approximate pattern matching in (clipped) parallelograms (Figure 6.7a) but can also be used for edit distance computation (Figure 6.7b) with a small modification. For a given text t of length n and a given pattern p of length m we consider the DP matrix which has $m + 1$ rows and $n + 1$ columns. Let a band of w consecutive diagonals be given where the left-most diagonal is the main diagonal shifted by c diagonals to the left, see Figure 6.8a. The algorithm diagonally slides a column vector D of $w + 1$ cells over the band. Analogously to Myers' algorithm, D is encoded by delta bit-vectors VP and VN of size w and a variable *errors* that tracks the cell values of the lower band boundary (dark cells in Figure 6.8a). Each sliding step consists of a horizontal and a vertical step. In the horizontal step the delta vectors $D0$, HP and HN are computed as in Myers' algorithm, see Figure 6.8c and lines 12–15 in Algorithm 6.2. From these delta vectors VP and VN are deduced and shifted by 1 bit to the right in the vertical step, see Figure 6.8d and lines 16–18. In contrast to Myers' and Hyrrö's algorithms, we shift and update the pattern bitmasks online. In this way, we save a time consuming preprocessing (shaded areas in Figure 6.9) and reduce the required memory from $|\Sigma| \times m$ bits per read to overall $|\Sigma| \times w$ bits.

In the beginning, D covers the intersection of the first column and all band diagonals plus the diagonal left of the band as shown Figure 6.8b. As D initially represents cells beyond the DP matrix, they have to be initialized such that they have no unintended influence on the cells within the DP matrix and such that the first DP row contains zero values for approximate pattern matching or increasing values for the edit distance calculation. Setting the pattern bitmasks to zero for cells beyond the DP matrix, $VN = 0^w$ and $VP = 1^w$ for approximate pattern matching or $VP = 1^{c+1}0^{w-c-1}$ for edit distance results in the desired initialization patterns depicted in Figure 6.8. The following lemma guarantees that non-band cells are not used in the DP recurrence, and as $D0$ represents



(a) *band parameters* **(b)** *initial state* **(c)** *compute next vector* **(d)** *shift vector*

(e) *approximate matching* **(f)** *edit distance*

Figure 6.8: *Band parameters (a). A band is uniquely defined by the number of consecutive diagonals w and the row c that intersects the left-most diagonal and the first column. The initial state (b) and the recursion steps (c,d) are shown on the top right. The column vector initializations for approximate pattern matching (e) and edit distance calculation (f) are shown below.*

the diagonal delta of all w diagonals, the algorithm computes exactly the band depicted in Figure 6.8a.

Lemma 6.4. *Cells left and right of the band are not used for the computation of band cells.*

Proof. Let $k \in \mathbb{N}$ with $k \leq \min(m - c, n)$ and $i = c + k$ and $j = k$ then $C_{i,j-1}$ is a cell on the diagonal left of the band and $C_{i,j}$ is in the left-most band diagonal. After shifting $D0$ by 1 bit to the right in line 16 of Algorithm 6.2 bit w in X is 0 and thus bit w in VN will be 0 after the assignment in line 17. Therefore holds $C_{i-1,j-1} \leq C_{i,j-1}$ and $C_{i,j-1}$ have no influence on the minimum stored in $C_{i,j}$. The cells in the diagonal right (or above) of the band have no influence as bit 0 in $D0$ depends only on the comparison of pattern and text characters and the vertical delta $D0[0] \Leftrightarrow (p[i] = t[j]) \wedge VN[0]$. The horizontal deltas between band and non-band cells have no influence on $D0[0]$ and thus no influence on band cells. ∎

Algorithm 6.2: BANDEDMYERS(t, p, k, w, c)

```
   input  : text t ∈ Σ*, pattern p ∈ Σ*, errors k and band parameters w, c
   output : text end positions of matches with up to k errors

 1 foreach x ∈ Σ do                               // initialize pattern bitmasks
 2     B[x] ← 0^w
 3 for j ← 0 to c − 1 do
 4     B[p[j]] ← B[p[j]] | 0^(c−j−1) 1 0^(w−c+j)

 5 VP ← 1^w; VN ← 0^w                            // initialize vertical delta vectors
 6 errors ← c

 7 for pos ← 0 to n − 1 do

 8     foreach x ∈ Σ do                           // shift and update pattern bitmasks
 9         B[x] ← B[x] ≫ 1
10     if pos + c < m then
11         B[p[pos + c]] ← B[p[pos + c]] | 1 0^(w−1)

12     X ← B[t[pos]] | VN                         // compute horizontal delta vectors
13     D0 ← ((VP + (X & VP)) ^ VP) | X
14     HN ← VP & D0
15     HP ← VN | ~ (VP | D0)
16     X ← D0 ≫ 1                                 // compute and shift vertical delta vectors
17     VN ← X & HP
18     VP ← HN | ~ (X | HP)

19     if pos ≤ m − c then                        // scoring and output
20         errors ← errors + 1 − ((D0 ≫ (w − 1)) & 1)
21     else
22         s = (w − 2) − (pos − (m − c + 1))
23         errors ← errors + ((HP ≫ s) & 1)
24         errors ← errors − ((HN ≫ s) & 1)
25     if pos ≥ m − c and errors ≤ k then
26         report occurrence ending at pos
```

Before the pattern bitmasks can be used to compute the next D vector, they need to be shifted by 1 bit to the right and for $pos + c < m$ in one bitmask bit w must be set to represent the next pattern character $p[pos+c]$. This is done in line 11. Eventually, *errors* must be tracked properly. As long as the second last cell of D is within the DP matrix the last bit of $D0$ is used to track *errors* down the left-most band diagonal in line 20. Otherwise, the horizontal deltas of the last matrix row are used in lines 23 and 24 to update *errors*.

To speed up the verification of false positive match candidates, we use the property that cell values along a DP diagonal are monotonically increasing from top to bottom and

errors can only decrease along the last matrix row. The last row contains $n + c - m$ band cells and thus the search can be stopped if $errors > k + n + c - m$.

We compared our algorithm with an unbanded implementation of Myers' [1999] algorithm combined with the optimization proposed by Ukkonen [1985], and banded and unbanded implementations of Sellers [1980] algorithm. The average verification times per pattern character on DNA and ASCII alphabets are shown in Figure 6.9. Hyyrö's algorithm could not be compared as an implementation was not publicly available.

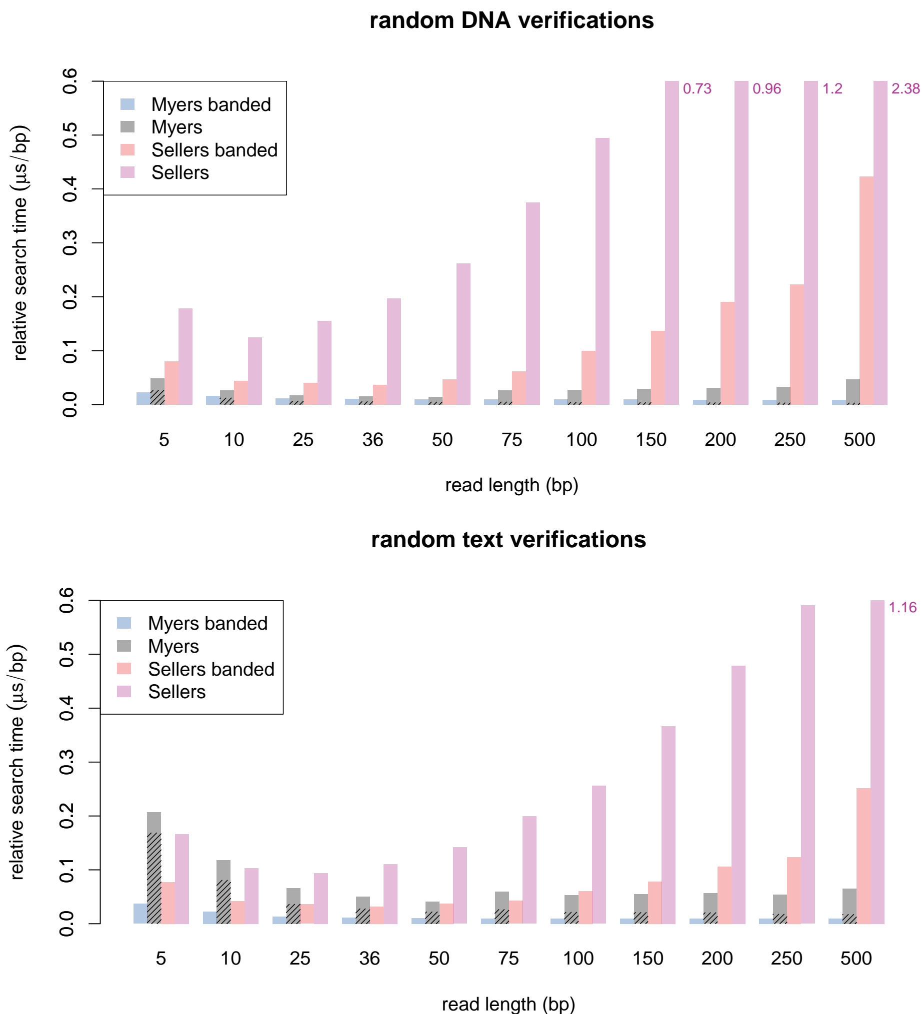


Figure 6.9: *Average verification time per read character required to verify a genomic substring of* 110 % *the read length. We compared banded and unbanded variants of the algorithms by Myers [1999]; Sellers [1980] on different alphabets* $|\Sigma| \in \{5, 256\}$ *and read lengths. The banded algorithms use a band width of* 10 % *the read length. The gray bars are split into preprocessing (shaded) and search (unshaded) times of Myers' algorithm.*

Algorithm 6.3: BANDEDMYERS_LARGEALPHABET(t, p, k, w, c)

input : text $t \in \Sigma^*$, pattern $p \in \Sigma^*$, errors k and band parameters w, c
output : text end positions of matches with up to k errors

```
1  foreach x ∈ Σ do                                   // initialize pattern bitmasks
2      B[x] ← 0^w
3      S[x] ← 0
4  for j ← max(c − w, 0) to c − 1 do
5      B[p[j]] ← B[p[j]] | 0^(c−j−1) 1 0^(w−c+j)
6  ⋮
7  for pos ← 0 to n − 1 do
8      if pos + c ≤ m then                       // shift and update pattern bitmasks
9          B[p[pos + c]] = (B[p[pos + c]] ≫ (pos − S[p[pos + c]])) | 10^(w−1)
10         S[p[pos + c]] = pos
11     B = B[t[pos]] ≫ (pos − S[t_pos])
12     X ← B | VN                                 // compute horizontal delta vectors
13     ⋮
```

Optimization for large alphabets. As shifting all pattern bitmasks in every sliding step takes $\mathcal{O}(|\Sigma|n)$ time[1], Algorithm 6.2 should only be used for small alphabets, e.g. the DNA alphabet. For large alphabets, a small adaptation shown in Algorithm 6.3 can be made that results in an overall running time of $\mathcal{O}(m + n + |\Sigma|)$[1]. In every step only one pattern bitmask, i.e. $B[p[pos + c]]$, is required to compute $D0$ and in only one bitmask, i.e. $B[t[pos]]$, bit w is set. We can omit to shift all other bitmasks by recording the number of yet to be conducted bit shifts in S and perfom the omitted shifts at once before reading (line 11) or updating (line 9) a pattern bitmask.

6.7 Paired-end mapping

In the paired-end sequencing protocol, DNA is fractionated into double-stranded fragments having lengths within a certain interval, e.g. 200–500 bp in the Illumnia protocol. Each fragment is sequenced from both 5'-ends and yields a pair of two reads $(l, r) \in \Sigma^* \times \Sigma^*$, see Figure 6.10. To determine the genomic origin of a read pair (l, r), we consider the two sets of all matches of l and r and call a pair of matches from both sets *valid* if the following holds:

1. l and r align with up to k errors each to opposite strands of the reference sequence.

2. They are aligned in correct orientation, i.e. the 3'-ends of both matches point toward each other.

[1] assuming that w is constant

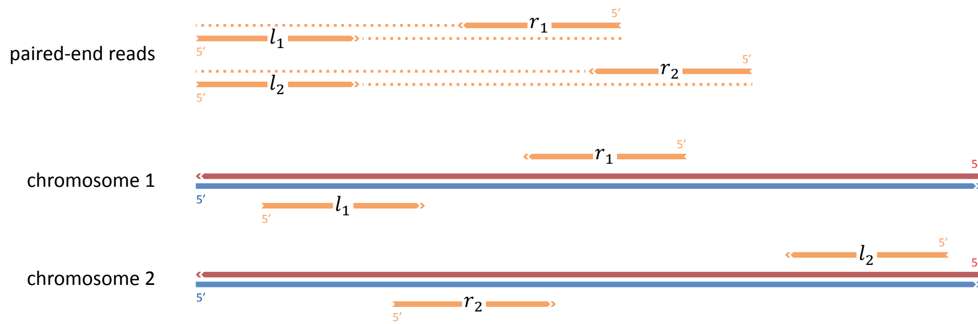


Figure 6.10: *Paired-end reads (l_i, r_i) are sequenced from both 5'-ends of double-stranded DNA fragments. As in most sequencing technologies the originating strand of a mate is unknown, pair matches must be searched on both strands.*

3. The library size is retained, i.e. given a mean library size μ and a tolerated deviation δ, the genomic distance d of both 5'ends is in the interval $d \in [\mu - \delta..\mu + \delta]$.

In most paired-end protocols the originating strand of a read is unknown. Hence, there are two possible orientations a valid pair match can have, see Figure 6.10.

We extended RazerS to search all valid pair matches. Given a set of paired-end reads $\mathcal{R} \subset \Sigma^* \times \Sigma^*$, it scans the reference genome from left to right in parallel with two filters having the distance of the minimal tolerated distance $\mu - \delta$. Each filter searches for match candidates of one of the two reads of all pairs. In order to be able to scan the same strand with the two filters, we reverse-complement the right reads first. Additionally, we record in a queue all preceding matches of the left filter within a distance of 2δ. Only if the right filter finds a match candidate whose mate match candidate is stored in the queue both candidates are verified. This guarantees that verifications are only done if both candidates are within the correct distance. If for a right read match multiple enqueued left read matches exist, we select matches with a minimal number of errors and among these the one with a minimal deviation from the library size μ.

As an optimization we use a lookup table to determine in constant time whether at least one match candidate of a left read is contained in the queue and link all candidates of the same read. Each candidate is verified one time at most and negatively verified candidates are removed from the linked list.

6.8 Match processing

The overlapping parallelograms of the SWIFT filter or the multiple seeds the pigeonhole filter may find in a single read match, result in multiple identical or nearly identical matches found in the verification step. To filter these *duplicates*, we regularly search for matches of the same read that have an identical begin or end position and keep only those with a minimal number of errors. Additionally we use a heuristic in the pigeonhole filter, that for multiple seeds on the same diagonal only one candidate region is generated.

If the user specifies a maximal number M of matches per read, we sort all matches ascendingly by the number of errors and remove all but the first M matches of each read.

For a read, the number of errors e in the M-th match is used to dynamically adjust the filter and verifier in order to search only for matches with less than e errors. If e equals 0 the read can be disabled completely.

There is another general ambiguity inherent to alignments with gaps that is related to the question which matches are different. Due to indels, a match with e errors, where $e < k$, has adjacent matches with at most $e+1, e+2, \dots, k$ errors. An intuitive approach would be to output only matches with a local minimum of errors. However there can be multiple local minima even separated by matches with more than k errors that share the same alignment trace. In [Holtgrewe *et al.*, 2011] we approach this problem more formally and require that two distinct matches have disjunctive optimal alignment traces and are separated by optimal alignments with more than k errors. We implemented these criteria in the verification step such that RazerS outputs all distinct matches. For more details, we refer to [Holtgrewe *et al.*, 2011].

6.9 Parallelization

We parallelized RazerS using OpenMP [Chandra *et al.*, 2001], a C++ language extension for shared memory parallel programming. To map reads in parallel with a fixed number of t threads, the set of reads is divided into t subsets of equal size which are statically assigned one-to-one to each thread. Each thread uses its own q-gram index and filter to search candidate matches of the read subset. We use open addressing indices (Section 5.4) which in total require $\mathcal{O}(|\mathcal{R}|)$ memory independent of the number of threads.

The parallel mapping of reads is performed window-wise in alternating phases of filtration and verification. We therefore partition both strands of the reference sequences into windows of equal length W, e.g. $W = 500{,}000$ bp. The threads simultaneously start filtering the first window of one strand and collect match candidates. Each thread that completed the filtration of a window divides the set of found candidates into packages of configured size, e.g. 100 candidates, or larger if the number of packages would exceed a maximal number, e.g. 100 packages. The packages are appended to a global queue (green boxes in Figure 6.11).

To realize a dynamic load-balancing, we allow threads that are behind others with filtration to immediately proceed with the next window. All other (leading) threads verify enqueued packages until the queue is empty. Found matches (blue boxes in Figure 6.11) are returned to the thread that enqueued the package, which filters duplicates and improves filtration parameters for reads with a sufficient number of matches (see Section 6.8). To save memory, the threads regularly append their found matches to a global external memory array of matches.

6.10 Experimental results

To evaluate the performance of RazerS, we conducted a number of experiments on simulated and real-world data and compared it with the best-mappers Bowtie 2, BWA, and

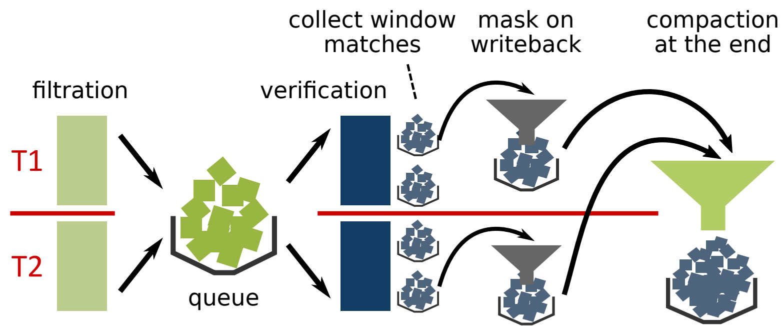


Figure 6.11: *Parallelization in RazerS. The set of reads is equally distributed over the set of threads T_i. Each thread filters the reference window-wise and adds verification packages (green) to a queue. Before processing the next window leading threads verify packages from the queue with dynamic load balancing. Duplicate matches are masked by each thread and the remaining matches (blue) are appended to a global array of matches.*

Soap 2 as well as the all-mappers Hobbes, mrFAST, and SHRiMP 2. For running time comparison, we ran the tools with 12 threads and used local disks for I/O. We used default parameters, except where stated otherwise. Read mappers that accept a maximal number of errors (mrFAST, Hobbes, Soap 2) were configured with the same error rate as RazerS. For a fair comparison with best-mappers, we configured RazerS in a second variant to also output one best match per read. See Section A.3 for the exact parametrization.

The involved real-world read sets are published in the European Nucleotide Archive [Leinonen *et al.*, 2011] and are given by their SRA/ENA id. As references we used whole genomes of E. coli (NCBI NC_000913.2), C. elegans (WormBase WS195), D. melanogaster (FlyBase release 5.42), and human (GRCh37.p2). The mapping times were measured on a cluster of nodes with 72 GB RAM and 2 Intel Xeon X5650 processors (each with 6 cores) per node running Linux 3.2.0.

6.10.1 Comparing the SWIFT and pigeonhole filters

RazerS provides support for two string metrics (Hamming and edit distance) and two filter variants (SWIFT and pigeonhole filter). To investigate which filter performs best on which kind of input and metric, we conducted an experimental evaluation of the time required to map different real datasets for varying mapping settings.

For this reason, we compared the mapping times of both filters and ran RazerS with 100 % and 99 % sensitivity for reads of lengths 30, 50, 70, and 100 bp for the references of E. coli, C. elegans, and chr. 2 of human with error rates between 0 and 10 % using Hamming and edit distance. To reduce influences from the operating system we measured the running times excluding I/O.

Figure 6.12 shows the running time ratios between mapping with the pigeonhole and SWIFT filter, where blue cells indicate a faster pigeonhole filter. We observe that for edit distance, the pigeonhole filter always leads to shorter running times than the SWIFT filter. For Hamming distance, the pigeonhole filter is well suited for low error rates (up to

reference	read length	read set ID	original length
E. coli	100	ERR022075	100
E. coli	70	ERR022075	100
E. coli	50	ERR022075	100
E. coli	30	ERR032371	36
C. elegans	100	SRR065390	100
C. elegans	70	SRR065390	100
C. elegans	50	SRR065390	100
C. elegans	30	SRR107574	34
human chr. 2	100	ERR012100	101
human chr. 2	70	SRR029194	88
human chr. 2	50	SRR029194	88
human chr. 2	30	ERR003244	37

Table 6.2: *This table gives information on which datasets were used when creating the experimental maps (Figure 6.12). If original length and read length m are not equal then the first m bases were used.*

6 %), while the SWIFT filter yields better mapping times for higher error rates. Astonishingly, the factors between the two methods range from 1:32 to 32:1. The differences in mapping times can be explained by the different characteristics of both filters. Compared to SWIFT, the simpler but less specific pigeonhole filter requires no counting and hence less processing overhead which compensates the increased number of verifications for low error rates. With an increase in error rate the specificity of both filters deteriorates equally for edit distance. For Hamming distance, gapped shapes compensate this degradation and make the SWIFT filter much more specific than the pigeonhole filter which is based on ungapped q-grams.

6.10.2 Analyzing the sensitivity estimation accuracy

We verify the correctness of both filtration sensitivity estimations by assessing the discrepancy between estimated and empirical sensitivity for the following two scenarios, where the first one serves as a sanity check:

(1) *Simulated data.* We simulate DNA sequence reads using position-dependent error probabilities and group them according to the number of implanted errors. After mapping the reads to the reference sequence we define the empirical sensitivity for each group as the proportion of reads that could be mapped back to their genomic origin. Using the same error distribution as for simulation, we compute the estimated sensitivity as described in Section 6.5.1.

(2) *Real data.* We map the set of reads once with 100 % sensitivity and keep as reference matches only those reads that map uniquely. Again, we group them according to the number of errors and determine the empirical sensitivity as for simulated data. The expected sensitivity is computed using the a posteriori probabilities (Section 6.5.3).

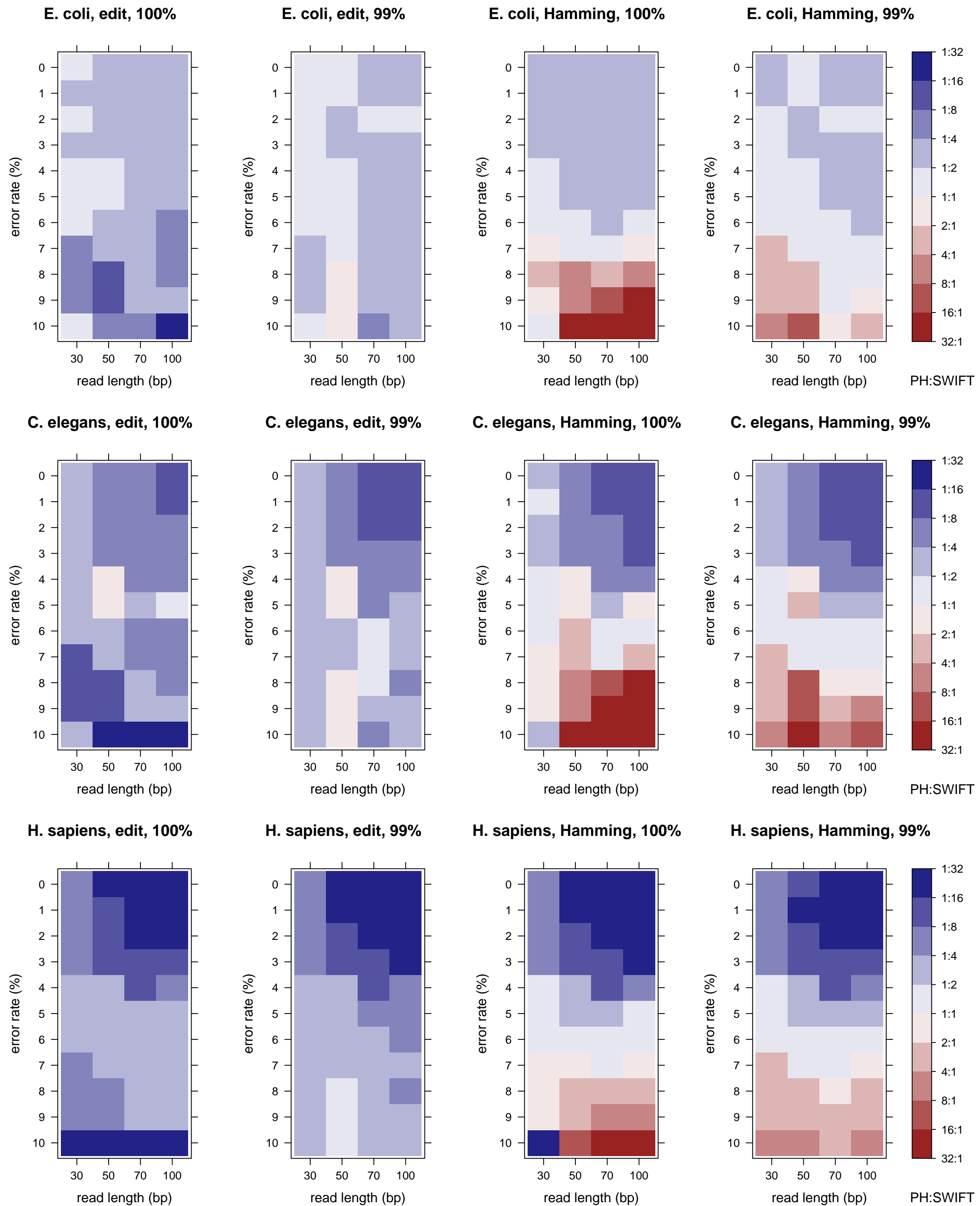


Figure 6.12: *Experimental map for reference sequences of E. coli, C. elegans, and chr. 2 of human. For read sets from different organisms we compared the time between mapping with the pigeonhole and SWIFT filter, while varying read length, string metric, error rate, and sensitivity. Only the mapping time was measured to eliminate the variance of the I/O time on the cluster as much as possible. RazerS was run with 12 threads. The color of each cell indicates the ratio of the running time between the pigeonhole and the SWIFT variant, where the pigeonhole variant was faster for blue cells. Ratios less/greater than 1:32/32:1 are plotted as 1:32/32:1.*

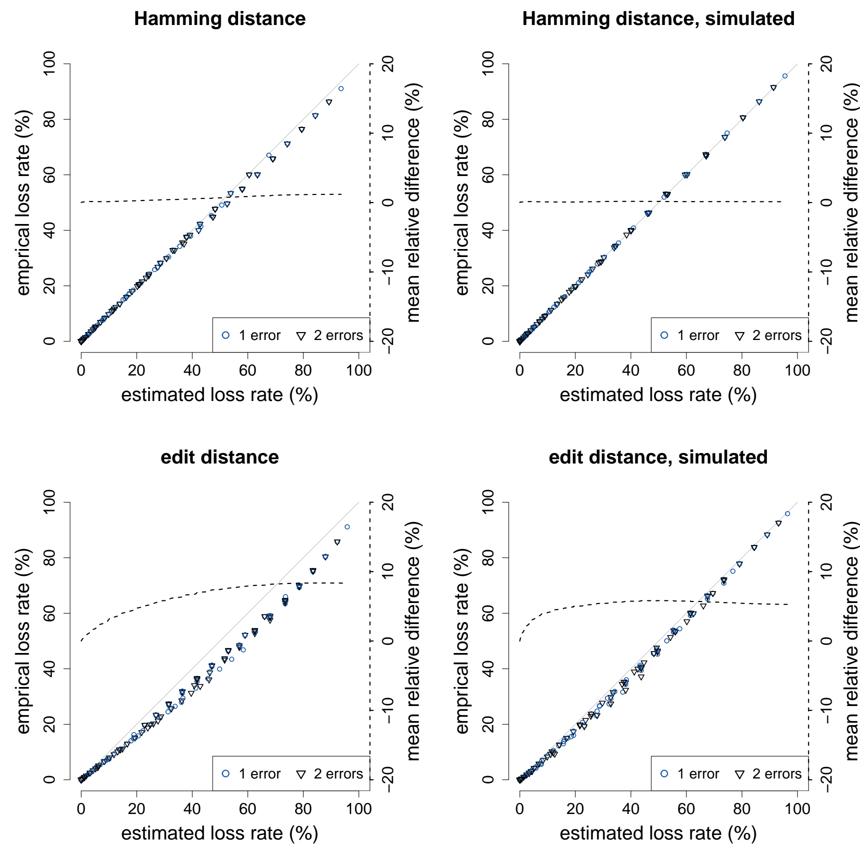


Figure 6.13: Comparison of estimated and empirical SWIFT filter loss rates (loss rate $=$ $1 -$ sensitivity) varying weight $q = 8, \ldots, 14$ and threshold $t = 1, \ldots, 20$. The dashed line reflects the mean of relative differences $1 - \frac{\text{empirical loss rate}}{\text{estimated loss rate}}$ of all estimated loss rates below a varying level.

SWIFT filter

Using the two protocols, we first examined the SWIFT filter sensitivity for simulated 36 bp reads and for a subset of the 36 bp reads in SRR001815. We inspected both Hamming as well as edit distance sensitivity and did the mapping for all ungapped shapes of weight q where $8 \leq q \leq 14$ and all thresholds t where $1 \leq t \leq 20$.

As a measure of accuracy we use the relative difference between empirical and estimated loss rate. We observe a very good agreement for Hamming distance (see Figure 6.13). For expected loss rates between 0 and 10 % the mean relative difference for simulated as well as real data is below 0.1 %. Considering edit distance, the expected loss rates overestimate the empirical loss. Between 0 and 10 % of expected loss the mean relative difference is below 4 % for simulated and below 2.8 % for real data. For edit distance the SWIFT parallelograms are broader and produce more random q-gram hits compared to Hamming distance mapping where the parallelograms are single diagonals. This leads to more matches than expected. The discrepancy for simulated data is slightly more pro-

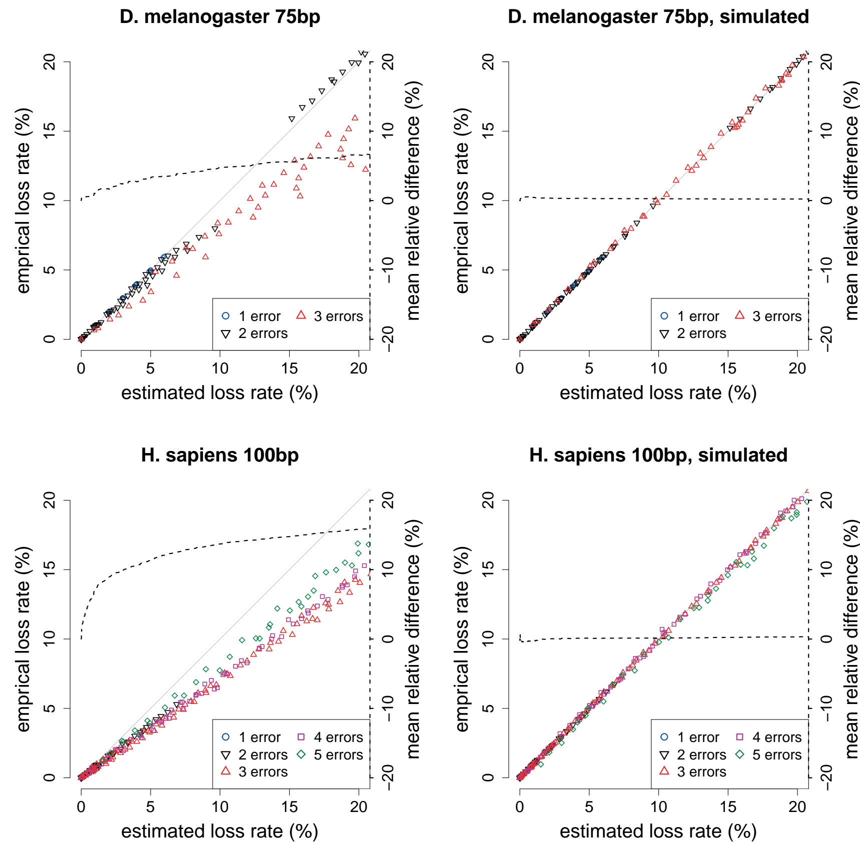


Figure 6.14: Comparison of estimated and empirical pigeonhole filter loss rates varying seed length $q = 16, \ldots, 31$ and the seed overlap $q - \Delta = 0, \ldots, 10$. Analogously to Figure 6.13, the dashed line reflects the mean of relative differences.

nounced. This can be explained by the observation that simulated matches are not necessarily optimal, i.e. reads can be mapped with less errors than originally implanted (e.g. an insertion next to a deletion will be aligned as one replacement). Most notably, in all cases the empirical sensitivities are higher than expected, thereby yielding better mapping results than estimated.

Pigeonhole filter

Second, we repeated the two protocols for the pigeonhole filter for simulated and real 10 M fly reads of length 75 bp (SRR060093) and 10 M human reads of length 100 bp (ERR012100). Therefore, we mapped the read sets with edit distance and used a (q, Δ)-seed filter while varying the q-gram length $q = 16, \ldots, 31$ and the q-gram overlap $q - \Delta = 0, \ldots, 10$.

The results in Figure 6.14 show a high level of agreement for simulated reads with a mean relative difference below 1 % for loss rates between 0 and 10 %. On real data the predicted loss rates between 0 and 10 % show a mean relative difference of 3 % on the

fly and 14 % on the human read set. We explain this deviation by an observed correlation of sequencing errors at adjacent positions towards the end of the read, whereas our model assumes independence of errors. This error correlation has also been observed in [Dohm *et al.*, 2008] and may be the result of molecules which are out of phase for multiple cycles in the sequencing process and lead to interferences with signals of adjacent bases. However, this correlation shows no negative influence as in none of our experiments the effective sensitivity was overestimated by our model.

6.10.3 Achieved speedup

To evaluate how much our implementation benefits from widely available multi-core architectures, we mapped a relatively large dataset (10 M reads of set ERR012100) against chr. 2 of the human genome. We ran RazerS with 1, 2, 4, and 8 threads (*dynamic* load balancing). The results were compared with the trivial parallelization (*static* load balancing) of splitting the read set into t parts of equal size and running t separate RazerS processes in parallel that use one thread each.

Both the runs with dynamic and static load balancing required about 89.5 min with one thread. Mapping reads with dynamic load balancing scaled almost linearly with speedups of 1.95, 3.95, and 7.46 for 2, 4, and 8 threads. Static load balancing scaled worse: The speedups were 1.90, 3.63, and 6.61. With 8 threads we effectively gained one more processor core with our dynamic balancing scheme compared to the static load balancing.

6.10.4 Rabema benchmark results

Next, we used the Rabema benchmark [Holtgrewe *et al.*, 2011] (v1.1) for a thorough evaluation and comparison of read mapping sensitivity. With Mason [Holtgrewe, 2010] we simulated 100 k reads of length 100 bp from the whole human genome and distributed sequencing errors like in a typical Illumina experiment (default settings). In the following, we will denote RazerS in edit distance mode using the pigeonhole filter with a sensitivity rate of x percent as R-x, e.g. R-100 for full sensitivity.

The benchmark contains the categories *all*, *all-best*, *any-best*, and *recall* and was performed with a maximal error rate of 5 %. In the all-category all matches with up to 5 % errors have to be found. In the categories any-best and all-best a mapper has to find for each read any or all matches with minimal edit distance. The category recall requires a mapper to find the *original* location of each read, which is a measure independent of the used scoring model (edit-distance or quality-based). For each category and mapper the Rabema benchmark determines the average fraction of found matches per read.

To compare the sensitivity fairly, we configured read mappers as best-mappers and as all-mappers if possible (BWA, Bowtie 2, and RazerS). We parametrized the best-mappers for high sensitivity and multiple matches. We do not consider running time here, since best-mappers are not designed for finding all matches and consequently consume more time (up to 3 hours in a run compared to several minutes). The aim here was to investigate sensitivity and recall.

	method	all	all-best	any-best	recall
best-mappers	Bowtie 2	92.04 99.18 98.72 96.80 93.44 81.94 40.19	96.16 97.79 97.85 95.80 94.83 93.37 88.86	98.08 100.00 99.96 97.55 96.62 94.93 90.46	95.94 98.01 97.72 95.55 94.24 92.79 89.52
	BWA	92.18 99.18 98.72 97.81 94.25 80.92 37.65	96.81 97.79 97.87 97.88 96.59 92.63 83.47	98.81 100.00 99.95 99.81 98.55 94.28 85.37	96.41 97.93 97.69 97.25 95.77 91.98 84.61
	Soap 2	65.93 99.18 95.55 91.34 8.67 0.70 0.00	69.89 97.79 94.74 91.37 8.98 0.79 0.00	71.37 100.00 96.78 93.18 9.21 0.81 0.00	69.91 98.05 94.62 91.20 11.85 1.41 0.36
	R-100	93.30 99.18 98.73 97.93 95.60 85.81 44.15	97.96 97.79 97.88 98.03 98.00 98.27 97.93	100.00 100.00 100.00 100.00 100.00 100.00 100.00	97.80 98.00 97.85 97.75 97.65 97.70 97.69
	R-95	93.10 99.18 98.73 97.93 95.49 84.76 42.82	97.75 97.79 97.88 98.03 97.88 97.03 94.97	99.79 100.00 100.00 100.00 99.89 98.74 97.00	97.60 98.03 97.85 97.74 97.52 96.56 94.99
all-mappers	Bowtie 2	95.69 99.98 99.91 99.45 97.99 90.69 55.14	98.85 99.74 99.79 98.61 98.21 97.55 93.84	99.16 100.00 99.98 99.01 98.63 97.94 94.17	98.54 99.74 99.58 98.27 97.64 96.87 94.40
	BWA	95.89 99.96 99.88 99.49 97.13 87.79 64.11	97.98 98.81 99.01 99.02 97.83 93.95 85.20	98.82 100.00 99.95 99.82 98.56 94.34 85.37	97.80 99.03 98.96 98.75 97.35 93.43 86.36
	Hobbes	96.56 99.41 99.00 98.76 97.80 93.20 73.05	97.08 97.23 96.59 97.01 97.38 98.16 97.42	98.01 97.92 97.51 97.96 98.43 99.12 98.46	96.41 95.49 95.84 96.54 97.03 97.98 97.79
	mrFAST	99.97 100.00 100.00 100.00 100.00 99.99 99.53	99.97 100.00 100.00 100.00 100.00 100.00 99.10	99.97 100.00 100.00 100.00 100.00 100.00 99.13	99.97 100.00 100.00 100.00 99.99 100.00 99.18
	SHRiMP 2	96.53 99.87 99.82 99.53 98.37 92.58 64.63	99.50 99.34 99.50 99.60 99.64 99.65 98.32	99.85 99.87 99.90 99.91 99.89 99.84 98.57	99.25 99.35 99.30 99.24 99.30 99.09 98.48
	R-100	100.00 100.00 100.00 100.00 100.00 100.00 100.00	100.00 100.00 100.00 100.00 100.00 100.00 100.00	100.00 100.00 100.00 100.00 100.00 100.00 100.00	100.00 100.00 100.00 100.00 100.00 100.00 100.00
	R-95	99.54 100.00 100.00 100.00 99.89 98.67 95.11	99.79 100.00 100.00 100.00 99.89 98.71 96.96	99.79 100.00 100.00 100.00 99.89 98.74 97.00	99.79 100.00 100.00 100.00 99.89 98.77 97.17

Table 6.3: *Rabema benchmark results. Rabema scores in percent (average fraction of matches found per read). Large numbers are the total scores in each Rabema category and small numbers show the category scores separately for reads with* $\left(\begin{smallmatrix}0&1&2\\3&4&5\end{smallmatrix}\right)$ *errors.*

The results are shown in Table 6.3. As expected, the all-mappers generally perform better than the best-mappers. Also, as expected, mappers lose more of the high-error locations than low-error locations. Surprisingly, Bowtie 2 and BWA are better than the all-mapper Hobbes. Soap 2 is low sensitive to reads with more than 2 errors as it allows no indels and at most 2 mismatches in total. By chance it aligns some of the reads with more errors as it replaces each `N` in the reads by a `G`. R-100 is the most sensitive method, followed by mrFAST (which is not fully sensitive for higher error rates), R-95, SHRiMP 2, and Bowtie 2. Even when configured as a best-mapper (i.e. only reporting one best match), RazerS achieves the best scores.

6.10.5 Variant detection results

The next experiment analyzes the applicability of RazerS and other read mappers in sequence variation pipelines. Similarly to the evaluation in [David *et al.*, 2011], we generated 5 million read pairs of length 2×100 bp with sequencing errors, SNPs, and indels from the whole human genome such that each read has an edit distance of at most 5 to its genomic origin. To distribute sequencing errors according to a typical Illumina run, we used the read simulator Mason with the default profile settings. The reads (pairs) were grouped according to the numbers of contained SNPs and indels, where the group (s, i) consists of reads (pairs) with s SNPs and i indels in total. We mapped the reads both as single and paired-end reads and measured the sensitivities separately for each class and read mapper.

A read (pair) was mapped *correctly* if an alignment (paired alignment) has been found within 10 bp of the genomic origin. It is considered to map *uniquely* if only one alignment

was reported by the mapper. For each class we define *recall* to be the fraction of all contained reads (pairs) and *precision* the fraction of uniquely mapped reads (pairs) that were mapped correctly. Table 6.4 shows the results for each read mapper and class for single-end (Table 6.4a) and paired-end reads (Table 6.4b). An extended version of this table is given in Section A.4 on page 158.

Comparing the all-mappers results, R-100 shows the highest recall and precision values on both the single and paired-end datasets. mrFAST is also full sensitive on the single-end dataset but has a low recall value of 8 % for pairs with 5 bp indels. SHRiMP 2 shows full precision in all classes and experiments but misses some non-unique alignments. Hobbes appears to have problems with indels and shows the lowest sensitivities in the all-mapper comparison.

Surprisingly, R-100 is the most sensitive best-mapper even in the non-variant class (0,0) where the simulated qualities could possibly give quality-based mappers an advantage. For paired-end reads where matches are also ranked by their deviation from the library size, R-100 is even more sensitive than the all-mappers Hobbes and mrFAST. As observed in [David *et al.*, 2011], quality-based mappers like Bowtie 2, BWA, and Soap 2 are not suited to reliably detect the origin of reads with variants. Their recall values deteriorate with more variants as they prefer alignments where mismatches can be explained by sequencing errors instead of natural sequence variants. The low sensitivity of Soap 2 is again due to its limitation to at most 2 mismatches.

6.10.6 Performance comparison

In the last experiment, we compare the real-world performance of RazerS with other read mappers. To this end, we mapped four different sets of 10 million Illumina read pairs of length 2 × 100 bp from E. coli, C. elegans, fly, and human, as well as six simulated datasets consisting of 1 million simulated read pairs of length 2 × 200 bp, 2 × 400 bp, and 2 × 800 bp from fly and human to their reference genomes. We mapped the reads both as single and paired-end reads with 4 % error rate and measured running times, peak memory consumptions, mapped reads (pairs), and reads (pairs) mapped with minimal edit distance. We compared RazerS in default mode with other all-mappers and configured it to output only one best match per read for the best-mapper comparison. Since mrFAST supports no shared-memory parallelization, we split the reads into packages of 500 k reads and mapped them with 12 concurrent processes of mrFAST. Hobbes' large memory consumption also required to map the reads package-wise but with a single process and 12 threads.

For the evaluation we use the commonly used measure of percentage of *mapped reads (pairs)*, i.e. the fraction of reads (pairs) that are reported as aligned in the result file of the mapper. However, as some mappers report alignments without constraints on the number of errors, we also determine the fraction of reads (pairs) whose best match has an error rate of at most 0 %, ..., 4% (small numbers in the mapped reads (pairs) column in Tables 6.5 and 6.5b).

We call a read (pair) *ε-mappable*, if it is aligned with an error rate of ε by at least one mapper and ε is the smallest such value. A mapper *correctly maps* an ε-mappable read

	method	(0,0)		(2,0)		(4,0)		(1,1)		(1,2)		(0,3)	
		prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.
best-mappers	Bowtie 2	97.6	97.3	94.6	92.0	92.6	82.5	95.3	93.3	93.5	92.3	96.1	95.4
	BWA	98.2	97.9	97.6	95.3	94.9	85.1	97.4	90.9	97.1	80.3	96.3	66.5
	Soap 2	98.1	82.9	97.4	31.0	0.0	0.0	90.6	6.2	0.0	0.0	0.0	0.0
	R-100	98.4	98.4	98.2	98.2	96.3	96.3	98.1	98.1	97.9	97.9	97.6	97.6
	R-95	98.4	98.3	98.2	97.3	96.1	91.7	98.2	97.6	97.9	97.6	97.5	97.5
all-mappers	Hobbes	99.9	99.9	99.9	99.9	100.0	100.0	100.0	99.8	100.0	93.6	99.6	90.5
	mrFAST	100.0	99.9	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	SHRiMP 2	100.0	99.4	100.0	99.7	100.0	99.7	100.0	99.5	100.0	99.2	100.0	99.6
	R-100	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	R-95	100.0	99.9	100.0	99.0	100.0	95.4	100.0	99.4	100.0	99.6	100.0	99.9

***(a)** single-end reads*

	method	(0,0)		(4,0)		(8,0)		(2,2)		(2,4)		(0,5)	
		prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.
best-mappers	Bowtie 2	98.8	98.8	98.3	96.4	100.0	92.0	98.5	97.6	98.3	97.5	98.5	98.3
	BWA	99.0	98.4	99.2	93.5	100.0	80.0	98.5	79.9	100.0	65.2	99.1	69.0
	Soap 2	98.7	95.5	98.9	53.2	0.0	0.0	97.3	53.3	96.6	46.3	98.5	79.7
	R-100	99.0	99.0	99.0	99.0	100.0	100.0	99.7	99.7	99.6	99.6	99.0	99.0
	R-95	99.0	98.9	99.1	96.3	100.0	88.0	99.1	97.9	100.0	99.2	98.7	98.6
all-mappers	Hobbes	97.5	93.2	98.0	94.6	100.0	100.0	96.5	80.1	99.5	86.0	97.7	85.2
	mrFAST	98.8	98.8	98.9	98.9	100.0	100.0	99.1	99.1	98.8	98.8	91.8	7.8
	SHRiMP 2	100.0	99.7	100.0	99.9	100.0	100.0	100.0	99.7	100.0	99.6	100.0	99.7
	R-100	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	R-95	100.0	99.9	100.0	97.3	100.0	88.0	100.0	98.8	100.0	99.2	100.0	99.9

***(b)** paired-end reads*

Table 6.4: *Variant detection results. For single-end (a) and paired-end reads (b) we show the percentages of found origins (recall) and fraction of unique reads mapped to their origin (precision) grouped by reads with s SNPs and i indels* (s, i).

(pair), if it reports at least one alignment with an error rate of ε. For each mapper we measured the percentage of *correctly mapped reads (pairs)*, i.e. the fraction of ε-mappable reads (pairs) for $\varepsilon \in [0, 4\%]$ that are correctly mapped. For a more detailed analysis we additionally give the percentages separately for sets of $\varepsilon = 0$, $\varepsilon \in (0, 1\%]$, ..., $\varepsilon \in (3, 4\%]$.

The results for the fly and human Illumina datasets as well as the simulated 800 bp fly dataset are shown in Tables 6.5a and 6.5b. More detailed tables of all datasets are given in Section A.5 on page 159.

As can be seen, R-100 aligns all reads with the minimal number of errors and achieves the best percentage of correctly mapped reads followed by R-95 in all experiments. A decrease in the specified sensitivity results in a decrease in running time and on the human genome R-95 is up to twice as fast as R-100. As in the previous experiments, the actual sensitivity is always higher than specified.

All-mapper comparison. For the single-end 100 bp datasets mrFAST is as sensitive but 4 times slower than R-100. On paired-end reads it is less sensitive and apparently has problems to map long reads with an increased number of absolute errors. In the results of the Illumina paired-end datasets we in fact noticed some alignments with more errors than asserted by mrFAST and an error rate above 4 %. Thus the number of totally

	method	SRR497711 D. melanogaster: time [min:s]	SRR497711: correctly mapped reads [%]	SRR497711: mapped reads [%]	ERR012100 H. sapiens: time [min:s]	ERR012100: correctly mapped reads [%]	ERR012100: mapped reads [%]	simulated, $m = 800$ D. melanogaster: time [min:s]	simulated: correctly mapped reads [%]	simulated: mapped reads [%]
best-mappers	Bowtie 2	2:00	99.65 (100.00 99.77 99.02 / 97.65 94.92)	85.71 (52.08 67.27 73.62 / 76.88 78.81)	5:37	99.62 (100.00 99.75 96.02 / 92.88 87.86)	96.72 (75.99 87.81 90.54 / 91.85 92.76)	13:48	96.73 (97.47 99.67 98.05 / 87.95 85.17)	99.99 (0.03 41.07 73.95 / 82.31 90.03)
best-mappers	BWA	5:35	98.96 (100.00 99.57 98.40 / 90.72 82.08)	79.37 (52.08 67.24 73.57 / 76.62 78.31)	13:45	99.66 (100.00 99.50 98.01 / 93.39 88.92)	93.53 (75.99 87.78 90.59 / 91.91 92.82)	5:38	74.96 (97.47 98.62 82.47 / 0.00 0.00)	68.09 (0.03 40.61 68.09 / 68.09 68.09)
best-mappers	Soap 2	1:55	91.78 (100.00 96.24 89.35 / 0.09 0.02)	72.49 (52.08 66.73 72.48 / 72.49 72.49)	2:34	96.45 (100.00 94.94 86.54 / 0.32 0.16)	89.73 (75.99 87.24 89.73 / 89.73 89.73)	0:54	41.21 (97.47 67.99 28.10 / 0.22 0.00)	38.14 (0.03 28.17 37.88 / 38.14 38.14)
best-mappers	R-100	1:28	100.00 (100.00 100.00 100.00 / 100.00 100.00)	78.92 (52.08 67.31 73.69 / 76.97 78.92)	85:56	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.99 (75.99 87.84 90.67 / 92.02 92.99)	1:17	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)
best-mappers	R-95	1:26	99.87 (100.00 100.00 100.00 / 99.11 96.34)	78.82 (52.08 67.31 73.69 / 76.94 78.82)	43:16	99.96 (100.00 100.00 100.00 / 99.30 96.92)	92.95 (75.99 87.84 90.67 / 92.01 92.95)	1:15	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)
all-mappers	Hobbes	4:51	96.49 (96.55 96.46 96.94 / 96.28 93.86)	76.16 (50.28 64.98 71.16 / 74.33 76.16)	265:48	95.97 (95.94 96.14 96.39 / 96.10 94.63)	89.24 (72.90 84.30 87.02 / 88.33 89.24)	–	–	–
all-mappers	mrFAST	4:01	100.00 (100.00 100.00 100.00 / 100.00 100.00)	78.92 (52.08 67.31 73.69 / 76.97 78.92)	413:40	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.99 (75.99 87.84 90.67 / 92.02 92.99)	5:16	65.25 (93.14 95.65 59.59 / 0.00 0.00)	69.32 (0.03 39.34 69.32 / 69.32 69.32)
all-mappers	SHRiMP 2	23:40	99.83 (99.99 99.99 99.74 / 98.71 96.33)	89.91 (52.07 67.30 73.66 / 76.92 78.83)	1312:09	99.81 (99.89 99.83 99.39 / 98.29 96.81)	99.06 (75.90 87.74 90.56 / 91.91 92.87)	796:06	95.70 (97.47 99.75 97.60 / 82.95 80.14)	99.31 (0.03 41.04 73.67 / 81.63 89.14)
all-mappers	R-100	1:51	100.00 (100.00 100.00 100.00 / 100.00 100.00)	78.92 (52.08 67.31 73.69 / 76.97 78.92)	118:26	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.99 (75.99 87.84 90.67 / 92.02 92.99)	1:20	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)
all-mappers	R-95	1:45	99.87 (100.00 100.00 100.00 / 99.11 96.34)	78.82 (52.08 67.31 73.69 / 76.94 78.82)	58:13	99.96 (100.00 100.00 100.00 / 99.30 96.92)	92.95 (75.99 87.84 90.67 / 92.01 92.95)	1:20	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)

***(a)** single-end reads*

	method	SRR497711 D. melanogaster: time [min:s]	SRR497711: correctly mapped pairs [%]	SRR497711: mapped pairs [%]	ERR012100 H. sapiens: time [min:s]	ERR012100: correctly mapped pairs [%]	ERR012100: mapped pairs [%]	simulated, $m = 800$ D. melanogaster: time [min:s]	simulated: correctly mapped pairs [%]	simulated: mapped pairs [%]
best-mappers	Bowtie 2	6:32	98.94 (100.00 98.82 96.96 / 95.26 90.21)	81.94 (32.50 60.48 69.88 / 72.47 72.94)	10:51	99.51 (99.97 99.80 97.70 / 94.37 84.85)	94.19 (15.04 77.57 85.16 / 86.58 86.89)	39:07	93.64 (– 99.20 93.91 / 83.32 73.14)	99.70 (0.00 24.15 57.94 / 69.85 71.10)
best-mappers	BWA	13:33	97.47 (100.00 98.48 91.02 / 82.51 68.36)	73.41 (32.51 60.41 69.30 / 71.57 71.92)	34:35	98.84 (99.99 99.66 93.72 / 84.75 63.84)	88.06 (15.04 77.50 84.86 / 86.16 86.39)	11:26	56.28 (– 95.85 49.28 / 0.00 0.00)	46.44 (0.00 23.32 40.44 / 40.44 40.44)
best-mappers	Soap 2	5:29	88.67 (100.00 93.05 59.12 / 17.90 0.01)	72.77 (32.58 59.65 65.93 / 66.51 66.62)	8:24	91.58 (99.99 97.68 43.05 / 9.61 0.01)	87.47 (15.07 77.33 81.46 / 81.70 81.77)	12:36	23.55 (– 49.58 13.91 / 0.002 0.00)	28.23 (0.00 12.38 17.64 / 17.83 18.00)
best-mappers	R-100	9:01	100.00 (100.00 100.00 100.00 / 100.00 100.00)	72.95 (32.50 60.63 70.04 / 72.52 72.95)	176:29	100.00 (100.00 100.00 100.00 / 100.00 100.00)	86.93 (15.04 77.65 85.27 / 86.62 86.93)	2:22	100.00 (– 100.00 100.00 / 100.00 100.00)	71.16 (0.00 24.22 58.38 / 70.03 71.16)
best-mappers	R-95	6:56	99.78 (100.00 100.00 99.28 / 97.44 93.24)	72.80 (32.50 60.63 69.98 / 72.39 72.80)	135:44	99.89 (100.00 100.00 99.47 / 97.74 91.70)	86.84 (15.04 77.65 85.23 / 86.55 86.84)	2:19	100.00 (– 100.00 100.00 / 100.00 100.00)	71.16 (0.00 24.22 58.37 / 70.02 71.16)
all-mappers	Hobbes	8:43	84.78 (84.27 86.02 84.71 / 78.85 77.84)	62.48 (27.39 51.81 59.99 / 62.08 62.48)	89:35	95.11 (95.68 95.57 92.20 / 85.12 89.86)	84.05 (14.39 74.46 81.95 / 83.53 84.05)	–	–	–
all-mappers	mrFAST	8:26	100.00 (100.00 99.99 99.99 / 99.99 99.98)	73.16 (32.50 60.63 70.04 / 72.52 72.95)	779:12	99.94 (99.98 99.96 99.82 / 99.56 98.73)	87.79 (15.04 77.64 85.26 / 86.61 86.91)	10:47	44.19 (– 91.35 27.29 / 0.00 0.00)	49.69 (0.00 24.50 43.35 / 43.35 43.35)
all-mappers	SHRiMP 2	47:07	99.67 (100.00 99.93 98.65 / 97.39 93.03)	87.36 (32.50 60.62 69.95 / 72.48 72.93)	2762:32	99.74 (99.91 99.88 99.07 / 97.44 90.67)	97.51 (15.03 77.57 85.15 / 86.53 86.83)	1617:26	91.64 (– 99.35 91.81 / 77.75 64.58)	98.62 (0.00 24.12 57.14 / 68.89 70.27)
all-mappers	R-100	7:59	100.00 (100.00 100.00 100.00 / 100.00 100.00)	72.95 (32.50 60.63 70.04 / 72.52 72.95)	184:27	100.00 (100.00 100.00 100.00 / 100.00 100.00)	86.93 (15.04 77.65 85.27 / 86.62 86.93)	2:30	100.00 (– 100.00 100.00 / 100.00 100.00)	71.16 (0.00 24.22 58.38 / 70.03 71.16)
all-mappers	R-95	7:36	99.78 (100.00 100.00 99.28 / 97.44 93.24)	72.80 (32.50 60.63 69.98 / 72.39 72.80)	166:22	99.89 (100.00 100.00 99.47 / 97.74 91.70)	86.84 (15.04 77.65 85.23 / 86.55 86.84)	2:29	100.00 (– 100.00 100.00 / 100.00 100.00)	71.16 (0.00 24.22 58.37 / 70.02 71.16)

***(b)** paired-end reads*

***Table 6.5:** Performance results of single-end (a) and paired-end (b) mapping. The left side shows the results for the first 10 M × 100 bp reads (pairs) of two Illumina datasets. The dataset on the right consists of 1 M × 800 bp simulated reads (pairs) with a stretched Illumina sequencing error profile. In large we show the percentage of totally mapped reads (pairs) and in small the percentages of reads (pairs) that are mapped with up to* $\left(\begin{smallmatrix} 0 & 1\% & 2\% \\ 3\% & 4\% & \end{smallmatrix}\right)$ *errors. Correctly mapped reads (pairs) show the fractions of reads (pairs) that were mapped with the overall minimal number of errors. There were none of the* 2×800 *bp pairs without error (denoted by a "–" in the 0-error class). Hobbes could not be run on reads longer than 100 bp.*

mapped pairs is slightly higher compared to R-100 on the Illumina paired-end reads. On single-end reads Hobbes is about 2 times slower and only on human paired-end reads faster (up to 2 times) than R-100. It maps 5–15 % less reads correctly and also the total number of mapped reads is less. Hobbes is not able to map reads longer than 100 bp and some single-end read packages could not be mapped due to repeated crashes (4 of 20 for C. elegans and 1 of 20 for human). As SHRiMP 2 does not use a maximal error rate it outputs more mapped reads than R-100 in total. However, the percentages of correctly mapped reads is less in all experiments. This could be due to its different scoring scheme, where two mismatches cost less than opening a gap, but it does not explain why it misses reads with 0 errors. SHRiMP 2 is 5–23 times slower than R-100 on the Illumina datasets and up to 600 times slower on the 800 bp datasets.

Best-mapper comparison. Compared to other best-mappers, R-95 is faster or comparably fast on all E. coli, C. elegans, and fly datasets. For human reads of length 100 bp or 200 bp it is 2–3 times slower than BWA and equally fast or faster for longer reads. BWA and Bowtie 2 could not be run with a maximal error rate and hence map more reads than R-100 in total, but less correctly (in terms of edit distance) as they optimize for errors at low-quality bases. With longer reads, BWA becomes less sensitive and BWA-SW might be the better choice. However, we could not compare BWA-SW as it does not align the reads from end to end. As seen before, Soap 2 is low sensitive to reads with more than 2 errors.

Memory requirement. The memory consumption of RazerS can be determined as follows: As all contigs of the reference genome are loaded and searched one after another, the size C of the largest contig is one summand. Another summand is the total number of bases in the input read set, e.g. $n \cdot m$ for n reads of length m. Each thread filters a subset of reads and uses a q-gram index whose size is linear in the number of contained q-grams. Therefore, the overall size of the indices is $\mathcal{O}(n \cdot (m - \max Q))$ when using a SWIFT filter with q-grams of shape Q, or $\mathcal{O}(n \cdot m/\Delta)$ when using a (q, Δ)-seed pigeonhole filter. Finally, enough space for the matches has to be allocated, which can be estimated by $\mathcal{O}(n \cdot \alpha)$ where α is the average number of matches per read. Hence the peak memory usage of RazerS is $\mathcal{O}(C + n \cdot (\alpha + m))$.

In practice, RazerS requires 9 or 15 GB for mapping 10 million reads of length 100 bp to the human genome in best-mode or all-mode. For the same input set, Bowtie 2 uses 3.3 GB, BWA uses 4.5 GB, Soap 2 uses 5.4 GB, SHRiMP 2 uses 38 GB. Due to the lack of parallelization or a high memory consumption we ran mrFAST and Hobbes on packages of 500 k reads where they required 11 GB and 70 GB of memory. Section A.5 on page 159 contains tables that also show the full memory requirements.

RazerS' memory consumption grows linearly with the number of reads and matches, i.e. about 10 GB are required for each additional 10 M $\times$ 100 bp reads. A large read set, e.g. an Illumina HiSeq run, can be mapped on clusters or low memory machines by splitting it into blocks of appropriate size and mapping them separately. The final mapping result can be obtained by concatenating the mapping results of each block.

Chapter 7 Frequency String Mining

In this chapter, we propose the *deferred frequency index* [Weese and Schulz, 2008], an application of the lazy suffix tree to efficiently solve arbitrary frequency based string mining problems in multiple databases.

After defining the string frequency, frequency predicates, and some well-known frequency string mining problems in Section 7.2, we introduce a novel discriminatory frequency predicate the *entropy substring mining problem* for multiple databases based on the notion of entropy from information theory. The predicate is motivated by the *emerging substring mining problem* that was introduced by Chan *et al.* [2003] for two databases. The idea is to find patterns that are representative for a small subset of databases, possibly one, and are absent in the rest of the databases.

We define the monotonic hull in Section 7.3 that allows to prune the set of suffix tree nodes to a required minimum. In the subsequent two sections, we first introduce the optimal algorithm by Fischer *et al.* [2006] and then give an in-depth presentation of our algorithm, which exploits the stability of counting sort used for the node expansion of the lazy suffix tree to compute the substring frequencies as a byproduct.

At last, we show in Section 7.6 how our approach can be applied to mine multiple databases with a variety of frequency constraints. We apply our new predicate and search for species specific protein domains in large protein databases. In experiments over a broad range of pattern domains and for different types of frequency string mining problems, we demonstrate that the deferred frequency index (DFI) is the fastest currently available algorithm for frequency based string mining. Although the algorithm of Fischer *et al.* [2006] has in theory a better memory consumption, we can show that our implementation uses in practice less memory. The two memory improved variants by Fischer *et al.* [2008] and Kügel and Ohlebusch [2008] use less memory in practice but are prohibitively slow or limited to conjunctive predicates, respectively.

7.1 Related work

There have been several approaches in the context of mining exact substrings with frequency constraints. Raedt *et al.* [2002] introduced the first $\mathcal{O}(n^3)$ algorithm based on the level-wise Apriori algorithm [Raedt *et al.*, 2002], where n is the total number of characters. This algorithm is not suitable for large databases due to repeated scanning of the whole databases. Subsequently, Lee and Raedt [2005] proposed to build a suffix trie

work	time	memory	multiple	pruning	optimal hull
Raedt *et al.* [2002]	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	-	•	-
Lee and Raedt [2005]	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	•	-	-
Chan *et al.* [2003]	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	-	•	-
Fischer *et al.* [2005]	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	•	-	-
Fischer *et al.* [2006]	$\mathcal{O}(n)$	$\mathcal{O}(n)$	•	-	-
DFI	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	•	•	•

Table 7.1: *Existing frequency string mining algorithms and their characteristics*

from the first database and stream the remaining databases against it. A more efficient algorithm based on suffix trees was suggested by Chan *et al.* [2003] incorporating three different pruning criteria. Fischer *et al.* [2005] were the first to use an enhanced suffix array (Chapter 3) of the concatenated databases.

Until 2006 all approaches calculated the frequencies of substrings in the databases in a naive manner. Fischer *et al.* [2006] developed a strategy to compute the frequencies of substrings in optimal time via range minimum queries [Fischer and Heun, 2006], which led to the first optimal $\mathcal{O}(n)$ time algorithm for frequency based string mining. The memory consumption of this approach was subsequently improved by Kügel and Ohlebusch [2008] and Fischer *et al.* [2008]. The details of the algorithms are explained in Section 7.4.

Fischer *et al.* [2006] achieved optimality at the expense of complicating the algorithm and adding another $\Theta(n)$ memory for every database. In addition, all algorithms of Fischer *et al.* need to construct the enhanced suffix array completely to calculate the frequencies of all substrings afterwards and cannot benefit from search space pruning.

Our approach proposed in [Weese and Schulz, 2008] can efficiently solve any frequency based string mining problem on an arbitrary number of databases. It is not only simple but also retains the problem-specific search space pruning of the algorithms by Raedt *et al.* [2002]; Chan *et al.* [2003], see Table 7.1. The frequencies are calculated during the construction of a suffix tree over all databases, which enables for the first time to limit the index construction to a problem-specific minimum referred to as the *optimal monotonic hull* (Section 7.3). Most of the previous approaches and problem definitions focussed on two databases (foreground and background database). In this work, we generalize the well-known *emerging substring mining problem* and devise a novel problem definition for mining multiple databases.

7.2 Definitions

In the following, we denote a non-empty set of strings $\mathcal{D} \subseteq \Sigma^*$ as *database* given an alphabet Σ. For arbitrary strings over Σ we define their frequency and support in a database.

Definition 7.1 (frequency and support)**.** Given a database $\mathcal{D}$. The *frequency* and the *sup-*

	ϕ	b	ba	bab	aab
$\mathcal{D}_1 = \{\text{abab, babb}\}$	$\text{freq}(\phi, \mathcal{D}_1, \dots \mathcal{D}_4)$	(2,2,2,2)	(2,1,2,1)	(2,0,2,0)	(0,2,0,0)
$\mathcal{D}_2 = \{\text{baab, aaab}\}$	$\text{supp}(\phi, \mathcal{D}_1)$	1	1	1	0
$\mathcal{D}_3 = \{\text{cbab, baba}\}$	$\text{supp}(\phi, \mathcal{D}_2)$	1	0.5	0	1
$\mathcal{D}_4 = \{\text{abac, bbbb}\}$	$\text{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi)$	1	2	∞	0
	$H(\phi, \mathcal{D}_1, \dots, \mathcal{D}_4)$	1	0.96	0.25	0

Table 7.2: *Example for the definition of* frequency vector*,* support*,* growth*, and* entropy*.*

port of a string $\phi \in \Sigma^*$ in $\mathcal{D}$ is defined as follows:

$$\text{freq}(\phi, \mathcal{D}) \quad := \quad \left|\left\{ d \in \mathcal{D} \;\middle|\; \phi \preceq d \right\}\right|, \tag{7.1}$$

$$\text{supp}(\phi, \mathcal{D}) \quad := \quad \frac{\text{freq}(\phi, \mathcal{D})}{|\mathcal{D}|}. \tag{7.2}$$

Obviously it holds $0 \leq \text{supp}(\phi, \mathcal{D}) \leq 1$ and the support can be considered as the normalized frequency. For multiple databases we define the frequency vector.

Definition 7.2 (frequency vector)**.** Given multiple databases $\mathcal{D}_1, \dots, \mathcal{D}_m$, the *frequency vector* of a string $\phi \in \Sigma^*$ is an element of $\mathbb{N}_0^m$ and defined as:

$$\text{freq}(\phi, \mathcal{D}_1, \dots, \mathcal{D}_m) \quad := \quad \Big(\text{freq}(\phi, \mathcal{D}_1), \dots, \text{freq}(\phi, \mathcal{D}_m)\Big). \tag{7.3}$$

On $\mathbb{N}_0^m$ we define a partial order "$\leq$", such that for two vectors $u, v \in \mathbb{N}_0^m$ holds $u \leq v \Leftrightarrow \forall_{i \in [1..m]} u_i \leq v_i$. Table 7.2 gives an example for the support value and frequency vector.

7.2.1 Predicates

A *frequency predicate* on a set of databases $\mathcal{D}_1, \dots, \mathcal{D}_m$ is defined as a boolean function over $\mathbb{N}^m$ with the additional constraint that the function yields false for the null vector. In general, our approach is applicable to the task of finding patterns $\phi \in \Sigma^*$ whose frequencies satisfy a predicate *pred* on a given database set $\mathcal{D}_1, \dots, \mathcal{D}_m$. We define the *solution set Th* as a function of *pred* [Raedt *et al.*, 2002]:

$$Th(pred) \quad = \quad \Big\{ \phi \in \Sigma^* \;\Big|\; pred\,(\text{freq}(\phi, \mathcal{D}_1, \dots, \mathcal{D}_m)) \text{ is true} \Big\}. \tag{7.4}$$

Please note that the solution set solely contains substrings of the given databases, as the null vector does not satisfy the frequency predicate. In the following, we consider three specific examples of frequency string mining problems and define their corresponding frequency predicates in the next section:

Problem 7.1 (frequent pattern mining problem)**.** *Given m databases* $\mathcal{D}_1, \dots, \mathcal{D}_m$ *of strings over* Σ *and* m *pairs of frequency thresholds* $(\text{min}_1, \text{max}_1), \dots, (\text{min}_m, \text{max}_m)$*, the* frequent pattern mining problem *is to find all strings* $\phi \in \Sigma^*$ *that satisfy* $\text{min}_i \leq \text{freq}(\phi, \mathcal{D}_i) \leq \text{max}_i$*, for all* $1 \leq i \leq m$*.*

This problem is conjunctive (explained in Section 7.2.3) and has been considered in a series of research papers [Fischer *et al.*, 2005; Raedt *et al.*, 2002; Lee and Raedt, 2005]. The next problem considers discriminatory strings for two databases $\mathcal{D}_1, \mathcal{D}_2 \in \Sigma^*$. $\mathcal{D}_1$ is usually called positive (foreground) set, where $\mathcal{D}_2$ is the negative (background) set. As a measure of difference the growth-rate from $\mathcal{D}_2$ to $\mathcal{D}_1$ for a string ϕ is defined as:

$$\text{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) \quad := \quad \begin{cases} \frac{\text{supp}(\phi, \mathcal{D}_1)}{\text{supp}(\phi, \mathcal{D}_2)}, & \text{if } \text{supp}(\phi, \mathcal{D}_2) \neq 0 \\ \infty, & \text{else.} \end{cases} \tag{7.5}$$

Problem 7.2 (emerging substring mining problem)**.** *Given a support condition ρ_s, where $\frac{1}{|\mathcal{D}_1|} \leq \rho_s \leq 1$, and a minimum growth rate $\rho_g > 1$, the* emerging substring mining problem *[Chan* et al.*, 2003] is to detect all strings $\phi \in \Sigma^*$, such that* $\text{supp}(\phi, \mathcal{D}_1) \geq \rho_s$ *and* $\textit{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\phi) \geq \rho_g$.

The minimum support rate ρ_s limits the solution space to representative strings of database $\mathcal{D}_1$, where ρ_g is a discrimination threshold. Patterns which satisfy the conditions of Problem 7.2 are called *emerging substrings*. If the growth rate of the pattern is infinite it is called *jumping emerging substring*, because it is a major discriminator between the databases under investigation. For example, Kobyliński and Walczak [2009] use *jumping emerging substrings* for image classification. The problem is *asymmetric* and the algorithm will return different outputs depending on the order of the input datasets. Compared to Problem 7.1, the *emerging substring mining problem* is more suitable to contrast data mining, as the growth rate reflects the specificity of a pattern for the foreground dataset more adequately than absolute frequency bounds.

Example 7.1. We now apply this problem to databases $\mathcal{D}_1$ and $\mathcal{D}_2$ given in Table 7.2 with $\rho_s = 1$ and $\rho_g = 2$. The corresponding frequency predicate *pred* for the *emerging substring mining problem* is a function that maps the frequency vector $(d_1, d_2) = \text{freq}(\phi, \mathcal{D}_1, \mathcal{D}_2)$ of a string $\phi \in \Sigma^*$ to a truth value as follows:

$$\begin{aligned} \textit{pred}(d_1, d_2) \quad &:= \quad (d_1 \geq \rho_s \cdot |\mathcal{D}_1|) \wedge (d_1 \cdot |\mathcal{D}_2| \geq \rho_g \cdot d_2 \cdot |\mathcal{D}_1|) \\ &= \quad (d_1 \geq 2) \wedge (d_1 \geq 2d_2). \end{aligned} \tag{7.6}$$

The set of patterns whose frequencies satisfy *pred* is $\textit{Th}(\textit{pred}) = \{\texttt{bab}, \texttt{ba}\}$. b for example is not an *emerging substring*, because $\text{supp}(\texttt{b}, \mathcal{D}_1) = 1$ but $\text{growth}_{\mathcal{D}_2 \to \mathcal{D}_1}(\texttt{b}) = 1 < \rho_g$.

In addition to [Weese and Schulz, 2008], we define a new discriminatory frequency predicate based on *entropy*. Entropy is a concept from *information theory* that measures the *information content* of a probability distribution [Cover and Thomas, 1991]. Given a discrete random variable X, with $p_i = P(X = i)$ the entropy H is defined as:

$$H(X) = -\sum_i p_i \ln p_i. \tag{7.7}$$

In accordance, we define the entropy H of a pattern ϕ in a set of databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$ as:

$$H(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m) = -\sum_{i=1}^{m} \frac{\text{supp}(\phi, \mathcal{D}_i)}{\sum_{j=1}^{m} \text{supp}(\phi, \mathcal{D}_j)} \log_m \frac{\text{supp}(\phi, \mathcal{D}_i)}{\sum_{j=1}^{m} \text{supp}(\phi, \mathcal{D}_j)}. \tag{7.8}$$

As probability distribution we use in (7.8) the normalized support distribution and use m as logarithmic base to normalize the entropy to a maximal value of 1.

Problem 7.3 (entropy substring mining problem)**.** *Given a support condition ρ_s and a maximum entropy bound α with $0 < \rho_s \leq 1$ and $0 \leq \alpha \leq 1$, the* entropy substring mining problem *is to detect all strings $\phi \in \Sigma^*$, such that* $\forall_{i \in [1..m]}$ $\text{supp}(\phi, \mathcal{D}_i) \geq \rho_s$ *and* $H(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m) \leq \alpha$.

The *entropy substring mining problem* is *symmetric* and returns the same patterns independently of the order of the input datasets. Patterns ϕ that are abundant in a small subset of databases and occur with low support in the complement set will tend to have a low entropy $H(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m)$, whereas a uniform support over all databases will result in the highest entropy of value 1. The problem can be considered as a symmetric generalization of the *emerging substring mining problem* to 2 and more databases. Table 7.2 shows the entropy values of different support distributions.

7.2.2 Monotonicity

We will now introduce the monotonicity property of frequency predicates that we use later to prune the search space of our algorithm. Examples 7.2–7.4 will show that each frequency predicate of the Problems 7.1–7.3 contains a monotonic subpredicate.

Definition 7.3. If for a frequency predicate $pred : \mathbb{N}^m \to \{\text{true}, \text{false}\}$ holds that:

$$\forall_{u,v \in \mathbb{N}^m, u \leq v} \Big(pred(u) \Rightarrow pred(v) \Big), \tag{7.9}$$

then *pred* is called *monotonic*.

Proposition 7.1. *For a monotonic*[1] *frequency predicate pred on databases $\mathcal{D}_1, \ldots, \mathcal{D}_m \subseteq \Sigma^*$ it holds that:*

$$\forall_{\phi,\psi \in \Sigma^*, \phi \preceq \psi} \Big(pred(\text{freq}(\psi, \mathcal{D}_1, \ldots, \mathcal{D}_m)) \Rightarrow pred(\text{freq}(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m)) \Big). \tag{7.10}$$

Proof. Each occurrence of ψ is also an occurrence of ϕ. Thus, $\text{freq}(\psi, \mathcal{D}_1, \ldots, \mathcal{D}_m) \leq \text{freq}(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m)$ holds. ∎

Example 7.2. As seen in Example 7.1 the frequency predicate for the *emerging substring mining problem* is:

$$pred(d_1, d_2) = (d_1 \geq \rho_s \cdot |\mathcal{D}_1|) \wedge (d_1 \cdot |\mathcal{D}_2| \geq \rho_g \cdot d_2 \cdot |\mathcal{D}_1|). \tag{7.11}$$

Generally, *pred* is not monotonic as shown in Example 7.1. Recall that `ba` is *emerging* although `b` is not. However, if we consider only the left inequality:

$$pred_{\text{m}}(d_1, d_2) := (d_1 \geq \rho_s \cdot |\mathcal{D}_1|), \tag{7.12}$$

[1] Please note that Raedt *et al.*; Fischer *et al.* consider *pattern* predicates and we consider *frequency* predicates. The properties of both are reciprocal and therefore they call *anti-monotonic* what we call *monotonic*.

$pred_{\mathrm{m}}$ is monotonic, as for all $u, v \in \mathbb{N}^2, u \le v$ holds $u_1 \ge \rho_s \cdot |\mathcal{D}_1| \Rightarrow v_1 \ge \rho_s \cdot |\mathcal{D}_1|$. Obviously it holds that $pred \Rightarrow pred_{\mathrm{m}}$, a fact which we use later. In the next two examples we show the predicates *pred* and corresponding monotonic predicates $pred_{\mathrm{m}}$ with $pred \Rightarrow pred_{\mathrm{m}}$ for the remaining two problems.

Example 7.3. Predicates for the *frequent pattern mining problem*:

$$\begin{aligned} pred(d) &= (\min_1 \le d_1 \le \max_1) \wedge \ldots \wedge (\min_m \le d_m \le \max_m), && (7.13) \\ pred_{\mathrm{m}}(d) &:= (\min_1 \le d_1) \wedge \ldots \wedge (\min_m \le d_m). && (7.14) \end{aligned}$$

Example 7.4. Predicates for the *entropy substring mining problem*:

$$pred(d) = \left(\forall_{i \in [1..m]} \frac{d_i}{|\mathcal{D}_i|} \ge \rho_s\right) \wedge \left(\sum_{i=1}^{m} \frac{d_i}{|\mathcal{D}_i| \cdot \omega(d)} \log_m \frac{d_i}{|\mathcal{D}_i| \cdot \omega(d)} \le \alpha\right), \quad (7.15)$$

$$pred_m(d) := \left(\forall_{i \in [1..m]} \frac{d_i}{|\mathcal{D}_i|} \ge \rho_s\right), \qquad \textit{with} \quad \omega(d) = \sum_{i=1}^{m} \frac{d_i}{|\mathcal{D}_i|}. \quad (7.16)$$

7.2.3 Conjunctive predicates

If a predicate *pred* on databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$ can be decomposed into a conjunction of predicates p_i on $\mathcal{D}_i$ such that holds:

$$pred(\mathrm{freq}(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m)) = p_1(\mathrm{freq}(\phi, \mathcal{D}_1)) \wedge \ldots \wedge p_m(\mathrm{freq}(\phi, \mathcal{D}_m)), \quad (7.17)$$

we call *pred* a *conjunctive* predicate and $pred_1, \ldots, pred_m$ its *factors*. It is easy to see that the predicate of Problem 7.1 is conjunctive whereas the predicates of Problem 7.2 and 7.3 are not.

7.3 Monotonic hull

We now show how to connect arbitrary frequency predicates with suffix trees. To do so, we give a theoretical description of the minimal set of nodes that need to be expanded.

Definition 7.4. Given frequency predicates *pred* and $pred_{\mathrm{hull}}$. $pred_{\mathrm{hull}}$ is called *a monotonic hull* of *pred*, if it is monotonic and $pred \Rightarrow pred_{\mathrm{hull}}$ holds.

The most trivial monotonic hull of each frequency predicate *pred* is $pred_{\mathrm{hull}} \equiv$ true. If we take a look at the generalized suffix tree T of databases $\mathcal{D}_1, \ldots, \mathcal{D}_m$, we make the following observations:

Proposition 7.2. *Let pred be an arbitrary frequency predicate and* $pred_{\mathrm{m}}$ *an arbitrary* monotonic *frequency predicate on* $\mathcal{D}_1, \ldots, \mathcal{D}_m$*. For all pairs of parents* $\overline{\alpha}$ *and children* $\overline{\alpha\beta}$ *in* T *it holds that:*

1. *If* $pred(\text{freq}(\alpha\beta, \mathcal{D}_1, \ldots, \mathcal{D}_m))$ *is true then* $pred(\text{freq}(\chi, \mathcal{D}_1, \ldots, \mathcal{D}_m))$ *is true for each string* χ *with* $\alpha \prec \chi \preceq \alpha\beta$.

2. *If* $pred_{\text{m}}(\text{freq}(\alpha\beta, \mathcal{D}_1, \ldots, \mathcal{D}_m))$ *is true then* $pred_{\text{m}}(\text{freq}(\alpha, \mathcal{D}_1, \ldots, \mathcal{D}_m))$ *is true.*

Proof. The frequency vectors of $\alpha\beta$ and χ with $\alpha \prec \chi \preceq \alpha\beta$ must be equal. If not, there would be a branching node between $\overline{\alpha}$ and $\overline{\alpha\beta}$ which contradicts the assumption $\overline{\alpha}$ would be the parent of $\overline{\alpha\beta}$. Hence 1. holds. 2. is a direct consequence of Proposition 7.1 as α is a substring of $\alpha\beta$. ■

In consequence of Proposition 7.2, it is sufficient to evaluate *pred* only on the nodes of T to compute the set $Th(pred)$. For every monotonic hull $pred_{\text{hull}}$ of *pred* the set of nodes, whose frequencies satisfies $pred_{\text{hull}}$, is a connected subgraph of T, which if non-empty contains the root node. Outside of this subgraph there is no node fulfilling *pred*. Our algorithm exclusively traverses this subgraph to compute the set $Th(pred)$. Hence, we are interested in keeping the subgraph as small as possible, leading to the next definition:

Definition 7.5. $pred_{\text{hull}}$ is called *the optimal monotonic hull* of *pred*, if it is a monotonic hull of *pred*, and for each monotonic hull $pred'_{\text{hull}}$ of *pred*, it holds that $pred_{\text{hull}} \Rightarrow pred'_{\text{hull}}$.

In other words, if $pred_{\text{hull}}$ is optimal, the corresponding subgraph is minimal.

7.4 The linear-time algorithm by Fischer et al.

In Section 7.1, we have summarized previous algorithms for frequency based string mining problems. We will give here a more detailed description for the optimal algorithm by Fischer, Heun, and Kramer (FHK algorithm) and two variants that have been proposed to reduce the memory consumption of the algorithm.

7.4.1 The original algorithm

For a given string s and a corresponding suffix array suftab, the lcp table lcp stores the length of the longest common prefix between lexicographically adjacent suffixes (Section 3.1). It can be shown that the length of the longest common prefix between non-adjacent suffixes of rank i and j, with $i < j$, is the minimum over lcp values between $i+1$ and j:

$$\begin{aligned} \mathsf{lcp}[i] &:= \left|\text{lcp}\left\{s_{\mathsf{suftab}[i-1]}, s_{\mathsf{suftab}[i]}\right\}\right|, & (7.18) \\ \left|\text{lcp}\left\{s_{\mathsf{suftab}[i]}, s_{\mathsf{suftab}[j]}\right\}\right| &= \min_{k \in (i..j]} \mathsf{lcp}[k]. & (7.19) \end{aligned}$$

To answer so-called range minimum queries of the form:

$$\text{RMQ}_{\mathsf{lcp}}(i,j) = \arg\min_{k \in (i..j]} \mathsf{lcp}[k] \qquad (7.20)$$

in constant time and determine the lcp value of arbitrary suffix pairs, Fischer and Heun [2006] proposed a data structure which can be constructed in $\mathcal{O}(n)$ time using $o(n)$ extra memory.

The main concept of the algorithm by Fischer, Heun, and Kramer [2006] is based on the following idea. Let $\mathcal{S}_{\mathcal{D}_i}(\omega)$ be the total number of occurrences of a string ω in database $\mathcal{D}_i = \{\phi_i^1, \dots, \phi_i^{|\mathcal{D}_i|}\}$ and the so-called correction term $\mathcal{C}_{\mathcal{D}_i}(\omega)$ be defined as the number of recurrences of ω in the same string of $\mathcal{D}_i$:

$$\mathcal{S}_{\mathcal{D}_i}(\omega) := \sum_{\phi \in \mathcal{D}_i} \left|\{j \mid \phi[j..j+|\omega|) = \omega\}\right|, \tag{7.21}$$

$$\mathcal{C}_{\mathcal{D}_i}(\omega) := \sum_{\phi \in \mathcal{D}_i} \max\left(0, \left|\{j \mid \phi[j..j+|\omega|) = \omega\}\right| - 1\right). \tag{7.22}$$

Then the frequency $\mathrm{freq}(\omega, \mathcal{D}_i)$ equals $\mathcal{S}_{\mathcal{D}_i}(\omega) - \mathcal{C}_{\mathcal{D}_i}(\omega)$. Consider the generalized suffix tree of the databases $\mathcal{D}_1, \dots, \mathcal{D}_m$ and let ω be the concatenation string of a branching node $\overline{\omega}$ with children $\overline{\omega\alpha_1}, \dots, \overline{\omega\alpha_l}$. Then the following holds:

$$\mathcal{S}_{\mathcal{D}_i}(\omega) = \sum_{j=1}^{l} \mathcal{S}_{\mathcal{D}_i}(\omega\alpha_j), \tag{7.23}$$

$$\mathcal{C}_{\mathcal{D}_i}(\omega) \geq \sum_{j=1}^{l} \mathcal{C}_{\mathcal{D}_i}(\omega\alpha_j). \tag{7.24}$$

To enable the recursive computation of $\mathcal{C}$, Fischer *et al.* precompute an auxiliary array $\mathcal{C}'$ for each of the m databases using lcp range minimum queries which takes $\Theta(m \cdot n)$ time and memory in total:

$$\mathcal{C}_{\mathcal{D}_i}(\omega) = \mathcal{C}'_{\mathcal{D}_i}(\omega) + \sum_{j=1}^{l} \mathcal{C}_{\mathcal{D}_i}(\omega\alpha_j). \tag{7.25}$$

They observed that $\mathcal{C}_{\mathcal{D}_i}(\omega)$ can be calculated by adding up for every string in $\mathcal{D}_i$ the number of its lexicographically adjacent suffix pairs that begin with ω. Consequently, $\mathcal{C}'_{\mathcal{D}_i}(\omega)$ is equal to the number of suffixes that share a longest-common-prefix ω with their next greater suffix in the same database string. $\mathcal{C}'$ can be computed in a linear scan of the suffix array recording for every suffix the last seen suffix in the same database string. The longest common prefix ω between both suffixes is determined by a RMQ and the value $\mathcal{C}'_{\mathcal{D}_i}(\omega)$ is increased by one[2].

The recursive computation of $\mathcal{S}$ and $\mathcal{C}$ is integrated into a bottom-up traversal of the suffix tree (described in Section 3.9). Algorithm 7.1 shows the outline of the whole linear-time algorithm by Fischer *et al.* [2006]. Their approach concatenates all strings in $\mathcal{D}_1, \dots, \mathcal{D}_m$ to a single *union string* and corrects border effects, e.g. miscounting substrings that cross string borders in the union string.

7.4.2 Space efficient variants

We will briefly describe two memory efficient approaches published by Kügel and Ohlebusch [2008]; Fischer *et al.* [2008] which are modifications of the original FHK algorithm.

[2] In fact, Fischer *et al.* implemented $\mathcal{C}'_{\mathcal{D}_i}$ as a string of length n and increase it at any ℓ-index position of the ω-interval.

Algorithm 7.1: FHK($\mathcal{D}_1, \dots, \mathcal{D}_m, pred$)

input : databases $\mathcal{D}_1, \dots, \mathcal{D}_m$, frequency predicate *pred*

```
// preprocessing
```
1 construct the suffix array SA of the union string over $\mathcal{D}_1, \dots, \mathcal{D}_m$
2 build and preprocess the lcp table for constant time range minimum queries
3 calculate the correction term auxiliary arrays $C'_{\mathcal{D}_1}, \dots, C'_{\mathcal{D}_m}$

```
// extraction phase
```
4 foreach *suffix tree node* $\overline{\omega}$ *in postorder DFS* do
5   compute $\mathcal{S}_{\mathcal{D}_i}(\omega)$ and $\mathcal{C}_{\mathcal{D}_i}(\omega)$ using the recurrences (7.23) and (7.25)
6   *Freq* $\leftarrow (\mathcal{S}_{\mathcal{D}_1}(\omega) - \mathcal{C}_{\mathcal{D}_1}(\omega), \dots, \mathcal{S}_{\mathcal{D}_m}(\omega) - \mathcal{C}_{\mathcal{D}_m}(\omega))$
7   if $pred(Freq)$ then
8   output strings χ with $\alpha \prec \chi \preceq \omega$, where $\overline{\alpha}$ is the parent node of $\overline{\omega}$

The modification by Kügel and Ohlebusch (KO) is only applicable to conjunctive predicates *pred* like the *frequent pattern mining problem* and computes separate solution sets $Th(pred_i)$ iteratively for each factor $pred_i$ on database $\mathcal{D}_i$. After each iteration the solution set $Th(pred_i)$ is intersected with the result of previous intersections, as the final solution set of a conjunctive predicate is the intersection of all separate solution sets $Th(pred) = \bigcap_{i=1}^{m} Th(pred_i)$. The intersection of two solution sets is based on an algorithm for suffix array merging [Jeon *et al.*, 2005] retaining linear time complexity. The memory consumption, however, merely depends on the total size of the largest database.

Fischer, Mäkinen, and Välimäki (FMV) decreased the memory consumption of the original FHK algorithm by compressing the suffix array [Navarro and Mäkinen, 2007], the lcp table [Sadakane, 2007], and the RMQ data structure [Fischer and Heun, 2007]. Their approach avoids the precomputation of the correction term auxiliary arrays and instead computes the values $\mathcal{C}'_{\mathcal{D}_i}(\omega)$ for nodes $\overline{\omega}$ on the DFS stack during the traversal and stores the $\mathcal{S}$ and $\mathcal{C}$ numbers in a searchable partial sum data structure [Mäkinen and Navarro, 2008]. The overall memory consumption of the FMV algorithm is $\mathcal{O}(n \log |\Sigma| + d \log n)$ bits and the runtime is $\mathcal{O}(n \log n)$, where d is the overall number of strings in the databases.

7.5 A fast algorithm based on lazy suffix trees

This section introduces the *deferred frequency index* (DFI) which is fundamentally based on a generalized lazy suffix tree, i.e. a lazy suffix tree of multiple sequences as proposed in Chapter 4. The DFI algorithm constructs only an upper part of a generalized suffix tree in a top-down manner using a modified *wotd* algorithm (Algorithm 4.1 on page 66).

7.5.1 The deferred frequency index

The main idea of our algorithm is to calculate the frequency vector for a tree node during the node expansion of its parent node in the *wotd* algorithm. In this way, we are able

Algorithm 7.2: DFI($T, \overline{\alpha}, pred, pred_{\text{hull}}$)

```
input    : unexpanded node ᾱ
1  Freq ← sortAndCountFreq(ᾱ)
2  foreach c ∈ Σ and R(αc) ≠ ∅ do
3      αcβ ← lcp R(αc)
4      if pred(Freq[c]) then
5          Output strings χ with αc ⪯ χ ⪯ αcβ
6      if pred_hull(Freq[c]) then
7          if |R(αc)| = 1 then                        // leaf node
8              add leaf ‾αcβ as a child of ᾱ in T
9          else                                       // branching node
10             add inner node ‾αcβ as a child of ᾱ in T
11             DFI(T, ‾αcβ, pred, pred_hull)          // recurse into child subtree
```

to restrict the set of node expansions to the optimal monotonic hull $pred_{\text{hull}}$ of a given predicate *pred*.

Algorithm 7.2 starts with DFI($T, \overline{\epsilon}, pred, pred_{\text{hull}}$) on a tree T consisting of only the *unexpanded* root node $\overline{\epsilon}$. First, sortAndCountFreq is called for the current node $\overline{\alpha}$ in line 1. Identically to Algorithm 4.1 on page 66, the set $R(\alpha)$ is divided into groups $R(\alpha c)$ of suffixes that share the same character $c \in \Sigma$ after their common prefix α. In addition, an array *Freq*, that stores in *Freq*[c] the frequency vector $\text{freq}(\alpha c, \mathcal{D}_1, \dots, \mathcal{D}_m)$, is returned. In the next section we explain the implementation details of function sortAndCountFreq. The longest common prefix of every non-empty group $R(\alpha c)$ is determined and assigned to $\alpha c\beta$ in line 3. If the predicate *pred* evaluated with the frequency vector *Freq*[c] is true, by Proposition 7.2 all strings χ with $\alpha c \preceq \chi \preceq \alpha c\beta$ belong to *Th*(*pred*) and are output. In line 6 $pred_{\text{hull}}$ is evaluated on *Freq*[c]. Only if true is returned, the subtree below the node $\overline{\alpha c\beta}$ may contain a node $\overline{\gamma}$ with $\gamma \in Th(pred)$ and will be expanded recursively. If false is returned, the node $\overline{\alpha c\beta}$ is not created, as no further subtree expansion is necessary.

Algorithm 7.2 is correct and outputs the set *Th*(*pred*) because of the following: For each database substring ϕ there is a path from the *root* ending in a node or on an edge to a node. This node has the same frequency vector as ϕ and will be visited if it satisfies $pred_{\text{hull}}$ and output iff it satisfies *pred*. As $pred_{\text{hull}}$ is a monotonic hull, no node that satisfies *pred* is left out by the algorithm. The algorithm consumes $\mathcal{O}(n \log n)$ time in the average and $\mathcal{O}(n^2)$ time in the worst case, and $\mathcal{O}(n)$ memory [Giegerich *et al.*, 2003].

For the *frequent pattern mining problem* and the *emerging* or *entropy substring mining problem* one only needs to replace *pred* and $pred_{\text{hull}}$ in Algorithm 7.2 with the predicates deduced in Examples 7.2–7.4, respectively. The monotonic hulls for these problems are also optimal as Propositions A.1–A.3 prove (see Appendix). Figure 7.1 shows the DFI for the *emerging substring mining problem* considered in Example 7.1.

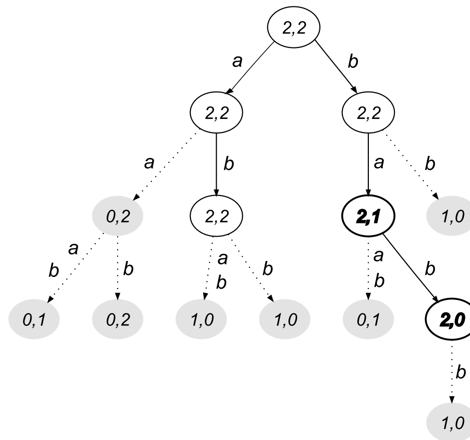


Figure 7.1: *The generalized suffix tree of our example databases $\mathcal{D}_1$ and $\mathcal{D}_2$. For clarity, we omitted sentinels $\$_j$. Considering the problem of Example 7.1, the DFI would expand only the white nodes. Grey nodes are not built. The bold nodes* $\overline{\mathtt{ba}}$, $\overline{\mathtt{bab}}$ *represent the* emerging substrings*. Each node holds the frequency vector of its corresponding substring (compare the frequencies of* $\overline{\mathtt{b}}$ *and* $\overline{\mathtt{ba}}$ *with Table 7.2).*

7.5.2 Algorithmic details

In this section, we explain the function SORTANDCOUNTFREQ in detail (Algorithm 7.3). The sets $R(\alpha)$ are not actually stored as sets of strings, but as intervals of the string suftab, which contains pairs (i, j) that represent suffixes s^i_j. This string is initialized ascendingly as described in Section 4.2.3. To determine which database a string is part of, we need a function *getDatabaseNo* that returns for each sequence number i the corresponding database number k, with $s^i \in \mathcal{D}_k$.

When SORTANDCOUNTFREQ($\overline{\alpha}, pred, pred_{\text{hull}}$) is called, suftab$[l..r)$ contains the start positions of suffixes beginning with α. Each start position corresponds to a suffix in $R(\alpha)$. Because $\overline{\alpha}$ is *unexpanded*, the suffixes in suftab$[l..r)$ have been sorted with counting sort [Cormen *et al.*, 2001] up to the first $|\alpha|$ characters by previous function calls. As counting sort is stable, the pairs (i, j) in suftab$[l..r)$ are in ascending order. In particular, the corresponding sequence numbers i are stored in contiguous blocks. Counting sort divides $R(\alpha)$ into buckets $R(\alpha c)$ for each character $c \in \Sigma$ (lines 3–4). The frequency of each bucket can simply be counted by counting blocks of equal sequence numbers (line 5).

We keep track of three arrays in the size of the alphabet, i.e. $|\Sigma|$, namely C, *Freq*, and *Last*. C is the original array from counting sort, and $C[c]$ counts the occurrences of αc. *Freq* stores frequency vectors, and *Freq*$[c][k]$ determines how often αc occurred in distinct sequences of $\mathcal{D}_k$. *Last* is used to construct *Freq* (lines 5–8).

Algorithm 7.3: SORTANDCOUNTFREQ($\overline{\alpha}$)

```
input   : unexpanded suffix tree node ᾱ
output  : freq(αc, D_1, ..., D_m) for each c ∈ Σ
require : suftab[l..r) stores all suffixes beginning with α, suffixes from the same
          sequence are contiguous in the interval
ensure  : suffixes from the same sequence are contiguous in output intervals
          suftab[Bucket[c]..Bucket[c + 1])

1  initialize C, Freq, Last with zeros
2  for k ← l to r − 1 do
3      (i, j) ← suftab[k]        // get sequence number i and suffix start position j
4      if j + |α| < |s^i| then                                  // ignore $-edges
5          c ← s^i[j + |α|]
6          C[c] ← C[c] + 1
7          if Last[c] ≠ i then                         // have we entered a new sequence?
8              Last[c] ← i
9              d ← GETDATABASENO(i)            // get database number d with s^i ∈ D_d
10             Freq[c][d] ← Freq[c][d] + 1

   // proceed with steps 3 and 4 of counting sort (Cormen et al., 8.2, ll. 6-12)
11 stably sort suffixes suftab[l..r) by the character at position |α|
12 return Freq      // now Freq[c] contains the frequency vector freq(αc, D_1, ..., D_m)
```

7.6 Experimental results

To evaluate the performance of our algorithm, we conducted a number of experiments with databases of different characteristics. We used a previously compiled set of human and Drosophila core promoters [Fitzgerald *et al.*, 2006], the UniProt proteome sets of H. sapiens and M. musculus, release 12.6, and C. elegans and A. thaliana, release 15.6 [UniProt Consortium, 2008], verses of the King James Bible and the Bible in Basic English, and posts of 20 newsgroups from the UCI Machine Learning Repository [Asuncion and Newman, 2007]. The alphabet sizes $|\Sigma|$, the number of contained sequences, and references of these databases can be found in Table 7.3.

For our tests we compared our algorithm with the algorithms FHK, KO, and FMV described in Section 7.4. All implementations were written in C++ and compiled using the same compiler options. They run under Linux on an Intel Xeon 3.2 GHz with 2 GB of RAM. To reduce influences from the operating system and secondary storage units, the output was redirected to the null-device, and each experiment was repeated 5 times. We measured the running time and memory consumption of both algorithms using the GNU tools `time` and `memusage`. In an additional run we successfully verified that the outputs of our and the other tools are identical.

In the following we compare the performance of our approach with existent tools on two and multiple real-world databases. Finally, we analyze the applicability of our newly introduced entropy-based predicate in an experiment with proteomes of four species.

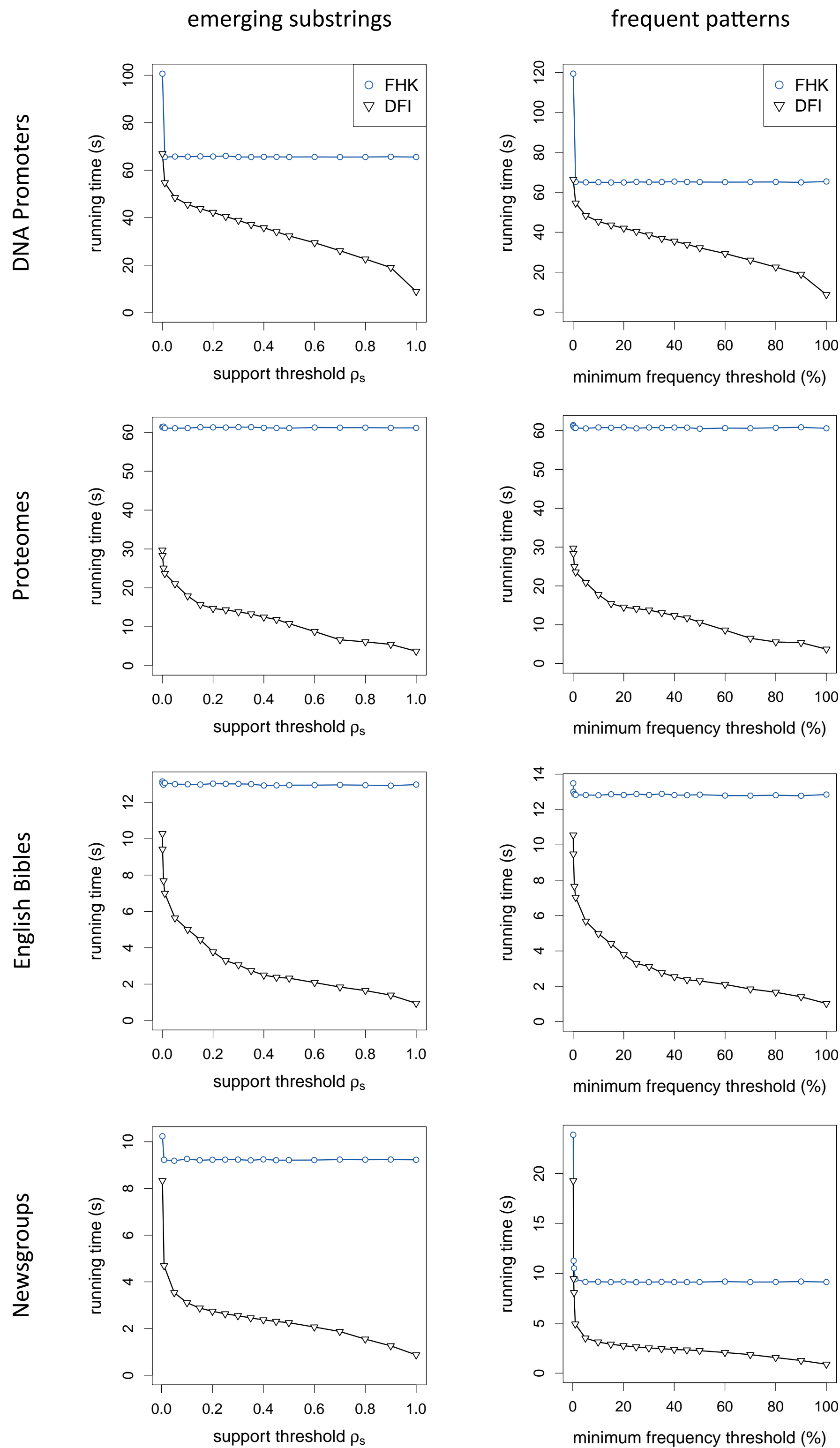


Figure 7.2: *Runtime comparison of the FHK algorithm [Fischer* et al.*, 2006] and our DFI implementation for the* emerging substring mining problem *and the* frequent pattern mining problem*. Experiment details are listed in Table 7.3.*

name	description	$\|\Sigma\|$	length [Mb]	#seqs	source
$\mathcal{D}_1$	Human Promoters	5	23	15011	Fitzgerald *et al.*
$\mathcal{D}_2$	Drosophila Promoters	5	17	10914	
$\mathcal{D}_1$	Human Proteome	24	18	40827	UniProt Consortium
$\mathcal{D}_2$	Mouse Proteome	24	16	35344	
$\mathcal{D}_1$	King James Bible	128	4.1	31102	English Bible Online
$\mathcal{D}_2$	Bible in Basic English	128	4.2	31102	http://www.o-bible.com
$\mathcal{D}_1$	Windows Newsgroup	128	3.9	2000	Machine Learning Repos.
$\mathcal{D}_2$	Computer Newsgroup	128	3.4	3000	Asuncion and Newman
$\mathcal{D}_{1-4}$	4 Proteomes	24	58	136562	UniProt Consortium
$\mathcal{D}_{1-20}$	20 Newsgroups	128	16	12273	Machine Learning Repos.

Table 7.3: *Characteristics for the different databases we used.*

7.6.1 Two databases

Of all other approaches the theoretically optimal FHK algorithm [Fischer *et al.*, 2006] has turned out to be the fastest algorithm in practice for Problem 7.1 and 7.2. Hence, we used the FHK algorithm as reference for the first experiment where we compared the runtime behavior under different parameters. We searched different pairs of databases for *emerging substrings* and for the solution of the *frequent pattern mining problem* with different values of ρ_s and varying $\min_1$, respectively. As ρ_g and $\max_2$ had no measurable influence on the tested algorithms, only the results for $\rho_g = 5$ and $\max_2 = |\mathcal{D}_2|/2$ are shown. The results for other values look similar [Fischer *et al.*, 2005]. We made no other restrictions, i.e. $\max_1 = \infty$, $\min_2 = 0$.

Figure 3 shows that our approach is in all cases faster than the FHK algorithm, even for small values ρ_s or $\min_1$ when the whole suffix tree needs to be constructed. As an example, for $\rho_s = 0.2$ the DFI is with 16 seconds on the proteome databases roughly four times faster than the FHK algorithm. Considering reasonable[3] values of $\rho_s < 0.2$ and $\min_1 < 0.2 \cdot |\mathcal{D}_1|$ our algorithm is 1.5–4 times faster in practice. The runtime peaks for small values of ρ_s or $\min_1$ are due to the high amount of strings in the solution space that were reported.

In the second experiment we compared running times and memory consumptions of the DFI, the FHK, and the memory efficient FMV algorithm for different parameter settings of the *emerging substring mining problem*, see Table 7.4. The FMV algorithm is the most memory efficient, as it occupies on average 1.69 times less memory than the DFI and 3.55 times less memory than the FHK algorithm. However, this reduction of memory consumption induces an average increase in running time by a factor of 212 compared to the DFI. The DFI is generally the fastest.

[3] Dong and Li [1999] report that a minimum support of 1 %–20 % for finding *emerging patterns* could contribute significantly to knowledge discovery.

parameters		databases	time [s]			memory [MB]		
ρ_s	ρ_g		DFI	FHK	FMV	DFI	FHK	FMV
0.05	5	DNA Promoters	**51**	67	6317	469	920	**246**
0.1	5	DNA Promoters	**47**	67	6180	468	920	**246**
0.05	5	Proteomes	**22**	62	7974	329	779	**233**
0.1	5	Proteomes	**18**	62	7949	328	779	**233**
0.05	5	Newsgroups	**3**	9	1243	82	167	**47**
0.1	5	Newsgroups	**3**	9	1238	81	167	**47**

Table 7.4: *Running times and memory consumption for the* emerging substring mining problem *for different parameters on two databases.*

It is interesting to note that the FHK algorithm as well as the DFI have an $O(n)$ memory consumption, but the DFI needs only about half of the memory. The FHK algorithm has an almost constant running time and memory consumption as it does not take advantage of the monotonic pruning of the suffix tree like our deferred approach does.

7.6.2 Multiple databases

In the third experiment we analyzed the memory behaviour of the Kügel and Ohlebusch [2008] variant of the FHK algorithm for the conjunctive *frequent pattern mining problem*. We used an adaptation of the FHK algorithm for more than two databases provided by Kügel and Ohlebusch. As described in Section 7.4, the memory footprint of the KO algorithm depends only on the size of the largest database and predicate parameters do not affect the memory and time consumption of the KO algorithm. The DFI is faster by a factor of 2–20 compared to the KO algorithm and consumes 3–10 times more memory, see Table 7.5. Generally speaking, the higher the number of databases the better the memory improvement of the KO algorithm compared to the other algorithms. Interestingly, the KO algorithm on 4 proteomes consumes less memory than the FMV algorithm on 2 of the 4 proteomes.

In Table 7.6 we list the results of the forth experiment, where we compared the performance of DFI and FHK for the new *entropy substring mining problem* on several databases. To do so, we adapted the implementation of the FHK algorithm for the new predicate. As can be seen, the memory footprint of the DFI is 3–9 times smaller compared to the FHK algorithm. Thus, the memory improvement of the DFI compared to the FHK algorithm is higher if more databases are used, considering that in the second experiment with two databases the memory footprint was two times smaller.

7.6.3 Detection of species specific protein domains

We sought out to test the new *entropy substring mining problem* on a biologically motivated example. For particular biological applications of the *frequent pattern mining prob-*

parameters		databases	time [s]			memory [MB]		
min_i	max_i		DFI	FHK	KO	DFI	FHK	KO
10	1000	4 Proteomes	**78**	154	224	660	1848	**182**
100	5000	4 Proteomes	**63**	153	224	584	1848	**182**
500	50000	4 Proteomes	**54**	153	224	575	1848	**182**
500	100000	4 Proteomes	**52**	154	223	575	1848	**182**
50	1000	20 Newsgroups	**7**	36	46	176	1526	**18**
200	1000	20 Newsgroups	**4**	37	46	175	1526	**18**
500	5000	20 Newsgroups	**2**	37	46	145	1526	**18**

Table 7.5: *Running times and memory consumption for the* frequent pattern mining problem *for various parameters on multiple databases.*

parameters		databases	time [s]		memory [MB]	
ρ_s	α		DFI	FHK	DFI	FHK
0.001	0.2	4 Proteomes	**65**	155	**629**	1848
0.005	0.4	4 Proteomes	**57**	155	**590**	1848
0.01	0.7	4 Proteomes	**52**	156	**578**	1848
0.01	0.2	20 Newsgroups	**59**	98	**217**	1526
0.05	0.2	20 Newsgroups	**13**	41	**178**	1526
0.1	0.4	20 Newsgroups	**11**	40	**177**	1526

Table 7.6: *Running times and memory consumption for the* entropy substring mining problem *for various parameters on multiple databases.*

lem or the *emerging substring mining problem*, we refer the reader to [Stöber *et al.*, 1996; Brāzma *et al.*, 1998; Birzele and Kramer, 2006; Mitašiūnaitė *et al.*, 2008].

Domains are functional substrings of proteins that are conserved in a large set of proteins, which is termed a protein family. We are interested in finding protein families typical for species. We formulate this as a contrast data mining problem, where we take the proteomes[4] of different species and search for protein domains that are specific to a small subset, possibly one, of the species. A low entropy of a pattern ϕ reflects specificity to a small subset of species, see Table 7.2. The hope is, that such pattern ϕ emanates from real species specific protein domains.

We applied the *entropy substring mining problem* problem to a dataset of four proteomes from different parts of the phylogenetic tree, namely the proteomes of *H. sapiens, M. musculus, A. thaliana, and C. elegans*. For this task we modified our algorithm to report only strings $\phi \in Th(pred)$ that are not substrings of longer strings $\gamma \in Th(pred)$ with

[4] A proteome is the set of all proteins of a given species.

pattern	entropy	frequency				reference
ϕ	$H(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_4)$	$\mathcal{D}_1$	$\mathcal{D}_2$	$\mathcal{D}_3$	$\mathcal{D}_4$	
KSDVY	0.230657	8	4	302	7	[Walker, 1994]
YSFGV	0.254649	4	5	335	14	
DVYSFG	0.263708	9	6	351	11	
SDVYS	0.356452	8	12	306	15	
MAYDRYVAIC	0.338613	94	374	0	0	[Xie *et al.*, 2000],
AYDRYVAIC	0.343253	97	375	0	0	[Zhang and Firestein, 2002]
YDRYVAIC	0.352435	135	493	0	0	
DRYVAIC	0.369871	193	632	0	0	
RYVAIC	0.379256	194	634	0	1	
PMLNPL	0.377389	98	306	0	0	[Zhang and Firestein, 2002]
LRNKDV	0.378017	79	283	2	0	
YSLRNKD	0.385805	97	287	0	0	
NPLIYSLRN	0.389852	102	294	0	0	

Table 7.7: *Entropy substring mining on the four proteomes* H. sapiens, M. musculus, A. thaliana, and C. elegans *($\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, $\mathcal{D}_4$) with $\rho_s = 0.008$ and $\alpha = 0.39$.*

$\text{freq}(\phi, \mathcal{D}_1, \ldots, \mathcal{D}_m) = \text{freq}(\gamma, \mathcal{D}_1, \ldots, \mathcal{D}_m)$. For ease of exposition, we also omitted strings that are prefixes of longer strings in the solution set. In Table 7.7 we list the *entropy substrings* that have been mined using the parameters $\rho_s = 0.008$ and $\alpha = 0.39$. All mined patterns belong to three distinct motifs that we discuss in the next paragraphs.

The substrings with the lowest entropy values are specific for the plant *A. thaliana*. The KSDVYSFGV motif is part of the kinase domain IX in receptor-like protein kinases (RLKs) in higher plants [Walker, 1994]. RLKs constitute a big protein family in higher plants, alone in *A. thaliana* there are more than 600 members [Morris and Walker, 2003]. They are involved in a number of different signaling pathways like cell differentiation, plant growth, and development.

The other two motifs that are found are abundant in mouse proteins and occur with a lower frequency in the human proteome, but are absent in the plant and worm proteomes. Both motifs are highly conserved domains of odorant or olfactory receptors (ORs) in mouse [Xie *et al.*, 2000; Zhang and Firestein, 2002]. ORs are located in cell membranes and are responsible for the detection of odor molecules. The first motif, MAYDRYVAIC, is part of the transition between the transmembrane II domain and the intercellular loop 2 of ORs in mouse. The second motif, PMLNPLIYSLRNKDV, describes a major part of the transmembrane domain VII of ORs in mouse. It was shown that these conserved protein domains are specific to Class II ORs in mouse [Zhang and Firestein, 2002]. As mentioned above, these two motifs do also occur in human proteins and these types of ORs constitute a large protein family that was found to be vertebrate specific [Berghard and Dryer, 1998]. Nevertheless, humans have only one-third the number of ORs than what has been found in mouse [Zhang and Firestein, 2002], reflected by the pattern frequency distribution in Table 7.7.

Chapter 8 Conclusion and Future Work

In this thesis, we presented data structures and algorithms with applications in the analysis of high-throughput sequencing data. We developed a uniform framework for constructing and accessing different substring indices of a single or multiple strings in main or external memory and showed its applicability for indexing multiple whole mammal genomes. Moreover, we provided algorithms for typical applications based on indices, e.g. exact and approximate pattern matching and repeat search, and in the last chapters introduced high-throughput sequencing applications based on two of the proposed indices. To make our framework and tools freely accessible to the research and user community, we implemented it as part of SeqAn [Döring *et al.*, 2008] a platform-independent generic C++ template library for sequence analysis. Due to its modularity, it was easily possible to integrate our framework into other alignment tools [Rausch *et al.*, 2008; Langmead *et al.*, 2009; Emde *et al.*, 2010; Kehr *et al.*, 2011; Emde *et al.*, 2012; Siragusa *et al.*, 2013a].

In Chapter 6, we presented RazerS, an efficient read mapping tool that guarantees to find all reads within a user-defined Hamming or edit distance. In addition, a fixed error model and a user-defined loss rate can be used to find the reads at higher speed with controlled sensitivity. RazerS hence provides a perfect sensitivity-time tradeoff. Our tool can also handle paired-end reads as well as arbitrary number of errors and arbitrary read lengths, which makes it usable for the new or improved technologies that will provide longer reads. The latter two features are unique among the current implementations. To provide a shared-memory parallelization we used OpenMP and dynamic load balancing. Compared to other state-of-art read mappers, RazerS shows the highest sensitivity with a comparable performance. It is the preferable tool for applications that require a high sensitivity even in the presence of repeats, e.g. variation detection pipelines. The novel algorithmic ideas used in RazerS, e.g. lossy filtering with sensitivity control or using a banded adaptation of Myers' algorithm for efficient bit-parallel verification, can also be applied to improve existing read mappers with a similar filtration-verification approach. Many algorithmic components of RazerS were integrated into SeqAn and the whole algorithm was basis of similar tools for local read alignment [Hauswedell, 2009] or the alignment of miRNA [Emde *et al.*, 2010] or split reads [Emde *et al.*, 2012].

In Chapter 7, we presented a new approach to constraint-based string mining that outperforms the best-known algorithms by Fischer *et al.* [2006, 2008]; Kügel and Ohlebusch [2008] in running time as the experiments show. The better running time can be attributed to various factors. Most importantly, the optimal monotonic hull of a fre-

quency predicate is incorporated to prune the search space to a minimum, resulting in the deferred frequency index (DFI). Moreover, the frequency information is extracted as a constant time byproduct during the suffix tree construction. Our algorithm inherits the good cache locality of the lazy suffix tree if expanded in a depth-first search fashion [Giegerich *et al.*, 2003]. We used the notion of entropy from information theory and introduced a symmetric, discriminatory predicate that generalizes the *emerging substring mining problem* for more than two databases. In an experiment with proteomes of four species we showed that it can be used to mine parts of protein domains that belong to species specific protein families. Generally, the DFI is the preferable algorithm for frequency based string mining. For huge datasets that the DFI cannot process in main memory, space efficient variants of the FHK algorithm [Fischer *et al.*, 2006] should be considered. For conjunctive predicates, the KO algorithm [Kügel and Ohlebusch, 2008] is the next best alternative. For non-conjunctive predicates, the FMV algorithm [Fischer *et al.*, 2008] can reduce the memory consumption at the price of a high increase in running time.

Future Work. The work presented in this thesis can be complemented in several aspects of future research. First, different compressed indices could be provided to enable larger texts to be processed in main memory with focus on generic approaches that are efficient in practice. In [Grossi *et al.*, 2003; Sadakane, 2003; Navarro and Mäkinen, 2007] the authors devise compressed indices which are based on succinct representations of the suffix array or the lcp table. In conjunction with a data structure for constant-time range-minimum queries as proposed in [Fischer and Heun, 2006], a compressed variant of the enhanced suffix array could be integrated into our framework as proposed in [Fischer *et al.*, 2008] and extended to multiple strings. Another memory improvement for small alphabets completely refrains from using the lcp or child table and instead uses a binary search to determine the children of a suffix tree node [Navarro and Baeza-Yates, 2000]. We implemented a prototype of the FM index [Ferragina *et al.*, 2004] which proved its applicability to high-throughput sequencing in different read mapping applications [Li and Durbin, 2009; Langmead *et al.*, 2009; Langmead and Salzberg, 2012] and allows to traverse the prefix trie of a text. Currently, we are integrating it into our framework and provide prefix trie iterators to ease the development of FM index based algorithms. Another interesting direction is dynamic indexing [Salson *et al.*, 2009, 2010], i.e. to update an index according to text changes. This approach not only saves the time required for constructing an index from scratch, it could also be used to determine and efficiently represent the changes a set of similar texts would induce on a reference index. We are developing a data structure that, instead of applying these changes directly, allows to access the (virtual) index of each text.

Our read mapping approach is with slight modifications also applicable to the dinucleotide based ABI/SOLiD sequencing technology. Therefore the reference sequence must be converted into *color space*, i.e. into a sequence of 4 dinucleotide colors instead of the 4 DNA bases, and the semi-global alignment of color-space reads could be adapted as proposed in [Rumble *et al.*, 2009]. Additionally, base-call qualities could not only be used for sensitivity control, but also to optionally rank the read alignments by their plausibil-

ity instead of the number of errors [Li *et al.*, 2008a]. We also plan to use SIMD extensions [Intel, 2011] and hardware accelerators, e.g. GPUs and FPGAs, to massively parallelize the verification of candidate regions.

Depending on the problem at hand, the implementation of our algorithm for frequency string mining could be improved. If the DFI should only be used to output the result of $Th(pred)$, the memory consumption of the algorithm could be further reduced. As each node is visited at most once, at any time only nodes of the suffix tree on the path from the *root* to the current node need to be stored. A small alphabet (e.g. DNA) leads to a dense suffix tree with many branching nodes at the top, as observed by Kurtz [1999]. In that case, an improvement in running time could be expected by replacing the top of the suffix tree with a q-gram index and in parallel traverse multiple q-gram buckets. In this way, the memory consumption could be improved by keeping in memory only the traversed subtree. Considering additional constraints during the mining process will play an important role in further algorithmic development, e.g. reducing the solution space of any mining approach to a succinct but representative set is one of the open challenges, as mentioned by Han *et al.* [2007]. For example, Kobyliński and Walczak [2009] aggregate all *minimal* jumping emerging substrings to train discriminative image classifiers. The top down construction of the DFI could limited to right minimal jumping emerging substrings. To check for left minimality would require either the use of suffix links [Ukkonen, 1995] or an additional post processing step. Another venue is to combine the framework of frequency based string mining with probabilistic automata that can be used to classify sequences, e.g. to build discriminative models as presented by Slonim *et al.* [2003]. Due to the efficiency of the presented approach it is possible now to construct probabilistic automata for a set of databases in expected linear time as an extension to our previous work [Schulz *et al.*, 2008b].

Appendix A
Appendix

A.1 High-throughput sequencing technologies in detail

In the following, we explain the mechanisms and characteristics of the three most prevalent technologies (according to [Kodama *et al.*, 2012]) and in brief describe the SMRT™ and HeliScope™ single molecule sequencing technologies. More details can be found in [Janitz, 2008; Mardis, 2008; Shendure and Ji, 2008].

Illumina

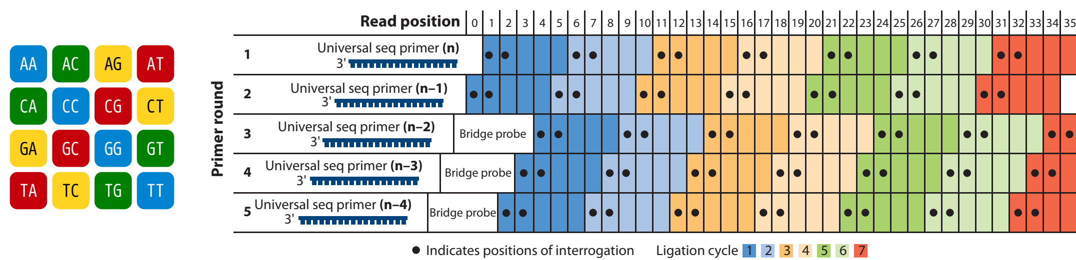
Illumina sequencing uses a cycle-based sequencing-by-synthesis approach. The DNA sample of interest is first fractionated by nebulization or sonication into smaller double-stranded fragments. After blunt-ending and phosphorylating, two unique adapters are ligated to the ends of the fragments. An eight-lane flow cell, whose surface is coated with single-stranded primers that correspond to the adapter sequences, is used to hybridize the single strands of the adapter-ligated fragments and bind them to the flow cell surface. In a process called bridge PCR these fragments are amplified to clusters, i.e. local spots of ≈1,000 identical copies of a single fragment.

The flow cell now contains millions of unique clusters and is sequenced in cycles. In each cycle fluorescently labeled nucleotides are added to the flow cell. Each nucleotide is a reversible terminator such that only one is incorporated to each nucleic acid chain in each cycle. After the single-base extension, the labeled nucleotides are excited by a laser and their emitted light is captured by a CCD camera, whereby the identical nucleotides in the clusters work as amplifiers. Before starting the next cycle the fluorescent labels are removed and the incorporated nucleotide is unblocked.

At the end all images are aligned, where clusters correspond to signals at identical image positions across the cycles. The intensities of the four colors in the i-th image at a certain cluster position are used to base-call the i-th base of the corresponding read and assign a quality score.

SOLiD

In contrast to Illumina sequencing, SOLiD (Sequencing by Oligonucleotide Ligation and Detection) is a cycle-based sequencing by ligation. The DNA sample is first fractionated into smaller fragments, which are then adapter-ligated. For the amplification, the



***(a)** color labels* ***(b)** primer rounds and ligation cycles*

Figure A.1: *SOLiD color labels of the 16 dinucleotides (a). The labeling allows to convert a sequence of overlapping dinucleotide colors into a sequence of bases, if one involved base is known. Fragments are sequenced in rounds of multiple ligation cycles (b). The universal primer is shortened after each round to interrogate all bases. Image by Mardis [2008].*

ABI/SOLiD platform uses emulsion PCR [Dressman *et al.*, 2003], where small magnetic beads are enclosed by water compartments in a water-in-oil emulsion. Thousands of primers corresponding to one of the adapters are tethered to the bead surface. The compartments work as microreactors and contain all reagents required for PCR. Through limited dilution, each bead-containing compartment include at most one fragment which is amplified on the bead surface. At the end of the amplification, each bead is coated with millions of copies of the original single-stranded adapter-ligated fragment. After breaking the emulsion, the beads are separated from the micro reactors using magnetic bead purification. The free 3' ends of the fragments are then chemically attached to a flow cell slide.

Prior to the first sequencing cycle, a universal primer that corresponds to the adapter is annealed at the 5' end of each amplified fragment. SOLiD uses a sequencing-by-ligation technique. A pool of 1024 octamer primers with all possible combinations of A, C, G, and T at the first 5 positions is fluorescently labeled according to the dinucleotide at the first 2 positions (at the 3' end). The 16 possible dinucleotides are mapped to 4 different colors as shown in Figure A.1a. In each cycle only one primer anneals to the 5' end of the nucleic acid chain. Then the flow cell is laser excited and imaged by a CCD camera. At the end of the cycle the last 3 bases of the ligated primers and the fluorescent labels are removed and the next cycle follows.

As in every cycle effectively 5-mers are ligated, only fragment bases at positions $1+5i$ and $2+5i$ can be examined. To determine the remaining bases, the whole sequencing step is repeated 4 times with a universal adapter that is one base shorter than the previous round, such that positions $0+5i$ and $1+5i$ can be examined in the second round and $4+5i$ and $5+5i$ in the third, and so on (see Figure A.1b)

At the end of the 5 sequencing rounds, all overlapping dinucleotides in a fragment prefix have been imaged. Analogously to Illumina sequencing, the images are aligned to identify beads, their emitted colors and corresponding quality scores. The result of the base-calling step is not a set of reads in base space (i.e. bases are A, C, G, or T) but in color space (bases are 0, 1, 2, or 3 representing colors).

Roche/454

Roche/454 sequencing, commercially available since 2004, uses a cycled pyrosequencing [Ronaghi *et al.*, 1996] and the same technique for sample preparation as SOLiD. Beginning with fragmentation and adapter ligation, the templates are then amplified on the surface of magnetic beads by emulsion PCR, after which each bead is coated by a million of copies of one DNA fragment. The beads are separated from the emulsion and distributed over a picotiter plate, whose surface is covered by millions of wells, where each provides space for only a single bead.

The actual sequencing is performed by the pyrosequencing method [Ronaghi *et al.*, 1996], in which luciferase and other enzymes are used to generate light from the polymerase-driven incorporation of nucleotides. In a fixed order of cycles the plate is flown with pure nucleotide solutions (e.g. beginning with `A`, followed by `G`,`C`,`T`,`A`,`G`,`C`,`T`,...). Wells in which one or more nucleotides are incorporated, emit light which is captured by a CCD camera at the bottom of the plate. The light intensity is proportional to the number of incorporated bases and must be used to infer the length of homopolymer stretches, as the incorporated nucleotides contain no terminating moiety. The sequence of the ligated adapter starts with `TCGA`, which allows measuring the intensities of single nucleotide incorporations for each well to calibrate the base-calling software. However, the base call accuracy deteriorates on large homopolymer runs (>6 bp). After the imaging, the unincorporated nucleotides are removed by an apyrase wash and the next cycle continues with the next nucleotide solution.

SMRT™

Pacific Biosciences introduced in 2010 a single molecule real time (SMRT) sequencer that enables sequencing a contiguous piece of length ≈1500 bp of a single molecule without prior amplification. The fundamental idea is to immobilize DNA polymerase and to film the incorporation of fluorescently labeled nucleotides in real time. As the sequencing is not cycled, the base-calling cannot accurately determine the length of homopolymer runs which must be inferred from signal lengths. However, this new approach permits sequencing reads of length similar to first generation sequencing and promises to detect methylated bases from deviations in the signal length.

HeliScope™

HeliScope™ sequencing is a combination of Illumina and Roche/454 sequencing. Like PacBio, it does not require fragment amplification and uses sequencing by synthesis with nucleotides that contain a terminating moiety. Instead all four nucleotides being added simultaneously to the flow cell, they are added in separate cycles (like 454). The imaging and base-calling steps are similar to Illumina sequencing.

A.2 Proving sensitivity recursions

In this section, we prove the correctness the recursions proposed in Lemma 6.2 and Lemma 6.3 (Section 6.5.1 on page 105) for the sensitivity computation of q-gram counting filters.

Proof of Lemma 6.2. Let $\mathcal{T}(i,e,t,T_2) \subseteq \Phi^i$, with $\Phi = \{\mathtt{M},\mathtt{R}\}$, be the set of Hamming transcripts with e errors, s.t. for every $T_1 \in \mathcal{T}(i,e,t,T_2)$ the concatenation T_1T_2 contains at least t substrings $\mathtt{M}^q$. For $i<0$, $e<0$, or $t<0$ we define $\mathcal{T}(i,e,t,T_2)=\emptyset$.

Randomly choose $i,e,t \in \mathbb{N}_0$, $i>0$, $T_2 \in \Phi^q$, and $T_1 \in \mathcal{T}(i,e,t,T_2)$. Now let $x,y \in \Phi$ be the last characters of T_1 and T_2 such that for appropriate $T'_1 \in \Phi^{i-1}$, $T'_2 \in \Phi^{q-1}$ holds $T_1 = T'_1x$, $T_2 = T'_2y$. As $T_1T_2 = T'_1xT'_2y$ contains at least t substrings $\mathtt{M}^q$ it follows that $T'_1xT'_2$ contains at least $t-\delta(T'_2y)$. Additionally, it holds that $e = \|T_1\|_{\mathrm{E}} = \|T'_1x\|_{\mathrm{E}} = \|T'_1\|_{\mathrm{E}} + \|x\|_{\mathrm{E}}$ and thus $\|T'_1\|_{\mathrm{E}} = e$, if $x=\mathtt{M}$, and $\|T'_1\|_{\mathrm{E}} = e-1$, if $x = \mathtt{R}$. Because $\mathrm{shift}(x,T_2) = xT'_2$ it follows $T'_1 \in \mathcal{T}(i-1,e,t-\delta(T_2),\mathrm{shift}(\mathtt{M},T_2))$ or $T'_1 \in \mathcal{T}(i-1,e-1,t-\delta(T_2),\mathrm{shift}(\mathtt{R},T_2))$ and thus:

$$\begin{aligned}\mathcal{T}(i,e,t,T_2) \subseteq \;& \mathcal{T}(i-1,e\phantom{{}-1},t-\delta(T_2),\mathrm{shift}(\mathtt{M},T_2))\mathtt{M} \\ \cup\;& \mathcal{T}(i-1,e-1,t-\delta(T_2),\mathrm{shift}(\mathtt{R},T_2))\mathtt{R}.\end{aligned} \tag{A.1}$$

Now, randomly choose $i',e',t' \in \mathbb{N}_0$, $x \in \Phi$, $T_2 \in \Phi^q$, and $T'_1 \in \mathcal{T}(i',e',t',\mathrm{shift}(x,T_2))$. It holds $|T'_1x| = i'+1$, $\|T'_1x\|_{\mathrm{E}} = e' + \|x\|_{\mathrm{E}}$, and if $T'_1\mathrm{shift}(x,T_2) = T'_1xT_2[0..|T_2|-1)$ contains at least t' substrings $\mathtt{M}^q$, then T'_1xT_2 contains at least $t'+\delta(T_2)$. Therefore, it follows that $T'_1x \in \mathcal{T}(i'+1,e'+\|x\|_{\mathrm{E}},t'+\delta(T_2),T_2)$ and thus:

$$\begin{aligned}\mathcal{T}(i,e,t,T_2) \supseteq \;& \mathcal{T}(i-1,e\phantom{{}-1},t-\delta(T_2),\mathrm{shift}(\mathtt{M},T_2))\mathtt{M} \\ \cup\;& \mathcal{T}(i-1,e-1,t-\delta(T_2),\mathrm{shift}(\mathtt{R},T_2))\mathtt{R}.\end{aligned} \tag{A.2}$$

By the definition of R it holds that $R(i,e,t,T_2) = \sum_{T_1\in\mathcal{T}(i,e,t,T_2)} p(T_1)$. Applied to (A.1) and (A.2), (6.6) follows.

T_2 contains exactly $\delta(T_2)$ substrings $\mathtt{M}^q$, therefore $\mathcal{T}(0,e,t,T_2) = \{\epsilon\}$ if $e=0$ and $0 \le t \le \delta(T_2)$, otherwise $\mathcal{T}(0,e,t,T_2)=\emptyset$. With $p(\epsilon)=1$, (6.5) follows. ∎

Proof of Lemma 6.3. This lemma can be proven analogously to the proof above. Let $\Phi = \{\mathtt{M},\mathtt{R},\mathtt{D},\mathtt{I}\}$ and $\mathcal{T}(i,e,t,T_2) \subseteq \Phi(i)$ be the set of transcripts with e errors, s.t. for every $T_1 \in \mathcal{T}(i,e,t,T_2)$, T_1T_2 contains at least t substrings $\mathtt{M}^q$.

Randomly choose $i,e,t \in \mathbb{N}_0$, $i>0$, $T_2 \in \Phi(q)$, $T_2[|T_2|-1] \neq \mathtt{I}$, and $T_1 \in \mathcal{T}(i,e,t,T_2)$. Let $x,y \in \Phi$ be the last characters of T_1 and T_2 such that for appropriate $T'_1,T'_2 \in \Phi^*$ holds $T_1 = T'_1x$, $T_2 = T'_2y$. Now it holds that $\|T'_1\|_{\mathrm{R}} = i - \|x\|_{\mathrm{R}}$ and $\|T'_1\|_{\mathrm{E}} = e - \|x\|_{\mathrm{E}}$. Additionally, $T'_1xT'_2$ and thus also $T'_1\mathrm{shift}(x,T'_2y)$ contain at least $t-\delta(T'_2y)$ substrings $\mathtt{M}^q$. This proves the "$\subseteq$" part of (6.14). We omit the analogue rest of the proof. ∎

A.3 Read mapper parametrization

In the following we describe the parameters we used for the comparison with other read mappers. `MIN` and `MAX` were placeholders for minimal and maximal insert size, `INS` is the mean insert size and `IERR` the allowed deviation (`INS = (MIN + MAX) / 2`, `IERR = (MAX - MIN) / 2`). For the tools using indices, we built the index using default options.

Bowtie 2. Version 2.0.0-beta6 was used. The number of threads was selected using the parameter `-p`. We used the parameter `--end-to-end` to enforce semi-global read alignments. For the Rabema experiment, we used the parameters `-k 100`. For all other experiments, we used the parameters `-k 1`. In paired-end mode, we used the parameters `--minins MIN --maxins MAX`.

BWA. Version 0.6.1-r104 was used. We used the parameter `-t` to select the number of reads in the `aln` step. The `sampe` and `samse` steps were performed using one thread since BWA does not offer a parallelization here. When mapping for the Rabema experiment, we passed the parameter `-N` to `aln` and `-n 100` to `samse`. Otherwise, we passed the parameter `-n 1` to `samse`. The insert size was not passed to BWA, however we pass the insert size and allowed error from BWA's output to the other read mappers.

Hobbes. Version 1.3 was used. Since we focus on edit distance, we used the 16-bit bit-vector version as described in [Ahmadi *et al.*, 2012]. We built the index using the recommended[1] q-gram length 11. Indels were enabled using `--indels`. Maximal edit distance was set using `-v`. Multi-threading was enabled using `-p`. For resource measurement, we used the output without CIGAR, for analyzing the results, we enabled CIGAR output using `--cigar`. In paired-end mode, we used the parameters `--pe --min MIN --max MAX`.

mrFAST. Version 2.1.0.6 was used. It was used as explained in the manual[2]. mrFAST does not support multithreading. We divided the input into blocks of 500 k reads and processed each chunk in a separate process using the program `ts`[3]. Long reads were split into packages of 100 k reads. This way, always 8 processes were executed in parallel. We set the edit distance error rate to 4 % of the read length.

RazerS. Version 3.1 was used. RazerS was parametrized as follows: The native or SAM output format was selected with `-of 0` or `-of 4`. Indel support was disabled with `--no-gaps` when required. The number of threads was set with the `-tc` parameter. The percent recognition rate was set using the `-rr` parameter, e.g. `-rr 100` or `-rr 99`. The error rate was set through the `-i` parameter, e.g. `-i 96` to map with 4 % errors[4]. The pigeonhole or SWIFT filter was selected using `-fl pigeonhole` or `-fl swift`. As an all-mapper, the parameter `-m 1000000` was used and as a best-mapper `-m 1` was used. In paired-end mode, the parameters used were `--library-length INS --library-error IERR`.

SHRiMP 2. Version 2.2.2 was used. The number of threads was selected with `--threads`. In paired-end mode, the options used are `--pair-mode opp-in --isize MIN,MAX`.

Soap 2. Version 2.1 was used. The number of threads was selected with `-p`. In paired-end mode, the options used are `-m MIN -x MAX`.

[1] http://hobbes.ics.uci.edu/manual.jsp

[2] http://mrfast.sourceforge.net/manual.html

[3] http://vicerveza.homeunix.net/~viric/soft/ts/

[4] RazerS uses the percent identity, which is 100 minus error rate in percents.

A.4 Extended variation detection tables

		(0,0)		(1,0)		(2,0)		(3,0)		(4,0)		(1,1)		(1,2)		(0,3)		(0,4)	
	method	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.
best-mappers	Bowtie 2	97.6	97.3	95.6	94.8	94.6	92.0	93.3	88.7	92.6	82.5	95.3	93.3	93.5	92.3	96.1	95.4	97.6	97.4
	BWA	98.2	97.9	97.1	96.4	97.6	95.3	96.5	90.2	94.9	85.1	97.4	90.9	97.1	80.3	96.3	66.5	97.5	67.1
	Soap 2	98.1	82.9	97.0	63.6	97.4	31.0	0.0	0.0	0.0	0.0	90.6	6.2	0.0	0.0	0.0	0.0	0.0	0.0
	R-100	98.4	98.4	97.7	97.7	98.2	98.2	97.5	97.5	96.3	96.3	98.1	98.1	97.9	97.9	97.6	97.6	98.4	98.4
	R-99	98.4	98.4	97.7	97.7	98.2	98.0	97.4	96.6	96.2	95.1	98.2	98.1	97.9	97.9	97.6	97.6	98.4	98.4
	R-95	98.4	98.3	97.7	97.5	98.2	97.3	97.5	94.9	96.1	91.7	98.2	97.6	97.9	97.6	97.5	97.5	98.4	98.4
all-mappers	Hobbes	99.9	99.9	99.9	99.9	99.9	99.9	99.9	99.9	100.0	100.0	100.0	99.8	100.0	93.6	99.6	90.5	99.6	87.6
	mrFAST	100.0	99.9	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	99.3
	SHRiMP 2	100.0	99.4	100.0	99.5	100.0	99.7	100.0	99.9	100.0	99.7	100.0	99.5	100.0	99.2	100.0	99.6	100.0	99.6
	R-100	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	R-99	100.0	100.0	100.0	99.9	100.0	99.8	100.0	99.1	100.0	98.9	100.0	99.9	100.0	100.0	100.0	100.0	100.0	100.0
	R-95	100.0	99.9	100.0	99.7	100.0	99.0	100.0	97.3	100.0	95.4	100.0	99.4	100.0	99.6	100.0	99.9	100.0	100.0

Table A.1: *Full variation detection results for* ***single-end reads*** *(Section 6.10.5). This table extends Table 6.4a on page 128.*

		(0,0)		(2,0)		(4,0)		(6,0)		(8,0)		(2,2)		(2,4)		(0,5)		(0,7)	
	method	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.	prec.	recl.
best-mappers	Bowtie 2	98.8	98.8	98.6	98.1	98.3	96.4	97.3	93.0	100.0	92.0	98.5	97.6	98.3	97.5	98.5	98.3	97.5	97.5
	BWA	99.0	98.4	99.0	96.7	99.2	93.5	99.3	86.6	100.0	80.0	98.5	79.9	100.0	65.2	99.1	69.0	100.0	60.0
	Soap 2	98.7	95.5	98.8	83.9	98.9	53.2	100.0	8.3	0.0	0.0	97.3	53.3	96.6	46.3	98.5	79.7	0.0	0.0
	R-100	99.0	99.0	99.0	99.0	99.0	99.0	98.7	98.7	100.0	100.0	99.7	99.7	99.6	99.6	99.0	99.0	100.0	100.0
	R-99	99.0	99.0	99.0	98.8	99.2	98.6	98.7	96.2	100.0	92.0	99.4	98.5	99.2	99.2	98.7	98.7	100.0	100.0
	R-95	99.0	98.9	99.0	98.3	99.1	96.3	98.7	93.6	100.0	88.0	99.1	97.9	100.0	99.2	98.7	98.6	100.0	100.0
all-mappers	Hobbes	97.5	93.2	97.6	93.7	98.0	94.6	95.3	92.7	100.0	100.0	96.5	80.1	99.5	86.0	97.7	85.2	100.0	86.4
	mrFAST	98.8	98.8	98.9	98.9	98.9	98.9	98.7	98.7	100.0	100.0	99.1	99.1	98.8	98.8	91.8	7.8	96.3	65.0
	SHRiMP 2	100.0	99.7	100.0	99.7	100.0	99.9	100.0	100.0	100.0	100.0	100.0	99.7	100.0	99.6	100.0	99.7	100.0	100.0
	R-100	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
	R-99	100.0	100.0	100.0	99.8	100.0	99.4	100.0	97.5	100.0	92.0	100.0	99.1	100.0	100.0	100.0	100.0	100.0	100.0
	R-95	100.0	99.9	100.0	99.2	100.0	97.3	100.0	94.9	100.0	88.0	100.0	98.8	100.0	99.2	100.0	99.9	100.0	100.0

Table A.2: *Full variation detection results for* ***paired-end reads****. This table extends Table 6.4b.*

A.5 Extended performance comparison tables

	dataset	ERR022075 E. coli					SRR065390 C. elegans				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	1:19	15:21	162	99.77 (100.00 99.52 97.51 / 94.39 90.72)	99.32 (84.64 93.43 95.73 / 96.84 97.51)	2:08	25:14	277	99.38 (100.00 99.30 93.38 / 88.61 84.03)	92.58 (75.01 83.74 86.20 / 87.61 88.57)
	BWA	3:04	13:17	184	99.78 (100.00 99.51 97.52 / 94.41 91.83)	97.98 (84.64 93.43 95.72 / 96.85 97.52)	5:21	36:50	325	99.46 (100.00 99.09 95.57 / 89.70 85.86)	89.33 (75.01 83.72 86.25 / 87.64 88.59)
	Soap 2	1:30	3:16	729	98.00 (100.00 99.09 96.75 / 0.46 0.12)	95.68 (84.64 93.39 95.67 / 95.68 95.68)	1:32	5:26	813	96.74 (100.00 96.57 92.38 / 0.33 0.04)	85.95 (75.01 83.50 85.94 / 85.95 85.95)
	R-100	0:50	2:00	5705	100.00 (100.00 100.00 100.00 / 100.00 100.00)	97.59 (84.64 93.47 95.79 / 96.92 97.59)	1:42	11:42	5841	100.00 (100.00 100.00 100.00 / 100.00 100.00)	88.79 (75.01 83.80 86.38 / 87.83 88.79)
	R-99	0:48	1:59	5705	100.00 (100.00 100.00 100.00 / 99.99 99.94)	97.59 (84.64 93.47 95.79 / 96.92 97.59)	1:38	11:17	5841	100.00 (100.00 100.00 100.00 / 99.99 99.80)	88.78 (75.01 83.80 86.38 / 87.83 88.78)
	R-95	0:47	1:57	5705	100.00 (100.00 100.00 100.00 / 99.87 99.56)	97.59 (84.64 93.47 95.79 / 96.92 97.59)	1:30	9:52	5841	99.98 (100.00 100.00 100.00 / 99.75 98.81)	88.77 (75.01 83.80 86.38 / 87.82 88.77)
all-mappers	Hobbes	2:02	12:55	769	91.46 (91.45 91.42 92.39 / 91.30 90.88)	89.28 (77.40 85.49 87.63 / 88.66 89.28)	26:46	287:26	3136	93.27 (93.21 93.86 93.69 / 93.42 90.88)	82.83 (69.92 78.18 80.60 / 81.95 82.83)
	mrFAST	0:36	4:30	8269	100.00 (100.00 100.00 100.00 / 100.00 99.99)	97.59 (84.64 93.47 95.79 / 96.92 97.59)	6:01	57:42	10497	100.00 (100.00 99.98 99.91 / 99.98 100.00)	88.79 (75.01 83.80 86.38 / 87.83 88.79)
	SHRiMP 2	4:17	49:42	1553	99.86 (99.99 99.69 98.66 / 97.11 94.49)	99.28 (84.63 93.44 95.72 / 96.83 97.48)	52:44	608:58	3372	98.70 (99.59 96.81 91.76 / 87.60 81.88)	91.91 (74.71 83.22 85.59 / 86.88 87.69)
	R-100	0:50	2:10	5705	100.00 (100.00 100.00 100.00 / 100.00 100.00)	97.59 (84.64 93.47 95.79 / 96.92 97.59)	2:20	16:57	9203	100.00 (100.00 100.00 100.00 / 100.00 100.00)	88.79 (75.01 83.80 86.38 / 87.83 88.79)
	R-99	0:49	2:09	5705	100.00 (100.00 100.00 100.00 / 99.99 99.94)	97.59 (84.64 93.47 95.79 / 96.92 97.59)	2:16	16:07	9036	100.00 (100.00 100.00 100.00 / 99.99 99.80)	88.78 (75.01 83.80 86.38 / 87.83 88.78)
	R-95	0:49	2:08	5705	100.00 (100.00 100.00 100.00 / 99.87 99.56)	97.59 (84.64 93.47 95.79 / 96.92 97.59)	2:11	14:24	8598	99.98 (100.00 100.00 100.00 / 99.75 98.81)	88.77 (75.01 83.80 86.38 / 87.82 88.77)

	dataset	SRR497711 D. melanogaster					ERR012100 H. sapiens				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	2:00	23:43	299	99.65 (100.00 99.77 99.02 / 97.65 94.92)	85.71 (52.08 67.27 73.62 / 76.88 78.81)	5:37	66:40	3374	99.62 (100.00 99.75 96.02 / 92.88 87.86)	96.72 (75.99 87.81 90.54 / 91.85 92.76)
	BWA	5:35	36:30	413	98.96 (100.00 99.57 98.40 / 90.72 82.08)	79.37 (52.08 67.24 73.57 / 76.62 78.31)	13:45	98:30	4475	99.66 (100.00 99.50 98.01 / 93.39 88.92)	93.53 (75.99 87.78 90.59 / 91.91 92.82)
	Soap 2	1:55	8:37	900	91.78 (100.00 96.24 89.35 / 0.09 0.02)	72.49 (52.08 66.73 72.48 / 72.49 72.49)	2:34	15:11	5481	96.45 (100.00 94.94 86.54 / 0.32 0.16)	89.73 (75.99 87.24 89.73 / 89.73 89.73)
	R-100	1:28	9:45	5795	100.00 (100.00 100.00 100.00 / 100.00 100.00)	78.92 (52.08 67.31 73.69 / 76.97 78.92)	85:56	1001:18	9096	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.99 (75.99 87.84 90.67 / 92.02 92.99)
	R-99	1:26	9:14	5795	99.98 (100.00 100.00 100.00 / 99.93 99.32)	78.90 (52.08 67.31 73.69 / 76.96 78.90)	73:09	848:59	8679	99.99 (100.00 100.00 100.00 / 99.94 99.41)	92.98 (75.99 87.84 90.67 / 92.02 92.98)
	R-95	1:26	9:24	5795	99.87 (100.00 100.00 100.00 / 99.11 96.34)	78.82 (52.08 67.31 73.69 / 76.94 78.82)	43:16	493:42	7640	99.96 (100.00 100.00 100.00 / 99.30 96.92)	92.95 (75.99 87.84 90.67 / 92.01 92.95)
all-mappers	Hobbes	4:51	39:57	2141	96.49 (96.55 96.46 96.94 / 96.28 93.86)	76.16 (50.28 64.98 71.16 / 74.33 76.16)	265:48	2851:00	70683	95.97 (95.94 96.14 96.39 / 96.10 94.63)	89.24 (72.90 84.30 87.02 / 88.33 89.24)
	mrFAST	4:01	37:32	10844	100.00 (100.00 100.00 100.00 / 100.00 100.00)	78.92 (52.08 67.31 73.69 / 76.97 78.92)	413:40	3987:49	11324	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.99 (75.99 87.84 90.67 / 92.02 92.99)
	SHRiMP 2	23:40	255:09	3801	99.83 (99.99 99.99 99.74 / 98.71 96.33)	89.91 (52.07 67.30 73.66 / 76.92 78.83)	1312:09	14466:34	38188	99.81 (99.89 99.83 99.39 / 98.29 96.81)	99.06 (75.90 87.74 90.56 / 91.91 92.87)
	R-100	1:51	11:59	7329	100.00 (100.00 100.00 100.00 / 100.00 100.00)	78.92 (52.08 67.31 73.69 / 76.97 78.92)	118:26	1384:18	15298	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.99 (75.99 87.84 90.67 / 92.02 92.99)
	R-99	1:49	11:21	7302	99.98 (100.00 100.00 100.00 / 99.93 99.32)	78.90 (52.08 67.31 73.69 / 76.96 78.90)	100:19	1169:22	15238	99.99 (100.00 100.00 100.00 / 99.94 99.41)	92.98 (75.99 87.84 90.67 / 92.02 92.98)
	R-95	1:45	10:40	7270	99.87 (100.00 100.00 100.00 / 99.11 96.34)	78.82 (52.08 67.31 73.69 / 76.94 78.82)	58:13	665:47	14384	99.96 (100.00 100.00 100.00 / 99.30 96.92)	92.95 (75.99 87.84 90.67 / 92.01 92.95)

Table A.3: *Extended experimental results for real-world* ***single-end*** *data, extending Table 6.5a on page 129.*

dataset		simulated, $m = 200$ D. melanogaster				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	0:29	5:23	318	99.46 (100.00 99.86 99.47 / 97.23 88.00)	99.70 (12.51 61.89 86.40 / 91.42 92.75)
best-mappers	BWA	1:09	6:41	440	98.49 (100.00 99.82 98.45 / 90.30 67.15)	91.83 (12.51 61.87 86.11 / 90.78 91.82)
best-mappers	Soap 2	0:21	1:40	1074	65.88 (100.00 96.41 4.12 / 0.008 0.00)	61.20 (12.51 60.18 61.20 / 61.20 61.20)
best-mappers	R-100	0:46	6:32	1529	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.87 (12.51 61.94 86.47 / 91.50 92.87)
best-mappers	R-99	0:46	6:59	1529	100.00 (100.00 100.00 100.00 / 99.98 99.88)	92.87 (12.51 61.94 86.47 / 91.50 92.87)
best-mappers	R-95	0:46	6:34	1529	99.99 (100.00 100.00 100.00 / 99.93 99.63)	92.86 (12.51 61.94 86.47 / 91.50 92.86)
all-mappers	mrFAST	1:11	10:29	6909	99.24 (100.00 100.00 100.00 / 96.34 62.09)	92.17 (12.51 61.94 86.47 / 91.32 92.17)
all-mappers	SHRiMP 2	7:25	64:30	3758	99.49 (99.98 99.97 99.78 / 96.97 81.86)	99.96 (12.51 61.93 86.41 / 91.36 92.60)
all-mappers	R-100	0:49	6:55	1529	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.87 (12.51 61.94 86.47 / 91.50 92.87)
all-mappers	R-99	0:48	6:19	1529	100.00 (100.00 100.00 100.00 / 99.98 99.88)	92.87 (12.51 61.94 86.47 / 91.50 92.87)
all-mappers	R-95	0:48	6:43	1529	99.99 (100.00 100.00 100.00 / 99.93 99.63)	92.86 (12.51 61.94 86.47 / 91.50 92.86)

dataset		simulated, $m = 400$ D. melanogaster				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	1:35	18:15	355	98.68 (100.00 99.80 99.12 / 91.93 83.01)	99.94 (1.56 52.43 82.17 / 86.62 89.21)
best-mappers	BWA	2:40	18:03	935	93.95 (100.00 99.59 96.69 / 63.29 4.45)	84.57 (1.56 52.31 81.30 / 84.38 84.57)
best-mappers	Soap 2	0:38	3:40	1554	56.07 (100.00 79.04 27.73 / 2.09 0.22)	50.99 (1.56 41.87 50.37 / 50.69 50.92)
best-mappers	R-100	0:56	7:05	2674	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.47 (1.56 52.48 82.26 / 86.79 89.47)
best-mappers	R-99	0:56	6:46	2674	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.47 (1.56 52.48 82.26 / 86.79 89.47)
best-mappers	R-95	0:55	6:51	2674	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.47 (1.56 52.48 82.26 / 86.79 89.47)
all-mappers	mrFAST	2:17	20:54	7972	94.15 (100.00 100.00 98.39 / 54.15 0.00)	85.02 (1.56 52.48 81.79 / 85.02 85.02)
all-mappers	SHRiMP 2	42:53	484:02	3814	98.44 (99.95 99.91 99.21 / 88.64 77.47)	99.85 (1.55 52.44 82.09 / 86.39 88.83)
all-mappers	R-100	0:58	6:57	2674	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.47 (1.56 52.48 82.26 / 86.79 89.47)
all-mappers	R-99	0:58	7:24	2674	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.47 (1.56 52.48 82.26 / 86.79 89.47)
all-mappers	R-95	0:55	6:42	2674	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.47 (1.56 52.48 82.26 / 86.79 89.47)

dataset		simulated, $m = 800$ D. melanogaster				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	13:48	164:45	628	96.73 (97.47 99.67 98.05 / 87.95 85.17)	99.99 (0.03 41.07 73.95 / 82.31 90.03)
best-mappers	BWA	5:38	36:44	1055	74.96 (97.47 98.62 82.47 / 0.00 0.00)	68.09 (0.03 40.61 68.09 / 68.09 68.09)
best-mappers	Soap 2	0:54	4:33	1546	41.21 (97.47 67.99 28.10 / 0.22 0.00)	38.14 (0.03 28.17 37.88 / 38.14 38.14)
best-mappers	R-100	1:17	8:10	4963	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)
best-mappers	R-99	1:17	7:52	4963	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)
best-mappers	R-95	1:15	7:52	4963	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)
all-mappers	mrFAST	5:16	49:22	9115	65.25 (93.14 95.65 59.59 / 0.00 0.00)	69.32 (0.03 39.34 69.32 / 69.32 69.32)
all-mappers	SHRiMP 2	796:06	9442:21	4023	95.70 (97.47 99.75 97.60 / 82.95 80.14)	99.31 (0.03 41.04 73.67 / 81.63 89.14)
all-mappers	R-100	1:20	8:48	4963	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)
all-mappers	R-99	1:20	8:38	4963	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)
all-mappers	R-95	1:20	8:35	4963	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.43 (0.03 41.13 74.13 / 82.65 90.43)

dataset		simulated, $m = 200$ H. sapiens				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	1:19	15:13	3372	99.02 (100.00 99.58 98.62 / 96.32 87.23)	99.57 (12.57 61.74 86.07 / 91.17 92.54)
best-mappers	BWA	4:53	44:30	6619	98.44 (100.00 99.81 98.36 / 90.31 66.51)	91.82 (12.57 61.85 86.05 / 90.78 91.81)
best-mappers	Soap 2	0:49	5:08	5651	65.82 (100.00 96.41 3.99 / 0.004 0.00)	61.15 (12.57 60.17 61.15 / 61.15 61.15)
best-mappers	R-100	26:34	299:17	3341	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.88 (12.57 61.93 86.43 / 91.51 92.88)
best-mappers	R-99	16:17	179:00	3341	100.00 (100.00 100.00 100.00 / 99.98 99.85)	92.88 (12.57 61.93 86.43 / 91.51 92.88)
best-mappers	R-95	13:55	153:13	3341	99.99 (100.00 100.00 100.00 / 99.94 99.63)	92.87 (12.57 61.93 86.43 / 91.51 92.87)
all-mappers	mrFAST	103:02	1024:12	7395	99.25 (100.00 100.00 100.00 / 96.47 61.90)	92.18 (12.57 61.93 86.43 / 91.34 92.18)
all-mappers	SHRiMP 2	546:29	5323:53	38570	99.32 (99.78 99.79 99.60 / 96.96 82.05)	99.95 (12.54 61.80 86.23 / 91.23 92.48)
all-mappers	R-100	28:30	323:07	3341	100.00 (100.00 100.00 100.00 / 100.00 100.00)	92.88 (12.57 61.93 86.43 / 91.51 92.88)
all-mappers	R-99	17:22	191:32	3341	100.00 (100.00 100.00 100.00 / 99.98 99.85)	92.88 (12.57 61.93 86.43 / 91.51 92.88)
all-mappers	R-95	14:40	160:48	3341	99.99 (100.00 100.00 100.00 / 99.94 99.63)	92.87 (12.57 61.93 86.43 / 91.51 92.87)

dataset		simulated, $m = 400$ H. sapiens				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	4:25	52:06	3413	98.56 (100.00 99.70 98.96 / 91.74 83.08)	99.94 (1.59 52.46 82.11 / 86.61 89.24)
best-mappers	BWA	20:21	185:24	11043	93.64 (100.00 99.53 96.06 / 62.45 4.39)	84.33 (1.59 52.36 81.10 / 84.15 84.33)
best-mappers	Soap 2	1:37	12:30	5938	56.22 (100.00 79.19 27.86 / 2.00 0.17)	51.09 (1.59 42.00 50.48 / 50.79 51.02)
best-mappers	R-100	44:21	511:05	3532	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.53 (1.59 52.57 82.28 / 86.83 89.53)
best-mappers	R-99	20:31	226:47	3532	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.53 (1.59 52.57 82.28 / 86.83 89.53)
best-mappers	R-95	20:31	227:05	3532	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.53 (1.59 52.57 82.28 / 86.83 89.53)
all-mappers	mrFAST	321:50	4143:26	7926	65.33 (69.17 69.39 68.36 / 37.33 0.00)	59.38 (1.10 36.49 56.87 / 59.38 59.38)
all-mappers	SHRiMP 2	1705:04	92111:02	45100	98.10 (99.47 99.56 98.86 / 88.49 77.61)	99.75 (1.58 52.34 81.82 / 86.14 88.60)
all-mappers	R-100	48:52	564:57	3532	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.53 (1.59 52.57 82.28 / 86.83 89.53)
all-mappers	R-99	21:46	241:16	3532	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.53 (1.59 52.57 82.28 / 86.83 89.53)
all-mappers	R-95	21:13	236:06	3532	100.00 (100.00 100.00 100.00 / 100.00 100.00)	89.53 (1.59 52.57 82.28 / 86.83 89.53)

dataset		simulated, $m = 800$ H. sapiens				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped reads [%]	mapped reads [%]
best-mappers	Bowtie 2	63:15	756:12	3686	96.83 (96.57 99.73 98.16 / 88.25 85.29)	100.00 (0.02 41.20 73.97 / 82.36 90.11)
best-mappers	BWA	26:00	257:32	10946	74.38 (96.57 97.72 82.00 / 0.00 0.00)	67.60 (0.02 40.37 67.60 / 67.60 67.60)
best-mappers	Soap 2	2:05	15:04	6122	41.51 (96.57 68.43 28.24 / 0.27 0.00)	38.22 (0.02 28.33 37.95 / 38.22 38.22)
best-mappers	R-100	102:48	1205:31	4964	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.49 (0.02 41.27 74.17 / 82.70 90.49)
best-mappers	R-99	38:50	442:12	4964	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.49 (0.02 41.27 74.17 / 82.70 90.49)
best-mappers	R-95	37:21	424:50	4964	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.49 (0.02 41.27 74.17 / 82.70 90.49)
all-mappers	mrFAST	1116:03	13750:58	9601	44.19 (62.66 64.69 40.39 / 0.00 0.00)	47.24 (0.01 26.70 47.24 / 47.24 47.24)
all-mappers	SHRiMP 2	–	–	–	–	–
all-mappers	R-100	105:48	1243:35	4964	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.49 (0.02 41.27 74.17 / 82.70 90.49)
all-mappers	R-99	41:41	477:03	4964	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.49 (0.02 41.27 74.17 / 82.70 90.49)
all-mappers	R-95	39:52	455:24	4964	100.00 (100.00 100.00 100.00 / 100.00 100.00)	90.49 (0.02 41.27 74.17 / 82.70 90.49)

Table A.4: *Extended experimental results for **long simulated single-end** data, extending Table 6.5a on page 129. The results are shown for 1 M single-end reads of the given lengths m simulated with Mason using the default (stretched) Illumina error model. Hobbes is not capable of mapping long reads and thus not shown here. Some mrFAST processes repeatedly crashed for the 400 bp fly dataset and the 400 and 800 bp human datasets which explains the low number of mapped reads. SHRiMP 2 was not able to map the 800 bp human dataset within 96 hours.*

	dataset	ERR022075 E. coli					SRR065390 C. elegans				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]
best-mappers	Bowtie 2	4:09	49:15	195	99.69 100.00 99.94 99.51 97.97 95.89	98.69 18.92 78.51 88.74 91.84 93.46	6:19	74:50	322	97.30 99.98 99.35 96.91 93.85 87.33	84.59 13.20 16.42 17.74 18.64 19.33
	BWA	7:35	30:58	343	99.66 100.00 99.94 99.48 97.54 95.39	95.87 18.92 78.51 88.74 91.82 93.44	12:32	75:47	479	92.01 99.97 95.50 82.94 67.12 56.05	81.20 13.56 16.88 18.21 19.00 19.56
	Soap 2	3:51	17:29	743	96.58 100.00 99.81 98.87 68.19 38.61	94.70 18.93 78.48 89.09 91.58 92.37	5:56	42:02	833	89.51 100.00 94.23 88.19 59.50 29.87	24.48 13.25 16.34 17.70 18.34 18.73
	R-100	1:53	7:33	11113	100.00 100.00 100.00 100.00 100.00 100.00	94.66 18.92 78.55 88.79 91.91 93.54	6:29	64:44	11230	100.00 100.00 100.00 100.00 100.00 100.00	20.38 13.20 16.43 17.78 18.70 19.43
	R-99	1:45	6:08	11113	99.99 100.00 100.00 100.00 99.98 99.79	94.66 18.92 78.55 88.79 91.91 93.54	6:20	62:39	11230	99.97 100.00 100.00 99.99 99.98 99.85	20.38 13.20 16.43 17.78 18.70 19.43
	R-95	1:44	6:22	11113	99.96 100.00 100.00 100.00 99.84 98.96	94.62 18.92 78.55 88.79 91.90 93.52	6:07	60:22	11230	99.81 100.00 100.00 99.99 99.83 99.21	20.35 13.20 16.43 17.78 18.70 19.43
all-mappers	Hobbes	5:21	39:44	1196	90.57 90.81 90.62 90.19 89.95 90.21	86.15 17.18 71.25 80.61 83.50 85.05	10:27	103:38	2852	86.49 92.91 85.94 72.90 65.00 62.82	18.66 12.26 15.09 16.14 16.82 17.41
	mrFAST	1:21	10:22	52868	99.99 99.99 99.99 99.99 99.99 99.99	95.31 18.92 78.55 88.78 91.90 93.54	14:02	134:13	54510	99.29 99.75 99.48 98.82 98.22 97.68	79.37 13.18 16.41 17.75 18.66 19.40
	SHRiMP 2	8:18	97:16	1612	99.85 100.00 99.99 99.80 99.06 97.80	97.74 18.92 78.54 88.76 91.85 93.46	105:30	1238:32	3468	94.81 99.77 96.89 88.75 80.41 73.48	53.92 13.32 16.58 17.88 18.72 19.36
	R-100	1:53	7:34	11113	100.00 100.00 100.00 100.00 100.00 100.00	94.66 18.92 78.55 88.79 91.91 93.54	7:49	78:35	11816	100.00 100.00 100.00 100.00 100.00 100.00	20.38 13.20 16.43 17.78 18.70 19.43
	R-99	1:48	6:40	11113	99.99 100.00 100.00 100.00 99.98 99.79	94.66 18.92 78.55 88.79 91.91 93.54	6:37	64:39	11588	99.97 100.00 100.00 99.99 99.98 99.85	20.38 13.20 16.43 17.78 18.70 19.43
	R-95	1:45	6:13	11113	99.96 100.00 100.00 100.00 99.84 98.96	94.62 18.92 78.55 88.79 91.90 93.52	6:33	63:35	11230	99.81 100.00 100.00 99.99 99.83 99.21	20.35 13.20 16.43 17.78 18.70 19.43

	dataset	SRR497711 D. melanogaster					ERR012100 H. sapiens				
	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]	mapped pairs [%]
best-mappers	Bowtie 2	6:32	77:40	349	98.94 100.00 99.03 98.43 97.52 96.11	81.94 32.50 50.89 60.48 66.19 69.88	10:51	128:58	3555	99.51 99.97 99.90 99.46 98.34 96.40	94.19 15.04 62.97 77.57 82.66 85.16
	BWA	13:33	77:14	503	97.47 100.00 98.85 97.77 93.52 87.15	73.41 32.51 50.87 60.41 65.93 69.30	34:35	241:41	4662	98.84 99.99 99.88 98.95 95.16 90.79	88.06 15.04 62.97 77.50 82.47 84.86
	Soap 2	5:29	38:22	940	88.67 100.00 94.93 89.40 70.79 41.11	72.77 32.58 50.33 59.65 64.13 65.93	8:24	61:32	5463	91.58 99.99 98.89 93.72 51.38 26.08	87.47 15.07 62.57 77.33 80.49 81.46
	R-100	9:01	93:39	11199	100.00 100.00 100.00 100.00 100.00 100.00	72.95 32.50 51.07 60.63 66.35 70.04	176:29	2077:35	18568	100.00 100.00 100.00 100.00 100.00 100.00	86.93 15.04 63.02 77.65 82.76 85.27
	R-99	7:00	69:26	11199	99.97 100.00 100.00 100.00 99.97 99.80	72.93 32.50 51.07 60.63 66.34 70.04	159:03	1872:33	16568	99.98 100.00 100.00 100.00 99.98 99.75	86.91 15.04 63.02 77.65 82.76 85.26
	R-95	6:56	68:48	11199	99.78 100.00 100.00 100.00 99.68 98.65	72.80 32.50 51.07 60.63 66.33 69.98	135:44	1599:10	13678	99.89 100.00 100.00 100.00 99.81 98.78	86.84 15.04 63.02 77.65 82.75 85.23
all-mappers	Hobbes	8:43	75:57	5870	84.78 84.27 85.76 86.51 85.66 83.24	62.48 27.39 43.41 51.81 56.81 59.99	89:35	884:05	47270	95.11 95.68 95.83 94.73 92.92 90.73	84.05 14.39 60.42 74.46 79.44 81.95
	mrFAST	8:26	81:25	48032	100.00 100.00 100.00 99.99 99.99 99.99	73.16 32.50 51.07 60.63 66.35 70.04	779:12	7649:19	57625	99.94 99.98 99.97 99.93 99.85 99.76	87.79 15.04 63.01 77.64 82.75 85.26
	SHRiMP 2	47:07	537:29	3838	99.67 100.00 99.99 99.84 99.21 97.77	87.36 32.50 51.07 60.62 66.30 69.95	2762:32	31710:26	38594	99.74 99.91 99.90 99.81 99.40 98.40	97.51 15.03 62.96 77.57 82.66 85.15
	R-100	7:59	76:28	13558	100.00 100.00 100.00 100.00 100.00 100.00	72.95 32.50 51.07 60.63 66.35 70.04	184:27	2167:14	27623	100.00 100.00 100.00 100.00 100.00 100.00	86.93 15.04 63.02 77.65 82.76 85.27
	R-99	7:44	73:54	13485	99.97 100.00 100.00 100.00 99.97 99.80	72.93 32.50 51.07 60.63 66.34 70.04	177:56	2100:43	25866	99.98 100.00 100.00 100.00 99.98 99.75	86.91 15.04 63.02 77.65 82.76 85.26
	R-95	7:36	72:22	13241	99.78 100.00 100.00 100.00 99.68 98.65	72.80 32.50 51.07 60.63 66.33 69.98	166:22	1956:20	23026	99.89 100.00 100.00 100.00 99.81 98.78	86.84 15.04 63.02 77.65 82.75 85.23

Table A.5: *Extended experimental results for real-world* ***paired-end*** *data, extending Table 6.5b on page 129.*

Simulated, $m = 200$, D. melanogaster

	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]		mapped pairs [%]	
best-mappers	Bowtie 2	1:25	16:37	390	98.91	100.00 99.87 99.59 / 98.98 96.98	99.24	1.35 17.49 45.66 / 64.52 71.86
	BWA	2:13	12:41	674	96.98	100.00 99.82 99.20 / 97.04 90.51	84.30	1.35 17.49 45.56 / 64.04 70.88
	Soap 2	1:55	16:35	1092	74.85	100.00 95.74 87.54 / 66.82 24.65	71.96	1.37 17.17 42.92 / 56.01 57.95
	R-100	1:26	12:21	3075	100.00	100.00 100.00 100.00 / 100.00 100.00	74.91	1.35 17.51 45.74 / 64.64 72.00
	R-99	1:25	12:44	3075	99.99	100.00 100.00 100.00 / 100.00 99.98	74.91	1.35 17.51 45.74 / 64.64 72.00
	R-95	1:23	12:42	3075	99.98	100.00 100.00 100.00 / 99.99 99.91	74.89	1.35 17.51 45.74 / 64.64 71.99
all-mappers	mrFAST	2:16	20:37	11104	98.46	99.97 99.97 99.97 / 99.63 95.11	84.68	1.35 17.51 45.73 / 64.57 71.57
	SHRiMP 2	13:10	133:22	3746	98.99	99.99 99.96 99.90 / 99.43 96.68	99.97	1.35 17.51 45.73 / 64.56 71.79
	R-100	1:33	14:02	3075	100.00	100.00 100.00 100.00 / 100.00 100.00	74.91	1.35 17.51 45.74 / 64.64 72.00
	R-99	1:30	13:23	3075	99.99	100.00 100.00 100.00 / 100.00 99.98	74.91	1.35 17.51 45.74 / 64.64 72.00
	R-95	1:26	12:32	3075	99.98	100.00 100.00 100.00 / 99.99 99.91	74.89	1.35 17.51 45.74 / 64.64 71.99

Simulated, $m = 400$, D. melanogaster

	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]		mapped pairs [%]	
best-mappers	Bowtie 2	5:45	68:17	515	97.37	100.00 99.79 99.45 / 98.32 92.65	99.62	0.02 6.16 35.88 / 57.96 65.03
	BWA	6:13	38:07	975	88.31	100.00 99.64 98.50 / 93.79 65.44	71.56	0.02 6.16 35.59 / 56.62 61.60
	Soap 2	4:40	44:16	1377	50.03	100.00 91.25 70.98 / 34.00 7.57	49.55	0.02 5.71 27.43 / 35.38 36.19
	R-100	1:45	13:44	5364	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.17 35.96 / 58.11 65.29
	R-99	1:48	14:30	5364	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.17 35.96 / 58.11 65.29
	R-95	1:41	13:31	5364	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.17 35.96 / 58.11 65.29
all-mappers	mrFAST	4:51	45:52	12493	88.68	100.00 99.97 99.79 / 95.80 60.61	72.13	0.02 6.17 35.90 / 57.24 62.37
	SHRiMP 2	83:57	979:35	3945	96.87	100.00 99.88 99.68 / 98.34 90.03	99.83	0.02 6.17 35.89 / 57.86 64.72
	R-100	1:54	15:36	5364	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.17 35.96 / 58.11 65.29
	R-99	1:53	14:53	5364	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.17 35.96 / 58.11 65.29
	R-95	1:51	15:15	5364	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.17 35.96 / 58.11 65.29

Simulated, $m = 800$, D. melanogaster

	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]		mapped pairs [%]	
best-mappers	Bowtie 2	39:07	468:30	1418	93.64	– 99.61 99.18 / 96.68 88.10	99.70	0.00 1.00 24.15 / 47.10 57.94
	BWA	11:26	73:40	1569	56.28	– 99.23 95.71 / 70.82 4.22	46.44	0.00 0.99 23.32 / 39.95 40.44
	Soap 2	12:36	124:23	1561	23.55	– 78.27 48.35 / 20.11 0.94	28.23	0.00 0.80 12.38 / 17.36 17.64
	R-100	2:22	16:05	9942	100.00	– 100.00 100.00 / 100.00 100.00	71.16	0.00 1.00 24.22 / 47.33 58.38
	R-99	2:19	15:24	9942	100.00	– 100.00 100.00 / 100.00 100.00	71.16	0.00 1.00 24.22 / 47.33 58.37
	R-95	2:19	15:27	9942	100.00	– 100.00 100.00 / 100.00 100.00	71.16	0.00 1.00 24.22 / 47.33 58.37
all-mappers	mrFAST	10:47	102:53	15392	44.19	– 95.01 91.19 / 40.34 0.00	49.69	0.00 0.95 24.50 / 43.35 43.35
	SHRiMP 2	1617:26	19264:14	4211	91.64	– 99.76 99.33 / 95.80 83.47	98.62	0.00 1.00 24.12 / 46.79 57.14
	R-100	2:30	15:55	9942	100.00	– 100.00 100.00 / 100.00 100.00	71.16	0.00 1.00 24.22 / 47.33 58.38
	R-99	2:30	15:45	9942	100.00	– 100.00 100.00 / 100.00 100.00	71.16	0.00 1.00 24.22 / 47.33 58.37
	R-95	2:29	15:47	9942	100.00	– 100.00 100.00 / 100.00 100.00	71.16	0.00 1.00 24.22 / 47.33 58.37

Simulated, $m = 200$, H. sapiens

	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]		mapped pairs [%]	
best-mappers	Bowtie 2	2:05	24:14	3439	98.75	99.99 99.86 99.54 / 98.72 96.62	99.29	1.40 17.45 45.61 / 64.42 71.75
	BWA	10:39	90:33	6790	96.88	100.00 99.84 99.16 / 96.97 90.29	84.24	1.40 17.46 45.52 / 64.00 70.86
	Soap 2	2:05	16:55	5667	74.84	100.00 95.93 87.57 / 66.83 24.57	71.89	1.41 17.14 42.86 / 55.95 57.90
	R-100	28:39	327:39	3759	100.00	100.00 100.00 100.00 / 100.00 100.00	74.91	1.40 17.47 45.69 / 64.59 71.97
	R-99	25:03	285:08	3759	99.99	100.00 100.00 100.00 / 100.00 99.98	74.91	1.40 17.47 45.69 / 64.59 71.97
	R-95	24:14	275:23	3759	99.98	100.00 100.00 100.00 / 99.99 99.92	74.90	1.40 17.47 45.69 / 64.59 71.97
all-mappers	mrFAST	217:29	2160:07	14319	98.44	99.96 99.97 99.96 / 99.64 95.05	84.77	1.40 17.47 45.69 / 64.52 71.55
	SHRiMP 2	996:11	10678:56	39094	98.82	99.82 99.83 99.75 / 99.27 96.62	99.98	1.40 17.46 45.64 / 64.45 71.70
	R-100	29:59	343:42	3759	100.00	100.00 100.00 100.00 / 100.00 100.00	74.91	1.40 17.47 45.69 / 64.59 71.97
	R-99	26:31	302:45	3759	99.99	100.00 100.00 100.00 / 100.00 99.98	74.91	1.40 17.47 45.69 / 64.59 71.97
	R-95	25:21	288:57	3759	99.98	100.00 100.00 100.00 / 99.99 99.92	74.90	1.40 17.47 45.69 / 64.59 71.97

Simulated, $m = 400$, H. sapiens

	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]		mapped pairs [%]	
best-mappers	Bowtie 2	7:43	90:09	3544	97.34	100.00 99.80 99.43 / 98.31 92.58	99.81	0.02 6.17 35.96 / 57.87 64.94
	BWA	38:15	378:52	11137	87.72	100.00 99.63 98.25 / 92.88 64.13	71.10	0.02 6.16 35.61 / 56.30 61.19
	Soap 2	4:46	43:57	6385	50.27	100.00 91.57 71.35 / 34.06 7.46	49.64	0.02 5.72 27.61 / 35.48 36.27
	R-100	34:07	390:25	5366	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.18 36.05 / 58.06 65.26
	R-99	27:30	312:19	5366	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.18 36.05 / 58.06 65.25
	R-95	28:03	319:08	5366	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.18 36.05 / 58.06 65.25
all-mappers	mrFAST	328:45	1991:56	13630	22.23	22.71 25.04 25.03 / 24.08 15.06	18.10	0.005 1.55 9.02 / 14.35 15.63
	SHRiMP 2	3243:49	187639:34	48973	96.64	100.00 99.60 99.43 / 98.08 90.01	99.86	0.02 6.16 35.90 / 57.66 64.52
	R-100	33:11	378:59	5366	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.18 36.05 / 58.06 65.26
	R-99	26:40	299:26	5366	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.18 36.05 / 58.06 65.25
	R-95	26:34	300:32	5366	100.00	100.00 100.00 100.00 / 100.00 100.00	69.68	0.02 6.18 36.05 / 58.06 65.25

Simulated, $m = 800$, H. sapiens

	method	time [min:s]	cpu time [min:s]	memory [Mb]	correctly mapped pairs [%]		mapped pairs [%]	
best-mappers	Bowtie 2	71:12	850:50	4380	93.84	– 99.81 99.28 / 96.87 88.23	99.90	0.00 0.96 24.23 / 47.11 57.94
	BWA	62:08	568:37	10955	55.46	– 98.70 94.13 / 69.75 4.11	45.78	0.00 0.95 23.00 / 39.33 39.81
	Soap 2	11:16	111:05	7385	23.83	– 79.51 49.05 / 20.12 0.91	28.27	0.00 0.78 12.53 / 17.48 17.76
	R-100	47:09	539:22	9944	100.00	– 100.00 100.00 / 100.00 100.00	71.11	0.00 0.96 24.29 / 47.33 58.39
	R-99	38:57	443:16	9944	100.00	– 100.00 100.00 / 100.00 100.00	71.11	0.00 0.96 24.29 / 47.33 58.39
	R-95	32:30	366:04	9944	100.00	– 100.00 100.00 / 100.00 100.00	71.11	0.00 0.96 24.29 / 47.33 58.39
all-mappers	mrFAST	1043:59	9427:14	18019	21.01	– 43.36 43.28 / 19.22 0.00	23.71	0.00 0.42 11.66 / 20.63 20.63
	SHRiMP 2	–	–	–	–		–	
	R-100	43:27	495:53	9944	100.00	– 100.00 100.00 / 100.00 100.00	71.11	0.00 0.96 24.29 / 47.33 58.39
	R-99	32:34	365:36	9944	100.00	– 100.00 100.00 / 100.00 100.00	71.11	0.00 0.96 24.29 / 47.33 58.39
	R-95	32:27	366:30	9944	100.00	– 100.00 100.00 / 100.00 100.00	71.11	0.00 0.96 24.29 / 47.33 58.39

Table A.6: *Extended experimental results for* ***long simulated paired-end*** *data, extending Table 6.5b on page 129. The results are shown for 2×1 M paired-end reads of the given lengths m simulated with Mason using the default (stretched) Illumina error model. Some mrFAST processes repeatedly crashed for the 400 bp fly dataset and the 400 and 800 bp human datasets which explains the low number of mapped pairs. SHRiMP 2 was not able to map the 800 bp human dataset within 96 hours.*

A.6 Proving hull optimality

In the following, we show that the monotonic hulls proposed in Examples 7.2, 7.3, and 7.4 (Section 7.2.2 on page 135) are optimal.

Proposition A.1. *Let $\mathcal{D}_1, \mathcal{D}_2$ be two databases, $\rho_s, \rho_g \in \mathbb{R}$, and pred* $: \mathbb{N}^2 \rightarrow \{true, false\}$ *be defined as:*

$$pred(d_1, d_2) = (d_1 \geq \rho_s \cdot |\mathcal{D}_1|) \wedge (d_1 \cdot |\mathcal{D}_2| \geq \rho_g \cdot d_2 \cdot |\mathcal{D}_1|). \tag{A.3}$$

The monotonic hull $pred_{hull}$ of pred with:

$$pred_{hull}(d_1, d_2) := (d_1 \geq \rho_s \cdot |\mathcal{D}_1|) \tag{A.4}$$

is optimal.

Proof. We assume $pred_{\text{hull}}$ is a non-optimal monotonic hull of *pred*. Then there exists a monotonic hull $pred'_{\text{hull}}$ of *pred* with $pred_{\text{hull}} \not\Rightarrow pred'_{\text{hull}}$. Thus, $d \in \mathbb{N}^2$ exist so that $pred_{\text{hull}}(d_1, d_2)$ is true and $pred'_{\text{hull}}(d_1, d_2)$ is false. By contraposition of the monotonicity criterion, $pred'_{\text{hull}}(d_1, 0)$ also is false. It holds that $pred(d_1, 0) = pred_{\text{hull}}(d_1, d_2) = \text{true}$ and $pred \not\Rightarrow pred'_{\text{hull}}$. This is a contradiction to $pred'_{\text{hull}}$ being a monotonic hull of *pred*. Hence the proposition holds. ■

Proposition A.2. *Let* $(\min_1, \ldots, \min_m), (\max_1, \ldots, \max_m) \in \mathbb{N}^m$, *with* $(\min_1, \ldots, \min_m) \leq (\max_1, \ldots, \max_m)$, *and pred* $: \mathbb{N}^m \rightarrow \{true, false\}$ *be defined as:*

$$pred(d) = (\min_1 \leq d_1 \leq \max_1) \wedge \ldots \wedge (\min_m \leq d_m \leq \max_m). \tag{A.5}$$

The monotonic hull $pred_{hull}$ of pred with:

$$pred_{hull}(d) := (\min_1 \leq d_1) \wedge \ldots \wedge (\min_m \leq d_m) \tag{A.6}$$

is optimal.

Proof. Analogously holds for a $pred'_{\text{hull}}$ and $d \in \mathbb{N}^m$: $pred_{\text{hull}}(d)$ is true and $pred'_{\text{hull}}(d)$ is false. Thus it holds that $(\min_1, \ldots, \min_m) \leq d$ and $pred'_{\text{hull}}(\min_1, \ldots, \min_m)$ also is false. It holds that $pred(\min_1, \ldots, \min_m) = \text{true}$ and $pred \not\Rightarrow pred'_{\text{hull}}$. This is a contradiction to $pred'_{\text{hull}}$ being a monotonic hull of *pred*. Hence the proposition holds. ■

Proposition A.3. *Let $\alpha, \rho_s \in \mathbb{R}$, $0 \leq \alpha, \rho_s \leq 1$, and pred* $: \mathbb{N}^m \rightarrow \{true, false\}$ *be defined as:*

$$pred(d) = \left(\forall_{i \in [1..m]} \frac{d_i}{|\mathcal{D}_i|} \geq \rho_s\right) \wedge \left(\sum_{i=1}^{m} \frac{d_i}{|\mathcal{D}_i| \cdot \omega(d)} \log_m \frac{d_i}{|\mathcal{D}_i| \cdot \omega(d)} \leq \alpha\right), \tag{A.7}$$

$$with \quad \omega(d) = \sum_{i=1}^{m} \frac{d_i}{|\mathcal{D}_i|}. \tag{A.8}$$

The monotonic hull $pred_{hull}$ of pred with:

$$pred_{hull}(d) := \left(\forall_{i \in [1..m]} \frac{d_i}{|\mathcal{D}_i|} \geq \rho_s\right) \tag{A.9}$$

is optimal.

Proof. Again assume non-optimality of $pred_{\text{hull}}$ and for a $pred'_{\text{hull}}$ and $d \in \mathbb{N}^m$: $pred_{\text{hull}}(d)$ is true and $pred'_{\text{hull}}(d)$ is false. Choose k such that $d_k/|\mathcal{D}_k| \geq \rho_s$ and $d' \in \mathbb{N}^m$ such that $d'_k = d_k$ and $d'_i = 0$ for $i \neq k$. Then $pred'_{\text{hull}}(d')$ is false because $pred'_{\text{hull}}$ is a *monotonic* hull. $pred(d')$ is true because $pred_{\text{hull}}(d')$ is true and the following holds:

$$\sum_{i=1}^{m} \frac{d'_i}{|\mathcal{D}_i| \cdot \omega(d')} \log_m \frac{d'_i}{|\mathcal{D}_i| \cdot \omega(d')} = \frac{d'_k}{|\mathcal{D}_k| \cdot \omega(d')} \log_m \frac{d'_k}{|\mathcal{D}_k| \cdot \omega(d')} \tag{A.10}$$

$$= \log_m 1 \tag{A.11}$$

$$= 0 \leq \alpha\,. \tag{A.12}$$

Thus $pred \not\Rightarrow pred'_{\text{hull}}$ follows which is a contradiction to $pred'_{\text{hull}}$ being a monotonic hull of *pred*. Hence the proposition holds. ■

Appendix B

Curriculum Vitae

For privacy reasons, the curriculum vitae is not contained in the online version of this thesis.

For privacy reasons, the curriculum vitae is not contained in the online version of this thesis.

For privacy reasons, the curriculum vitae is not contained in the online version of this thesis.

Appendix C Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

David Weese
June 5, 2013

BIBLIOGRAPHY

Abouelhoda, M., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, **2**, pages 53–86.

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2002a). The enhanced suffix array and its applications to genome analysis. In *Proc. of the 2nd Workshop on Algorithms in Bioinformatics*, volume 2452 of *LNCS*, pages 449–463. Springer.

Abouelhoda, M. I., Ohlebusch, E., and Kurtz, S. (2002b). Optimal exact string matching based on suffix arrays. In *Proc. of the 9th International Symposium on String Processing and Information Retrieval*, volume 2476 of *LNCS*, pages 31–43. Springer.

Adamidi, C., Wang, Y., Gruen, D., Mastrobuoni, G., You, X., Tolle, D., Dodt, M., Mackowiak, S. D., Gogol-Döring, A., Oenal, P., Rybak, A., Ross, E., Alvarado, A. S., Kempa, S., Dieterich, C., Rajewsky, N., and Chen, W. (2011). De novo assembly and validation of planaria transcriptome by massive parallel sequencing and shotgun proteomics. *Genome Res.*, **21**(7), pages 1193–1200.

Agrawal, R., Imielinski, T., and Swami, A. N. (1993). Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216. ACM Press.

Ahmadi, A., Behm, A., Honnalli, N., Li, C., Weng, L., and Xie, X. (2012). Hobbes: optimized gram-based methods for efficient read alignment. *Nucleic Acids Res.*, **40**(6), page e41.

Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: an aid to bibliographic search. *Commun. ACM*, **18**, pages 333–340.

Alkan, C., Kidd, J. M., Marques-Bonet, T., Aksay, G., Antonacci, F., Hormozdiari, F., Kitzman, J. O., Baker, C., Malig, M., Mutlu, O., Sahinalp, S. C., Gibbs, R. A., and Eichler, E. E. (2009). Personalized copy number and segmental duplication maps using next-generation sequencing. *Nat. Genet.*, **41**(10), pages 1061–1067.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *J. Mol. Biol.*, **215**(3), pages 403–410.

Asuncion, A. and Newman, D. (2007). UCI machine learning repository. `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

Baeza-Yates, R. A. and Navarro, G. (1999). Faster approximate string matching. *Algorithmica*, **23**(2), pages 127–158.

Baron, D. and Bresler, Y. (2005). Antisequential suffix sorting for BWT-based data compression. *IEEE T. Comput.*, **54**(4), pages 385–397.

Barski, A., Cuddapah, S., Cui, K., Roh, T., Schones, D., Wang, Z., Wei, G., Chepelev, I., and Zhao, K. (2007). High-resolution profiling of histone methylations in the human genome. *Cell*, **129**, pages 823–837.

Bentley, D. R., Balasubramanian, S., Swerdlow, H. P., Smith, G. P., Milton, J., Brown, C. G., Hall, K. P., Evers, D. J., Barnes, C. L., Bignell, H. R., Boutell, J. M., Bryant, J., Carter, R. J., Keira Cheetham, R., Cox, A. J., Ellis, D. J., Flatbush, M. R., Gormley, N. A., Humphray, S. J., Irving, L. J., Karbelashvili, M. S., Kirk, S. M., Li, H., Liu, X., Maisinger, K. S., Murray, L. J., Obradovic, B., Ost, T., Parkinson, M. L., Pratt, M. R., Rasolonjatovo, I. M. J., Reed, M. T., Rigatti, R., Rodighiero, C., Ross, M. T., Sabot, A., Sankar, S. V., Scally, A., Schroth, G. P., Smith, M. E., Smith, V. P., Spiridou, A., Torrance, P. E., Tzonev, S. S., Vermaas, E. H., Walter, K., Wu, X., Zhang, L., Alam, M. D., Anastasi, C., Aniebo, I. C., Bailey, D. M. D., Bancarz, I. R., Banerjee, S., Barbour, S. G., Baybayan, P. A., Benoit, V. A., Benson, K. F., Bevis, C., Black, P. J., Boodhun, A., Brennan, J. S., Bridgham, J. A., Brown, R. C., Brown, A. A., Buermann, D. H., Bundu, A. A., Burrows, J. C., Carter, N. P., Castillo, N., Chiara E. Catenazzi, M., Chang, S., Neil Cooley, R., Crake, N. R., Dada, O. O., Diakoumakos, K. D., Dominguez-Fernandez, B., Earnshaw, D. J., Egbujor, U. C., Elmore, D. W., Etchin, S. S., Ewan, M. R., Fedurco, M., Fraser, L. J., Fuentes Fajardo, K. V., Scott Furey, W., George, D., Gietzen, K. J., Goddard, C. P., Golda, G. S., Granieri, P. A., Green, D. E., Gustafson, D. L., Hansen, N. F., Harnish, K., Haudenschild, C. D., Heyer, N. I., Hims, M. M., Ho, J. T., Horgan, A. M., Hoschler, K., Hurwitz, S., Ivanov, D. V., Johnson, M. Q., James, T., Huw Jones, T. A., Kang, G.-D., Kerelska, T. H., Kersey, A. D., Khrebtukova, I., Kindwall, A. P., Kingsbury, Z., Kokko-Gonzales, P. I., Kumar, A., Laurent, M. A., Lawley, C. T., Lee, S. E., Lee, X., Liao, A. K., Loch, J. A., Lok, M., Luo, S., Mammen, R. M., Martin, J. W., McCauley, P. G., McNitt, P., Mehta, P., Moon, K. W., Mullens, J. W., Newington, T., Ning, Z., Ling Ng, B., Novo, S. M., O'Neill, M. J., Osborne, M. A., Osnowski, A., Ostadan, O., Paraschos, L. L., Pickering, L., Pike, A. C., Pike, A. C., Chris Pinkard, D., Pliskin, D. P., Podhasky, J., Quijano, V. J., Raczy, C., Rae, V. H., Rawlings, S. R., Chiva Rodriguez, A., Roe, P. M., Rogers, J., Rogert Bacigalupo, M. C., Romanov, N., Romieu, A., Roth, R. K., Rourke, N. J., Ruediger, S. T., Rusman, E., Sanches-Kuiper, R. M., Schenker, M. R., Seoane, J. M., Shaw, R. J., Shiver, M. K., Short, S. W., Sizto, N. L., Sluis, J. P., Smith, M. A., Ernest Sohna Sohna, J., Spence, E. J., Stevens, K., Sutton, N., Szajkowski, L., Tregidgo, C. L., Turcatti, G., Vandevondele, S., Verhovsky, Y., Virk, S. M., Wakelin, S., Walcott, G. C., Wang, J., Worsley, G. J., Yan, J., Yau, L., Zuerlein, M., Rogers, J., Mullikin, J. C., Hurles, M. E., McCooke, N. J., West, J. S., Oaks, F. L., Lundberg, P. L., Klenerman, D., Durbin, R., and Smith, A. J. (2008). Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, **456**(7218), pages 53–59.

Berghard, A. and Dryer, L. (1998). A novel family of ancient vertebrate odorant receptors. *J. Neurobiol.*, **37**(3), pages 383–392.

Berry, M. J. and Linoff, G. S. (1997). *Data Mining Techniques: For Marketing, Sales, and Customer Support.*, pages 51–62. John Wiley & Sons, 1 edition.

Birzele, F. and Kramer, S. (2006). A new representation for protein secondary structure prediction based on frequent patterns. *Bioinformatics*, **22**(21), pages 2628–2634.

Brāzma, A., Jonassen, I., Vilo, J., and Ukkonen, E. (1998). Predicting gene regulatory elements in silico on a genomic scale. *Genome Res.*, **8**(11), pages 1202–1215.

Burkhardt, S. and Kärkkäinen, J. (2002). One-gapped q-gram filters for levenshtein distance. In *Proc. of the 13th Annual Symposium on Combinatorial Pattern Matching*, CPM ’02, pages 225–234. Springer.

Burkhardt, S. and Kärkkäinen, J. (2003). Better filtering with gapped q-grams. *Fund. Inform.*, **56**(1,2), pages 51–70.

Burkhardt, S., Crauser, A., Ferragina, P., Lenhof, H.-P., Rivals, E., and Vingron, M. (1999). q-gram based database searching using a suffix array (QUASAR). In *Proc. of the 3rd Annual International Conference on Research in Computational Molecular Biology*, RECOMB ’99, pages 77–83. ACM Press.

Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. Technical Report 124, Digital SRC Research Report.

Chan, S., Kao, B., Yip, C. L., and Tang, M. (2003). Mining emerging substrings. In *Proc. of the 8th International Conference on Database Systems for Advanced Applications*, DASFAA ’03, pages 119–126. IEEE Computer Society.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Chen, W., Kalscheuer, V., Tzschach, A., Menzel, C., Ullmann, R., Schulz, M. H., Erdogan, F., Li, N., Kijas, Z., Arkesteijn, G., Pajares, I. L., Goetz-Sothmann, M., Heinrich, U., Rost, I., Dufke, A., Grasshoff, U., Glaeser, B., Vingron, M., and Ropers, H. H. (2008). Mapping translocation breakpoints by next-generation sequencing. *Genome Res.*, **18**, pages 1143–1149.

Ching, Y.-T., Mehlhorn, K., and Smid, M. H. M. (1990). Dynamic deferred data structuring. *Inf. Process. Lett.*, **35**(1), pages 37–40.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Cover, T. H. and Thomas, J. A. (1991). *Elements of Information Theory*. Wiley-Interscience.

Cox, A. J. (2006). Eland: efficient local alignment of nucleotide data. unpublished.

Damerau, F. (1964). A technique for computer detection and correction of spelling errors. *Commun. ACM*, **7**(3), pages 171–176.

David, M., Dzamba, M., Lister, D., Ilie, L., and Brudno, M. (2011). SHRiMP2: sensitive yet practical short read mapping. *Bioinformatics*, **27**(7), pages 1011–1012.

Dementiev, R. and Sanders, P. (2003). Asynchronous parallel disk sorting. In *Proc. of the 15th Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '03. ACM Press.

Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. (2005). Better external memory suffix array construction. In C. Demetrescu, R. Sedgewick, and R. Tamassia, editors, *ALENEX/ANALCO*, pages 86–97. SIAM.

Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. (2008a). Better external memory suffix array construction. *J. Exp. Algorithmics*, **12**, pages 3.4:1–3.4:24.

Dementiev, R., Kettner, L., and Sanders, P. (2008b). Stxxl: standard template library for xxl data sets. *Software Pract. Exper.*, **38**, pages 589–637.

Dohm, J. C., Lottaz, C., Borodina, T., and Himmelbauer, H. (2008). Substantial biases in ultra-short read data sets from high-throughput DNA sequencing. *Nucleic Acids Res.*, **36**(16), page e105.

Dong, G. and Li, J. (1999). Efficient mining of emerging patterns: discovering trends and differences. In *Proc. of the 5th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '99, pages 43–52. ACM Press.

Döring, A., Weese, D., Rausch, T., and Reinert, K. (2008). SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinf.*, **9**, page 11.

Dressman, D., Yan, H., Traverso, G., Kinzler, K. W., and Vogelstein, B. (2003). Transforming single DNA molecules into fluorescent magnetic particles for detection and enumeration of genetic variations. *Proc. of the National Academy of Sciences*, **100**(15), pages 8817–8822.

Emde, A.-K., Grunert, M., Weese, D., Reinert, K., and Sperling, S. R. (2010). MicroRazerS: rapid alignment of small RNA reads. *Bioinformatics*, **26**(1), pages 123–124.

Emde, A.-K., Schulz, M. H., Weese, D., Sun, R., Vingron, M., Kalscheuer, V. M., Haas, S. A., and Reinert, K. (2012). Detecting genomic indel variants with exact breakpoints in single- and paired-end sequencing data using splazers. *Bioinformatics*, **28**(5), pages 619–627.

Ewing, B. and Green, P. (1998). Base-calling of automated sequencer traces using Phred. *Genome Res.*, **8**(3), pages 186–194.

Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. (2004). An alphabet-friendly fm-index. In *Proc. of the 11th Symposium on String Processing and Information Retrieval (SPIRE)*, volume 3246 of *LNCS*, pages 150–160. Springer.

Fischer, J. and Heun, V. (2006). Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 4009 of *LNCS*, pages 36–48. Springer.

Fischer, J. and Heun, V. (2007). A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. of the 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE)*, volume 4614 of *LNCS*, pages 459–470. Springer.

Fischer, J., Heun, V., and Kramer, S. (2005). Fast frequent string mining using suffix arrays. In *Proc. of the 5th IEEE International Conference on Data Mining*, ICDM '05, pages 609–612. IEEE Computer Society.

Fischer, J., Heun, V., and Kramer, S. (2006). Optimal string mining under frequency constraints. In *Proc. of the 10th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, volume 4213 of *LNCS*, pages 139–150. Springer.

Fischer, J., Mäkinen, V., and Välimäki, N. (2008). Space efficient string mining under frequency constraints. In *Proc. of the 8th IEEE International Conference on Data Mining*, ICDM '08, pages 193–202. IEEE Computer Society.

Fitzgerald, P. C., Sturgill, D., Shyakhtenko, A., Oliver, B., and Vinson, C. (2006). Comparative genomics of drosophila and human core promoters. *Genome Biol.*, **7**, page R53.

Fonseca, N. A., Rung, J., Brazma, A., and Marioni, J. C. (2012). Tools for mapping high-throughput sequencing data. *Bioinformatics*. 10.1093/bioinformatics/bts605.

Giegerich, R. and Kurtz, S. (1995). A comparison of imperative and purely functional suffix tree constructions. *Sci. Comput. Program.*, **25**, pages 187–218.

Giegerich, R., Kurtz, S., and Stoye, J. (2003). Efficient implementation of lazy suffix trees. *Software Pract. Exper.*, **33**(11), pages 1035–1049.

Glenn, T. C. (2011). Field guide to next-generation DNA sequencers. *Mol. Ecol. Resour.*, **11**(5), pages 759–769.

Gog, S. and Ohlebusch, E. (2011). Fast and lightweight lcp-array construction algorithms. In M. Müller-Hannemann and R. F. F. Werneck, editors, *Proc. of the Workshop on Algorithm Engineering and Experiments*, ALENEX '11, pages 25–34. SIAM.

Göke, J., Schulz, M. H., Lasserre, J., and Vingron, M. (2012). Estimation of pairwise sequence similarity of mammalian enhancers with word neighbourhood counts. *Bioinformatics*, **28**(5), pages 656–663.

Gotoh, O. (1982). An improved algorithm for matching biological sequences. *J. Mol. Biol.*, **162**(3), pages 705–708.

Grossi, R., Gupta, A., and Vitter, J. S. (2003). High-order entropy-compressed text indexes. In *Proc. of the 14th annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: Computer science and computational biology*. Cambridge University Press, New York, NY, USA.

Haanpää, H. (2004). Minimum sum and difference covers of abelian groups. *J. Integer Sequences*, **7**, pages 1–10.

Han, J., Pei, J., Yin, Y., and Mao, R. (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Min. Knowl. Discov.*, **8**(1), pages 53–87.

Han, J., Cheng, H., Xin, D., and Yan, X. (2007). Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, **15**(1), pages 55–86.

Hauswedell, H. (2009). *BLAST-like Local Alignments with RazerS*. Bachelor's thesis, Freie Universität Berlin.

Herms, I. and Rahmann, S. (2008). Computing alignment seed sensitivity with probabilistic arithmetic automata. In K. Crandall and J. Lagergren, editors, *Algorithms in Bioinformatics*, volume 5251 of *LNCS*, pages 318–329. Springer.

Hillier, L. W., Marth, G. T., Quinlan, A. R., Dooling, D., Fewell, G., Barnett, D., Fox, P., Glasscock, J. I., Hickenbotham, M., Huang, W., Magrini, V. J., Richt, R. J., Sander, S. N., Stewart, D. A., Stromberg, M., Tsung, E. F., Wylie, T., Schedl, T., Wilson, R. K., and Mardis, E. R. (2008). Whole-genome sequencing and variant discovery in C. elegans. *Nat. Methods*, **5**(2), pages 183–188.

Hoffmann, S., Otto, C., Kurtz, S., Sharma, C. M., Khaitovich, P., Vogel, J., Stadler, P. F., and Hackermüller, J. (2009). Fast mapping of short sequences with mismatches, insertions and deletions using index structures. *PLoS Comput. Biol.*, **5**(9), page e1000502.

Holtgrewe, M. (2010). Mason – a read simulator for second generation sequencing data. Technical Report TR-B-10-06, Institut für Mathematik und Informatik, Freie Universität Berlin.

Holtgrewe, M., Emde, A.-K., Weese, D., and Reinert, K. (2011). A novel and well-defined benchmarking method for second generation read mapping. *BMC Bioinf.*, **12**, page 210.

Hu, M. and Liu, B. (2004). Mining opinion features in customer reviews. In *Proc. of the 19th National Conference on Artificial Intelligence*, AAAI '04, pages 755–760. AAAI Press.

Huson, D. H., Auch, A. F., Qi, J., and Schuster, S. C. (2007). MEGAN analysis of metagenomic data. *Genome Res.*, **17**(3), pages 377–386.

Hyyrö, H. (2003). A bit-vector algorithm for computing levenshtein and damerau edit distances. *Nordic J. of Computing*, **10**, pages 29–39.

Intel (2011). *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation.

Janitz, M. (2008). *Next-Generation Genome Sequencing: Towards Personalized Medicine*. John Wiley & Sons.

Jensen, M. and Pagh, R. (2008). Optimality in external memory hashing. *Algorithmica*, **52**, pages 403–411.

Jeon, J.-E., Park, H., and Kim, D.-K. (2005). Efficient construction of generalized suffix arrays by merging suffix arrays. *Comput. Syst. Theor.*, **32**(6), pages 268–278.

Ji, X., Bailey, J., and Dong, G. (2007). Mining minimal distinguishing subsequence patterns with gap constraints. *Knowl. Inf. Syst.*, **11**(3), pages 259–286.

Jiang, H. and Wong, W. H. (2008). Seqmap: mapping massive amount of oligonucleotides to the genome. *Bioinformatics*, **24**(20), pages 2395–2396.

Jokinen, P. and Ukkonen, E. (1991). Two algorithms for approximate string matching in static texts. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, volume 520 of *LNCS*, pages 240–248. Springer.

Kärkkäinen, J. and Sanders, P. (2003). Simple linear work suffix array construction. In *Automata, Languages and Programming*, volume 2719 of *LNCS*, pages 943–955. Springer.

Kärkkäinen, J., Sanders, P., and Burkhardt, S. (2006). Linear work suffix array construction. *J. ACM*, **53**, pages 918–936.

Kärkkäinen, J., Manzini, G., and Puglisi, S. J. (2009). Permuted longest-common-prefix array. In *Proc. of the 20th Annual Symposium on Combinatorial Pattern Matching*, CPM ’09, pages 181–192. Springer.

Karp, R. M., Motwani, R., and Raghavan, P. (1987). Deferred data structuring. Technical Report UCB/CSD-87-320, EECS Department, University of California, Berkeley.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. of the 12th Annual Symposium on Combinatorial Pattern Matching*, CPM ’01, pages 181–192. Springer.

Kehr, B., Weese, D., and Reinert, K. (2011). Stellar: fast and exact local alignments. *BMC Bioinf.*, **12**(Suppl 9), page S15.

Kent, W. (2002). Blat–the blast-like alignment tool. *Genome Res.*, **12**(4), pages 656–64.

Kim, D. K., Sim, J. S., Park, H., and Park, K. (2005). Constructing suffix arrays in linear time. *J. Discrete Algorithms*, **3**(2–4), pages 126–142.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd Edition)*. Addison-Wesley.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley.

Ko, P. and Aluru, S. (2005). Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, **3**(2–4), pages 143–156.

Kobyliński, Ł. and Walczak, K. (2009). Jumping emerging substrings in image classification. In X. Jiang and N. Petkov, editors, *Computer Analysis of Images and Patterns*, volume 5702 of *LNCS*, pages 732–739. Springer.

Kodama, Y., Shumway, M., and Leinonen, R. (2012). The sequence read archive: explosive growth of sequencing data. *Nucleic Acids Res.*, **40**(D1), pages D54–D56.

Kügel, A. and Ohlebusch, E. (2008). A space efficient solution to the frequent string mining problem for many databases. *Data Min. Knowl. Discov.*, **17**(1), pages 24–38.

Kurtz, S. (1999). Reducing the space requirement of suffix trees. *Software Pract. Exper.*, **29**(13), pages 1149–1171.

Lam, T. W., Sung, W. K., Tam, S. L., Wong, C. K., and Yiu, S. M. (2008). Compressed indexing and local alignment of DNA. *Bioinformatics*, **24**(6), pages 791–797.

Lander, E. S., Linton, L. M., Birren, B., Nusbaum, C., Zody, M. C., Baldwin, J., Devon, K., Dewar, K., Doyle, M., FitzHugh, W., Funke, R., Gage, D., Harris, K., Heaford, A., Howland, J., Kann, L., Lehoczky, J., LeVine, R., McEwan, P., McKernan, K., Meldrim, J., Mesirov, J. P., Miranda, C., Morris, W., Naylor, J., Raymond, C., Rosetti, M., Santos, R., Sheridan, A., Sougnez, C., Stange-Thomann, N., Stojanovic, N., Subramanian, A., Wyman, D., Rogers, J., Sulston, J., Ainscough, R., Beck, S., Bentley, D., Burton, J., Clee, C., Carter, N., Coulson, A., Deadman, R., Deloukas, P., Dunham, A., Dunham, I., Durbin, R., French, L., Grafham, D., Gregory, S., Hubbard, T., Humphray, S., Hunt, A., Jones, M., Lloyd, C., McMurray, A., Matthews, L., Mercer, S., Milne, S., Mullikin, J. C., Mungall, A., Plumb, R., Ross, M., Shownkeen, R., Sims, S., Waterston, R. H., Wilson, R. K., Hillier, L. W., McPherson, J. D., Marra, M. A., Mardis, E. R., Fulton, L. A., Chinwalla, A. T., Pepin, K. H., Gish, W. R., Chissoe, S. L., Wendl, M. C., Delehaunty, K. D., Miner, T. L., Delehaunty, A., Kramer, J. B., Cook, L. L., Fulton, R. S., Johnson, D. L., Minx, P. J., Clifton, S. W., Hawkins, T., Branscomb, E., Predki, P., Richardson, P., Wenning, S., Slezak, T., Doggett, N., Cheng, J. F., Olsen, A., Lucas, S., Elkin, C., Uberbacher, E., Frazier, M., Gibbs, R. A., Muzny, D. M., Scherer, S. E., Bouck, J. B., Sodergren, E. J., Worley, K. C., Rives, C. M., Gorrell, J. H., Metzker, M. L., Naylor, S. L., Kucherlapati, R. S., Nelson, D. L., Weinstock, G. M., Sakaki, Y., Fujiyama, A., Hattori, M., Yada, T., Toyoda, A., Itoh, T., Kawagoe, C., Watanabe, H., Totoki, Y., Taylor, T., Weissenbach, J., Heilig, R., Saurin, W., Artiguenave, F., Brottier, P., Bruls, T., Pelletier, E., Robert, C., Wincker, P., Smith, D. R., Doucette-Stamm, L., Rubenfield, M., Weinstock, K., Lee, H. M., Dubois, J., Rosenthal, A., Platzer, M., Nyakatura, G., Taudien, S., Rump, A., Yang, H., Yu, J., Wang, J., Huang, G., Gu, J., Hood, L., Rowen, L., Madan, A., Qin, S., Davis, R. W., Federspiel, N. A., Abola, A. P., Proctor, M. J., Myers, R. M., Schmutz, J., Dickson, M., Grimwood, J., Cox, D. R., Olson, M. V., Kaul, R., Raymond, C., Shimizu, N., Kawasaki, K., Minoshima, S., Evans, G. A., Athanasiou, M., Schultz, R., Roe, B. A., Chen, F., Pan, H., Ramser, J., Lehrach, H., Reinhardt, R., McCombie, W. R., de la Bastide, M., Dedhia, N.,

Blöcker, H., Hornischer, K., Nordsiek, G., Agarwala, R., Aravind, L., Bailey, J. A., Bateman, A., Batzoglou, S., Birney, E., Bork, P., Brown, D. G., Burge, C. B., Cerutti, L., Chen, H. C., Church, D., Clamp, M., Copley, R. R., Doerks, T., Eddy, S. R., Eichler, E. E., Furey, T. S., Galagan, J., Gilbert, J. G., Harmon, C., Hayashizaki, Y., Haussler, D., Hermjakob, H., Hokamp, K., Jang, W., Johnson, L. S., Jones, T. A., Kasif, S., Kaspryzk, A., Kennedy, S., Kent, W. J., Kitts, P., Koonin, E. V., Korf, I., Kulp, D., Lancet, D., Lowe, T. M., McLysaght, A., Mikkelsen, T., Moran, J. V., Mulder, N., Pollara, V. J., Ponting, C. P., Schuler, G., Schultz, J., Slater, G., Smit, A. F., Stupka, E., Szustakowski, J., Thierry-Mieg, D., Thierry-Mieg, J., Wagner, L., Wallis, J., Wheeler, R., Williams, A., Wolf, Y. I., Wolfe, K. H., Yang, S. P., Yeh, R. F., Collins, F., Guyer, M. S., Peterson, J., Felsenfeld, A., Wetterstrand, K. A., Patrinos, A., Morgan, M. J., de Jong, P., Catanese, J. J., Osoegawa, K., Shizuya, H., Choi, S., Chen, Y. J., and Szustakowki, J. (2001). Initial sequencing and analysis of the human genome. *Nature*, **409**(6822), pages 860–921.

Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with Bowtie 2. *Nat. Methods*, **9**(4), pages 357–359.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. L. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, **10**(3), page R25.

Lee, S. D. and Raedt, L. D. (2005). An efficient algorithm for mining string databases under constraints. In B. Goethals and A. Siebes, editors, *Knowledge Discovery in Inductive Databases (KDID '04)*, volume 3377 of *LNCS*, pages 108–129. Springer.

Leinonen, R., Akhtar, R., Birney, E., Bower, L., Cerdeno-Tárraga, A., Cheng, Y., Cleland, I., Faruque, N., Goodgame, N., Gibson, R., Hoad, G., Jang, M., Pakseresht, N., Plaister, S., Radhakrishnan, R., Reddy, K., Sobhany, S., Ten Hoopen, P., Vaughan, R., Zalunin, V., and Cochrane, G. (2011). The European nucleotide archive. *Nucleic Acids Res.*, **39**(suppl 1), pages D28–D31.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics – Doklady*, **10**, pages 707–710.

Ley, T. J., Mardis, E. R., Ding, L., Fulton, B., McLellan, M. D., Chen, K., Dooling, D., Dunford-Shore, B. H., McGrath, S., Hickenbotham, M., Cook, L., Abbott, R., Larson, D. E., Koboldt, D. C., Pohl, C., Smith, S., Hawkins, A., Abbott, S., Locke, D., Hillier, L. W., Miner, T., Fulton, L., Magrini, V., Wylie, T., Glasscock, J., Conyers, J., Sander, N., Shi, X., Osborne, J. R., Minx, P., Gordon, D., Chinwalla, A., Zhao, Y., Ries, R. E., Payton, J. E., Westervelt, P., Tomasson, M. H., Watson, M., Baty, J., Ivanovich, J., Heath, S., Shannon, W. D., Nagarajan, R., Walter, M. J., Link, D. C., Graubert, T. A., DiPersio, J. F., and Wilson, R. K. (2008). DNA sequencing of a cytogenetically normal acute myeloid leukaemia genome. *Nature*, **456**(7218), pages 66–72.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, **25**(14), pages 1754–1760.

Li, H. and Homer, N. (2010). A survey of sequence alignment algorithms for next-generation sequencing. *Brief. Bioinform.*, **11**(5), pages 473–483.

Li, H., Ruan, J., and Durbin, R. (2008a). Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Res.*, **18**(11), pages 1851–1858.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., and 1000 Genome Project Data Processing Subgroup (2009a). The sequence alignment/map format and SAMtools. *Bioinformatics*, **25**(16), pages 2078–2079.

Li, M., Ma, B., Kisman, D., and Tromp, J. (2003). Patternhunter II: highly sensitive and fast homology search. *Genome Inform.*, **14**, pages 164–175.

Li, R., Li, Y., Kristiansen, K., and Wang, J. (2008b). Soap: short oligonucleotide alignment program. *Bioinformatics*, **24**(5), pages 713–714.

Li, R., Yu, C., Li, Y., Lam, T.-W., Yiu, S.-M., Kristiansen, K., and Wang, J. (2009b). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, **25**(15), pages 1966–1967.

Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., and Wang, J. (2010). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, **20**(2), pages 265–272.

Lin, H., Zhang, Z., Zhang, M. Q., Ma, B., and Li, M. (2008). Zoom! zillions of oligos mapped. *Bioinformatics*, **24**(21), pages 2431–2437.

Ma, B., Tromp, J., and Li, M. (2002). PatternHunter: faster and more sensitive homology search. *Bioinformatics*, **18**(3), pages 440–445.

Mäkinen, V. and Navarro, G. (2008). Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms*, **4**(3), pages 32:1–32:38.

Manber, U. and Myers, E. (1993). Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, **22**(5), pages 935–948.

Manzini, G. (2004). Two space saving tricks for linear time lcp array computation. In T. Hagerup and J. Katajainen, editors, *Algorithm Theory – SWAT '04*, volume 3111 of *LNCS*, pages 372–383. Springer.

Manzini, G. and Ferragina, P. (2004). Engineering a lightweight suffix array construction algorithm. *Algorithmica*, **40**, pages 33–50.

Mardis, E. R. (2008). Next-generation DNA sequencing methods. *Annu. Rev. Genomics Hum. Genet.*, **9**(1), pages 387–402.

Marschall, T., Martin, M., and Rahmann, S. (2009). A BWT-based suffix array construction. In *Biological Sequence Analysis Using the SeqAn C++ Library*, pages 261–282. CRC Press.

McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *J. ACM*, **23**(2), pages 262–272.

Meissner, A., Mikkelsen, T. S., Gu, H., Wernig, M., Hanna, J., Sivachenko, A., Zhang, X., Bernstein, B. E., Nusbaum, C., Jaffe, D. B., Gnirke, A., Jaenisch, R., and Lander, E. S. (2008). Genome-scale DNA methylation maps of pluripotent and differentiated cells. *Nature*, **454**(7205), pages 766–770.

Mitašiūnaitė, I., Rigotti, C., Schicklin, S., Meynie, L., Boulicaut, J.-F., and Gandrillon, O. (2008). Extracting signature motifs from promoter sets of differentially expressed genes. *In Silico Biol.*, **8**(0043), pages 17–39.

Montgomery, S., Sammeth, M., Gutierrez-Arcelus, M., Lach, R., Ingle, C., Nisbett, J., Guigo, R., and Dermitzakis, E. (2010). Transcriptome genetics using second generation sequencing in a Caucasian population. *Nature*, **464**(7289), pages 773–777.

Morris, E. R. and Walker, J. C. (2003). Receptor-like protein kinases: the keys to response. *Curr. Opin. Plant Biol.*, **6**(4), pages 339–342.

Morrison, D. R. (1968). Patricia – practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, **15**(4), pages 514–534.

Mortazavi, A., Williams, B., McCue, K., Schaeffer, L., and Wold, B. (2008). Mapping and quantifying mammalian transcriptomes by RNA-seq. *Nat. Methods*, **5**(7), pages 621–628.

Myers, E. W. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, **46**(3), pages 395–415.

Nakamura, K., Oshima, T., Morimoto, T., Ikeda, S., Yoshikawa, H., Shiwa, Y., Ishikawa, S., Linak, M. C., Hirai, A., Takahashi, H., Altaf-Ul-Amin, M., Ogasawara, N., and Kanaya, S. (2011). Sequence-specific error profile of Illumina sequencers. *Nucleic Acids Res.*, **39**(13), page e90.

Navarro, G. and Baeza-Yates, R. (2000). A hybrid indexing method for approximate string matching. *J. Discrete Algorithms*, **1**(1), pages 205–239.

Navarro, G. and Mäkinen, V. (2007). Compressed full-text indexes. *ACM Comput. Surv.*, **39**(1), pages 2:1–2:61.

Navarro, G. and Raffinot, M. (2002). *Flexible Pattern Matching in Strings*, chapter 6.5, pages 162–166. Cambridge University Press.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, pages 443–453.

Ning, Z., Cox, A. J., and Mullikin, J. C. (2001). SSAHA: A fast search method for large DNA databases. *Genome Res.*, **11**(10), pages 1725–1729.

Owolabi, O. and McGregor, D. R. (1988). Fast approximate string matching. *Software Pract. Exper.*, **18**(4), pages 387–393.

Puglisi, S. J., Smyth, W. F., and Turpin, A. H. (2007). A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, **39**, pages 1–31.

Raedt, L. D., Jaeger, M., Lee, S. D., and Mannila, H. (2002). A theory of inductive query answering. In *Proc. of the 2nd IEEE International Conference on Data Mining*, ICDM '02, pages 123–130. IEEE Computer Society.

Rasmussen, K. R., Stoye, J., and Myers, E. W. (2006). Efficient q-gram filters for finding all ε-matches over a given length. *J. Comput. Biol.*, **13**(2), pages 296–308.

Rausch, T., Emde, A.-K., Weese, D., Döring, A., Notredame, C., and Reinert, K. (2008). Segment-based multiple sequence alignment. *Bioinformatics*, **24**(16), pages i187–192.

Rausch, T., Koren, S., Denisov, G., Weese, D., Emde, A.-K., Döring, A., and Reinert, K. (2009). A consistency-based consensus algorithm for de novo and reference-guided sequence assembly of short reads. *Bioinformatics*, **25**(9), pages 1118–1124.

Redhead, E. and Bailey, T. L. (2007). Discriminative motif discovery in DNA and protein sequences using the DEME algorithm. *BMC Bioinf.*, **8**, page 385.

Richard, H., Schulz, M. H., Sultan, M., Nürnberger, A., Schrinner, S., Balzereit, D., Dagand, E., Rasche, A., Lehrach, H., Vingron, M., Haas, S. A., and Yaspo, M.-L. (2010). Prediction of alternative isoforms from exon expression levels in RNA-seq experiments. *Nucleic Acids Res.*, **38**(10), page e112.

Roberts, A., Pimentel, H., Trapnell, C., and Pachter, L. (2011). Identification of novel transcripts in annotated genomes using RNA-seq. *Bioinformatics*, **27**(17), pages 2325–2329.

Robertson, G., Schein, J., Chiu, R., Corbett, R., Field, M., Jackman, S. D., Mungall, K., Lee, S., Okada, H. M., Qian, J. Q., Griffith, M., Raymond, A., Thiessen, N., Cezard, T., Butterfield, Y. S., Newsome, R., Chan, S. K., She, R., Varhol, R., Kamoh, B., Prabhu, A.-L., Tam, A., Zhao, Y., Moore, R. A., Hirst, M., Marra, M. A., Jones, S. J. M., Hoodless, P. A., and Birol, I. (2010). De novo assembly and analysis of RNA-seq data. *Nat. Methods*, **7**(11), pages 909–912.

Rodrigue, S., Materna, A. C., Timberlake, S. C., Blackburn, M. C., Malmstrom, R. R., Alm, E. J., and Chisholm, S. W. (2010). Unlocking short read sequencing for metagenomics. *PLoS ONE*, **5**(7), page e11840.

Ronaghi, M., Karamohamed, S., Pettersson, B., Uhlén, M., and Nyrén, P. (1996). Real-time DNA sequencing using detection of pyrophosphate release. *Anal. Biochem.*, **242**(1), pages 84–89.

Rumble, S. M., Lacroute, P., Dalca, A. V., Fiume, M., Sidow, A., and Brudno, M. (2009). SHRiMP: Accurate mapping of short color-space reads. *PLoS Comput. Biol.*, **5**(5), page e1000386.

Sadakane, K. (2003). New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, **48**, pages 294–313.

Sadakane, K. (2007). Compressed suffix trees with full functionality. *Theor. Comput. Syst.*, **41**(4), pages 589–607.

Salson, M., Lecroq, T., Léonard, M., and Mouchard, L. (2009). A four-stage algorithm for updating a burrows–wheeler transform. *Theoret. Comput. Sci.*, **410**(43), pages 4350–4359.

Salson, M., Lecroq, T., Léonard, M., and Mouchard, L. (2010). Dynamic extended suffix arrays. *J. Discrete Algorithms*, **8**(2), pages 241–257.

Sanger, F., Nicklen, S., and Coulson, A. R. (1977). DNA sequencing with chain-terminating inhibitors. *PNAS*, **74**(12), pages 5463–5467.

Schmidt, D., Wilson, M. D., Ballester, B., Schwalie, P. C., Brown, G. D., Marshall, A., Kutter, C., Watt, S., Martinez-Jimenez, C. P., Mackay, S., Talianidis, I., Flicek, P., and Odom, D. T. (2010). Five-vertebrate ChIP-seq reveals the evolutionary dynamics of transcription factor binding. *Science*, **328**(5981), pages 1036–1040.

Schulz, M. H., Weese, D., Rausch, T., Döring, A., Reinert, K., and Vingron, M. (2008a). Fast and adaptive variable order markov chain construction. In K. Crandall and J. Lagergren, editors, *Algorithms in Bioinformatics*, volume 5251 of *LNCS*, pages 306–317. Springer.

Schulz, M. H., Bauer, S., and Robinson, P. N. (2008b). The generalised k-truncated suffix tree for time- and space-efficient searches in multiple DNA or protein sequences. *Int. J. Bioinform. Res. Appl.*, **4**(1), pages 81–95.

Schulz, M. H., Zerbino, D. R., Vingron, M., and Birney, E. (2012). Oases: robust de novo RNA-seq assembly across the dynamic range of expression levels. *Bioinformatics*, **28**(8), pages 1086–1092.

Sellers, P. H. (1980). The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, **1**(4), pages 359–373.

Shendure, J. and Ji, H. (2008). Next-generation DNA sequencing. *Nat. Biotechnol.*, **26**(10), pages 1135–1145.

Shi, F. (1996). Suffix arrays for multiple strings: A method for on-line multiple string searches. In J. Jaffar and R. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, volume 1179 of *LNCS*, pages 11–22. Springer.

Simpson, J. T. and Durbin, R. (2012). Efficient de novo assembly of large genomes using compressed data structures. *Genome Res.*, **22**(3), pages 549–556.

Siragusa, E., Weese, D., and Reinert, K. (2013a). Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic Acids Res.*, **41**(7), page e78.

Siragusa, E., Weese, D., and Reinert, K. (2013b). Scalable string similarity search/join with approximate seeds and multiple backtracking. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 370–374. ACM.

Slonim, N., Bejerano, G., Fine, S., and Tishby, N. (2003). Discriminative feature selection via multiclass variable memory Markov models. *EURASIP J. Applied Signal Processing*, **2**, pages 93–102.

Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, pages 195–197.

Stein, L. (2010). The case for cloud computing in genome informatics. *Genome Biol.*, **11**(5), page 207.

Stöber, G., Nöthen, M. M., Pörzgen, P., Brüss, M., Bönisch, H., Knapp, M., Beckmann, H., and Propping, P. (1996). Systematic search for variation in the human norepinephrine transporter gene: Identification of five naturally occurring missense mutations and study of association with major psychiatric disorders. *Am. J. Med. Genet.*, **67**(6), pages 523–532.

Trapnell, C., Williams, B., Pertea, G., Mortazavi, A., Kwan, G., Van Baren, M., Salzberg, S., Wold, B., and Pachter, L. (2010). Transcript assembly and quantification by RNA-seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nat. Biotechnol.*, **28**(5), pages 511–515.

Ukkonen, E. (1985). Finding approximate patterns in strings. *J. Algorithms*, **6**(1), pages 132–137.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, **14**(3), pages 249–260.

UniProt Consortium (2008). The universal protein resource (UniProt). *Nucleic Acids Res.*, **36**(suppl_1), pages D190–195. `ftp://ftp.ebi.ac.uk/pub/databases/integr8/uniprot/proteomes`.

Venter, J. C., Adams, M. D., Myers, E. W., Li, P. W., Mural, R. J., Sutton, G. G., Smith, H. O., Yandell, M., Evans, C. A., Holt, R. A., Gocayne, J. D., Amanatides, P., Ballew, R. M., Huson, D. H., Wortman, J. R., Zhang, Q., Kodira, C. D., Zheng, X. H., Chen, L., Skupski, M., Subramanian, G., Thomas, P. D., Zhang, J., Gabor Miklos, G. L., Nelson, C., Broder, S., Clark, A. G., Nadeau, J., McKusick, V. A., Zinder, N., Levine, A. J., Roberts, R. J., Simon, M., Slayman, C., Hunkapiller, M., Bolanos, R., Delcher, A., Dew, I., Fasulo, D., Flanigan, M., Florea, L., Halpern, A., Hannenhalli, S., Kravitz, S., Levy, S., Mobarry, C., Reinert, K., Remington, K., Abu-Threideh, J., Beasley, E., Biddick, K., Bonazzi, V., Brandon, R., Cargill, M., Chandramouliswaran, I., Charlab, R., Chaturvedi, K., Deng, Z., Di Francesco, V., Dunn, P., Eilbeck, K., Evangelista, C., Gabrielian, A. E., Gan, W., Ge, W., Gong, F., Gu, Z., Guan, P., Heiman, T. J., Higgins, M. E., Ji, R. R., Ke, Z., Ketchum, K. A., Lai, Z., Lei, Y., Li, Z., Li, J., Liang, Y., Lin, X., Lu, F., Merkulov, G. V., Milshina, N., Moore, H. M., Naik, A. K., Narayan,

V. A., Neelam, B., Nusskern, D., Rusch, D. B., Salzberg, S., Shao, W., Shue, B., Sun, J., Wang, Z., Wang, A., Wang, X., Wang, J., Wei, M., Wides, R., Xiao, C., Yan, C., Yao, A., Ye, J., Zhan, M., Zhang, W., Zhang, H., Zhao, Q., Zheng, L., Zhong, F., Zhong, W., Zhu, S., Zhao, S., Gilbert, D., Baumhueter, S., Spier, G., Carter, C., Cravchik, A., Woodage, T., Ali, F., An, H., Awe, A., Baldwin, D., Baden, H., Barnstead, M., Barrow, I., Beeson, K., Busam, D., Carver, A., Center, A., Cheng, M. L., Curry, L., Danaher, S., Davenport, L., Desilets, R., Dietz, S., Dodson, K., Doup, L., Ferriera, S., Garg, N., Gluecksmann, A., Hart, B., Haynes, J., Haynes, C., Heiner, C., Hladun, S., Hostin, D., Houck, J., Howland, T., Ibegwam, C., Johnson, J., Kalush, F., Kline, L., Koduru, S., Love, A., Mann, F., May, D., McCawley, S., McIntosh, T., McMullen, I., Moy, M., Moy, L., Murphy, B., Nelson, K., Pfannkoch, C., Pratts, E., Puri, V., Qureshi, H., Reardon, M., Rodriguez, R., Rogers, Y. H., Romblad, D., Ruhfel, B., Scott, R., Sitter, C., Smallwood, M., Stewart, E., Strong, R., Suh, E., Thomas, R., Tint, N. N., Tse, S., Vech, C., Wang, G., Wetter, J., Williams, S., Williams, M., Windsor, S., Winn-Deen, E., Wolfe, K., Zaveri, J., Zaveri, K., Abril, J. F., Guigó, R., Campbell, M. J., Sjolander, K. V., Karlak, B., Kejariwal, A., Mi, H., Lazareva, B., Hatton, T., Narechania, A., Diemer, K., Muruganujan, A., Guo, N., Sato, S., Bafna, V., Istrail, S., Lippert, R., Schwartz, R., Walenz, B., Yooseph, S., Allen, D., Basu, A., Baxendale, J., Blick, L., Caminha, M., Carnes-Stine, J., Caulk, P., Chiang, Y. H., Coyne, M., Dahlke, C., Mays, A., Dombroski, M., Donnelly, M., Ely, D., Esparham, S., Fosler, C., Gire, H., Glanowski, S., Glasser, K., Glodek, A., Gorokhov, M., Graham, K., Gropman, B., Harris, M., Heil, J., Henderson, S., Hoover, J., Jennings, D., Jordan, C., Jordan, J., Kasha, J., Kagan, L., Kraft, C., Levitsky, A., Lewis, M., Liu, X., Lopez, J., Ma, D., Majoros, W., McDaniel, J., Murphy, S., Newman, M., Nguyen, T., Nguyen, N., Nodell, M., Pan, S., Peck, J., Peterson, M., Rowe, W., Sanders, R., Scott, J., Simpson, M., Smith, T., Sprague, A., Stockwell, T., Turner, R., Venter, E., Wang, M., Wen, M., Wu, D., Wu, M., Xia, A., Zandieh, A., and Zhu, X. (2001). The sequence of the human genome. *Science*, **291**, pages 1304–1351.

Vyverman, M., De Baets, B., Fack, V., and Dawyndt, P. (2012). Prospects and limitations of full-text index structures in genome analysis. *Nucleic Acids Res.*, **40**(15), pages 6993–7015.

Walker, J. C. (1994). Structure and function of the receptor-like protein kinases of higher plants. *Plant Mol. Biol.*, **26**, pages 1599–1609.

Wang, J., Wang, W., Li, R., Li, Y., Tian, G., Goodman, L., Fan, W., Zhang, J., Li, J., Zhang, J., Guo, Y., Feng, B., Li, H., Lu, Y., Fang, X., Liang, H., Du, Z., Li, D., Zhao, Y., Hu, Y., Yang, Z., Zheng, H., Hellmann, I., Inouye, M., Pool, J., Yi, X., Zhao, J., Duan, J., Zhou, Y., Qin, J., Ma, L., Li, G., Yang, Z., Zhang, G., Yang, B., Yu, C., Liang, F., Li, W., Li, S., Li, D., Ni, P., Ruan, J., Li, Q., Zhu, H., Liu, D., Lu, Z., Li, N., Guo, G., Zhang, J., Ye, J., Fang, L., Hao, Q., Chen, Q., Liang, Y., Su, Y., san, A., Ping, C., Yang, S., Chen, F., Li, L., Zhou, K., Zheng, H., Ren, Y., Yang, L., Gao, Y., Yang, G., Li, Z., Feng, X., Kristiansen, K., Wong, G. K.-S., Nielsen, R., Durbin, R., Bolund, L., zhang, X., Li, S., Yang, H., and Wang, J. (2008). The diploid genome sequence of an Asian individual. *Nature*, **456**(7218), pages 60–65.

Weese, D. (2006). *Entwurf und Implementierung eines generischen Substring-Index*. Diploma thesis, Humboldt University Berlin.

Weese, D. and Schulz, M. H. (2008). Efficient string mining under constraints via the deferred frequency index. In *Proc. of the 8th Industrial Conference on Data Mining (ICDM '08)*, volume 5077 of *LNAI*, pages 374–388. Springer.

Weese, D., Emde, A.-K., Rausch, T., Döring, A., and Reinert, K. (2009). RazerS – fast read mapping with sensitivity control. *Genome Res.*, **19**(9), pages 1646–1654.

Weese, D., Holtgrewe, M., and Reinert, K. (2012). RazerS 3: faster, fully sensitive read mapping. *Bioinformatics*, **28**(20), pages 2592–2599.

Weese, D., Schulz, M. H., Holtgrewe, M., and Richard, H. (2013). Fiona: a versatile and automatic strategy for read error correction. *to appear*.

Weiner, P. (1973). Linear pattern matching algorithms. In *Proc. of the 14th Symposium on Switching and Automata Theory*, SWAT '73, pages 1–11. IEEE Computer Society.

Xie, S. Y., Feinstein, P., and Mombaerts, P. (2000). Characterization of a cluster comprising 100 odorant receptor genes in mouse. *Mamm. Genome*, **11**(12), pages 1070–1078.

Zhang, X. and Firestein, S. (2002). The olfactory receptor gene superfamily of the mouse. *Nat. Neurosci.*, **5**(12), pages 124–133.

Zhang, Z., Berman, P., and Miller, W. (1998). Alignments without low-scoring regions. *J. Comput. Biol.*, **5**(2), pages 197–210.

LIST OF FIGURES

1.1 DNA double helix . . . 4
1.2 A multiple read alignment . . . 11

2.1 Suffix tree examples . . . 16
2.2 A transcript between two sequences . . . 17
2.3 Global pairwise sequence alignments . . . 20

3.1 Suffix array and lcp table example . . . 27
3.2 Suffix tree, lcp-interval tree, linked ℓ-indices, and child table . . . 28
3.3 Skew step 1: Sample and sort triples . . . 32
3.4 Skew step 2: Extend sorting to remaining triples . . . 32
3.5 Skew step 3: Merge sorted lists of suffixes . . . 34
3.6 Construction peak memory usage . . . 51
3.7 Binary search on the suffix array . . . 53
3.8 Binary search with mlr-heuristic . . . 54
3.9 Comparison of ESA based search algorithms . . . 54
3.10 Suffix tree iterators . . . 59
3.11 Repeat examples . . . 60

4.1 Different states of the lazy suffix tree . . . 67
4.2 Original lazy suffix tree data structure . . . 69
4.3 Our lazy suffix tree data structure . . . 72
4.4 Approximate pattern search times . . . 80

5.1 q-gram index example . . . 84
5.2 q-gram index construction time and memory consumption . . . 89
5.3 q-gram lemma worst cases . . . 92
5.4 Filters based on q-gram counting . . . 93

6.1 Match extraction in RazerS . . . 100
6.2 q-gram counting in parallelograms . . . 103
6.3 Filters used in RazerS . . . 104
6.4 Typical Illumina error profile . . . 105
6.5 Examples for (q, Δ)-seed filters . . . 109
6.6 Parameters of the pigeonhole filter . . . 110
6.7 Applications of the banded alignment algorithm . . . 113
6.8 Clipped band parameters, steps of the algorithm, and initialization . . . 114

6.9 Average verification time per read character 116
6.10 Paired-end reads 118
6.11 Parallelization in RazerS 120
6.12 Mapping time ratios between the SWIFT and pigeonhole filter 122
6.13 Validation of the SWIFT filter sensitivity estimation 123
6.14 Validation of the pigeonhole filter sensitivity estimation 124

7.1 Generalized suffix tree example with string frequencies 141
7.2 Running time comparisons 143

A.1 SOLiD color labels of the 16 dinucleotides 154

LIST OF TABLES

1.1 Comparison of high-throughput sequencing technologies 5

3.1 Difference covers 35
3.2 Shift values used in SKEW3 and SKEW7 37
3.3 Enhanced suffix array construction algorithms available in SeqAn 49
3.4 Datasets used for ESA experiments 50
3.5 Construction times for ASCII datasets 50
3.6 Construction times for DNA datasets 51
3.7 Construction times and peak memory usage for large DNA datasets 52

6.1 Comparison of read mapping tools 98
6.2 Datasets used for the experimental maps 121
6.3 Rabema benchmark results 126
6.4 Variation detection results 128
6.5 Performance results 129

7.1 Existing frequency string mining algorithms and their characteristics . . 132
7.2 Examples for frequency, support, growth, and entropy 133
7.3 Databases used in our experiments 144
7.4 Emerging substring mining results 145
7.5 Frequent pattern mining results 146
7.6 Entropy substring mining results 146
7.7 Entropy substring mining on four proteomes 147

A.1 Full variation detection results for single-end reads 158
A.2 Full variation detection results for paired-end reads 158
A.3 Extended experimental results for real-world single-end data. 159
A.4 Extended experimental results for long simulated single-end data. 160
A.5 Extended experimental results for real-world paired-end data. 161
A.6 Extended experimental results for long simulated paired-end data. 162

LIST OF ALGORITHMS

3.1 skew3(s) . . . 37
3.2 skew7(s) . . . 38
3.3 skew7_ExtMem(s) . . . 40
3.4 skew7_Multi($s^1, \dots, s^m$) . . . 42
3.5 constructLcpTable(s, suftab) . . . 43
3.6 constructLcpTable_Inplace(s, suftab) . . . 44
3.7 constructLcpTable_ExtMem(s, suftab) . . . 46
3.8 constructLcpTable_Multi($s^1, \dots, s^m$, suftab) . . . 47
3.9 bottomUpTraversal(lcp) . . . 48
3.10 constructChildTable(lcp) . . . 49
3.11 findLower(s, p) . . . 53
3.12 findUpper(s, p) . . . 53
3.13 findLowerH(s, p) . . . 54
3.14 findUpperH(s, p) . . . 54
3.15 goRoot($iter$) . . . 55
3.16 isNextl(i) . . . 55
3.17 goDown($iter$) . . . 55
3.18 goRight($iter$) . . . 55
3.19 goNext_Pre(it) . . . 56
3.20 goNext_Post(it) . . . 56
3.21 repLength($iter$) . . . 57
3.22 representative($iter$) . . . 58
3.23 parentEdgeLabel($iter$) . . . 58
3.24 getOccurrences($iter$) . . . 58

4.1 wotdEager($T, \overline{\alpha}$) . . . 66
4.2 createSuffixTree(s) . . . 67
4.3 goRoot($iter$) . . . 75
4.4 updateRb($iter$) . . . 75
4.5 goDown($iter$) . . . 75
4.6 goRight($iter$) . . . 75
4.7 repLength($iter$) . . . 75
4.8 exactMultiRecursion($iterA, iterB, i$) . . . 77
4.9 approximateRecursion($pattern, iter, i, e$) . . . 79
4.10 approximatePatternSearch($pattern, errors$) . . . 79
4.11 approximateMultiRecursion($iterA, iterB, i, e$) . . . 81

4.12 approximateMultiPatternSearch($patterns, errors$) 81

5.1 constructQGramIndex(s, Q) 85
5.2 constructQGramIndex_Multi($s^1, \ldots, s^m, Q$) 86
5.3 constructQGramIndex_ExtMem(s, Q) 87
5.4 GetBkt(c, C) 88
5.5 ReqBkt(c, C) 88
5.6 getOccurrences(t, pos, dir) 90

6.1 $\delta(T)$ for gapped shapes 108
6.2 bandedMyers(t, p, k, w, c) 115
6.3 bandedMyers_LargeAlphabet(t, p, k, w, c) 117

7.1 FHK($\mathcal{D}_1, \ldots, \mathcal{D}_m, pred$) 139
7.2 DFI($T, \overline{\alpha}, pred, pred_{\text{hull}}$) 140
7.3 sortAndCountFreq($\overline{\alpha}$) 142

LIST OF NOTATIONS

Notation	Description	Page
bp	base pair, character of the alphabet {`A`,`C`,`G`,`T`}	3
Mb	megabase, 1 million base pairs or characters	50
MB	megabyte, 1 MB = 1024 kB = 1,048,576 byte	51
GB	gigabyte, 1 GB = 1024 MB	52
Σ, Ψ, Φ	finite alphabets	13
Σ^*	set of all possible strings over the alphabet Σ	13
Σ^n	set of all possible strings over the alphabet Σ with length n	13
ϵ	empty string	13
$\lvert s \rvert$	length of string s	13
$s[i]$	character of s at position i (counting from 0)	13
$s_i, \mathrm{suf}(s, i)$	suffix of s beginning at position i	13
$[i..j]$	set of integers $i, i+1, \ldots, j$	13
$[i..j)$	set of integers $i, i+1, \ldots, j-1$	13
$\mathbb{N}_0$	set of non-negative integers	13
$\prec, \preceq$	strict and non-strict substring relation	13
$\langle \ldots \mid \ldots \rangle$	definition of a string analogous to the set notation	14
$\mathrm{lcp}\, \mathcal{S}$	longest common prefix of a set $\mathcal{S}$ of strings	14
$<_{\mathrm{lex}}$	lexicographical order	14
$<_q$	lexicographical prefix order, compares only prefixes of length q	14
$\$, \$^j$	(virtual) sentinel characters to well-define the suffix tree	15
$concat(v)$	edge label concatenation on the path from root to suffix tree node v	16
$\overline{s}$	suffix tree node whose edge label concatenation is s	16
$\mathrm{rank}(a)$	rank of character a in the underlying alphabet	83
$\mathtt{R}, \mathtt{D}, \mathtt{I}$	edit operations that replace, delete, or insert a character	17
$\lVert T \rVert_{\mathrm{E}}$	number of edit operations in transcript T	17
$\lVert T \rVert_{\mathrm{R}}$	number of matches, replacements, and deletions in transcript T	107
$\mathcal{R}$	sequenced reads, set of strings	101
G	reference sequence, string	101
$\mathcal{D}$	database, set of strings	132
$\mathrm{freq}(\phi, \mathcal{D})$	absolute number of strings in $\mathcal{D}$ that contain ϕ at least once	133
$\mathrm{supp}(\phi, \mathcal{D})$	relative number of strings in $\mathcal{D}$ that contain ϕ at least once	133

INDEX

alignment, **17**, 80, 90, 93, 99, 101, 111
all-mapper, **99**, 119, 126, 128
alphabet, **13**, 62, 115
    integer alphabet, 30
array, 13

backward search, 99
best-mapper, **99**, 119, 125, 129
Burrows-Wheeler transform, 99

candidate region, **90**, 101–103, 111
Cartesian order, 14
character, 13
compressed index, 29
concatenation string, **16**, 56, 57, 66, 75

database, 132
deferred data structure, 65
difference cover, 34
    minimal, 34
    perfect, 34
DNA, 3

edit
    distance, **17**, 91, 111
    operations, 17
emerging substring, 134
    mining problem, 131, **134**, 135, 145, 146
entropy, 134
    substring mining problem, 131, **135**, 145, 146

FM index, 99
frequency, 132
    predicate, 133
    vector, 133
frequent pattern mining problem, **133**, 136, 138, 142–146, 163

generalized
    repeated pair, 61
    suffix array, 40
    suffix tree, **16**, 71, 136, 139, 141
growth rate, 133, **134**

Hamming distance, **17**, 90, 91, 111
hull, *see* monotonic hull

jumping emerging substring, 134

k-error
    match, **19**, 98, 108
    problem, 19
k-mismatch, 19
    problem, **19**, 78

ℓ-indices, 29
ℓ-interval, 26
lcp table, 25
lcp values, 26
lcp-interval, 26
    tree, 29
lexicographical
    naming, 15
    order, 14
longest common prefix, 14

maximal repeat, *see* repeat
maximal unique match, 59
minimal coverage, 91
mlr-heuristic, 50
monotonic hull, **136**, 137, 163
monotonic predicate, 135
MUM, *see* maximal unique match

ω-interval, 26

partial suffix tree, 66
pigeonhole filter, **102**, 108

prefix trie, 99
protein, 4

q-gram, 13, **83**
  code, 83
  index, 84
    construction, **85**, 86
    search, 90
  lemma, **90**, 101
  shape, 83
q-hit, **93**, 101
q-match, 101
QUASAR, 91

radix tree, 76
read mapping, 101
repeat
  maximal, 59
  supermaximal, 59
repeated pair, 59
  generalized, 61
  left maximal, 59
  maximal, 59
  right maximal, 59
representative, *see* concatenation string
RNA, 4
RNA-seq, 4

self-index, 29
SeqAn, 7
shape, 83
skew algorithm, 30
string, 13
succinct index, 29
suffix, 13
suffix array, 25
  enhanced, 25
  generalized, 40
  inverse, 25
suffix tree, 16
  generalized, **16**, 71, 136, 139, 141
  lazy, 65
suffix trie, **16**, 131
supermaximal repeat, *see* repeat
support, 132
SWIFT, **93**, 102, 105

table, 13
top-down iterator, 54
transcript, 17
tuple, 13

unique match, 63