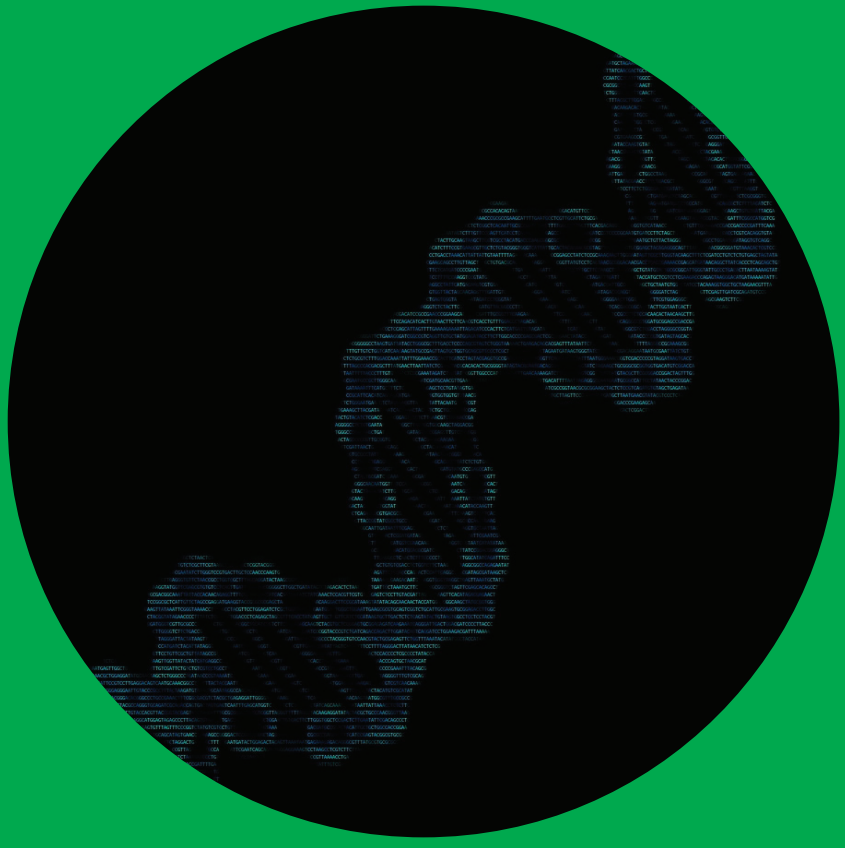
Department of Computer Science and Engineering

String Searching Methods for Bioinformatics

Kalle Karhu



Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 130/2013

String Searching Methods for Bioinformatics

Kalle Karhu

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 (Konemiehentie 2, Espoo) of the school on 20th September 2013 at 12 noon.

Aalto University
School of Science
Department of Computer Science and Engineering
String Algorithms Group

Supervising professor
Professor Jorma Tarhio

Thesis advisor
Professor Jorma Tarhio

Preliminary examiners
Professor Thierry Lecroq, University of Rouen, France
Dr Marie-France Sagot, Université Claude Bernard, France

Opponent
Professor Esko Ukkonen, University of Helsinki, Finland

Aalto University publication series
DOCTORAL DISSERTATIONS 130/2013

ISBN 978-952-60-5298-4 (printed)
ISBN 978-952-60-5299-1 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-5299-1

Unigrafia Oy
Helsinki 2013

Finland

441 697
Printed matter

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Kalle Karhu

Name of the doctoral dissertation
String Searching Methods for Bioinformatics

Publisher School of Science

Unit Department of Computer Science and Engineering

Series Aalto University publication series DOCTORAL DISSERTATIONS 130/2013

Field of research Software Technology

Manuscript submitted 12 April 2013 **Date of the defence** 20 September 2013

Permission to publish granted (date) 20 June 2013 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

The cost of obtaining biologically relevant data via sequencing has been declining rapidly, far surpassing the decline in computing costs. This is highlighting a need for more efficient, and thus cheaper, ways to analyze all of this data. Analyzing such data commonly requires searching through the text representing it in one way or another. The focus of this thesis is on improving the efficiency of the computational approaches that one may wish to use when searching through such texts. More precisely, it addresses three subproblems related to text searches in bioinformatics.

First, we consider the approximate, indexed alignment of long sequences. We present an approach using an index that combines q-sampling and block addressing for the initial approximate location of promising alignments, which are then studied more carefully using a multi-pattern, q-gram algorithm. Based on our experimental results, this approach is able to answer alignment queries notably faster than previous approaches, using only a fraction of the memory required by them. We additionally show that the quality of alignments and even the exon mappings produced by this approach are not worse than those produced using previous approaches.

Second, we consider indexed multi-pattern matching. For this subproblem, a set of multiple patterns is preprocessed, speeding up our search of this set from an index structure. This thesis presents the first experimental results on this type of an indexed, multi-pattern matching setting together with new theoretical insights. Practical approaches to this setting are presented, and our experimental results suggest that the presented approaches to preprocessing notably improve later searches from the corresponding index structures. Namely, compressed suffix arrays and bidirectional FM-indexes are considered in our study.

Finally, we consider protein motif discovery. We present a new graph-theoretical approach based on de Bruijn graphs. Moreover, we show how to further improve the query times of this approach using similarity indexing. Our experiments suggest that the presented approaches produce motif predictions of equal quality notably faster than previous methods.

Keywords sequence alignment, indexed multi-pattern matching, motif discovery

ISBN (printed) 978-952-60-5298-4 **ISBN (pdf)** 978-952-60-5299-1

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Helsinki **Location of printing** Espoo **Year** 2013

Pages 132 **urn** http://urn.fi/URN:ISBN:978-952-60-5299-1

A? Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Kalle Karhu

Väitöskirjan nimi
Merkkijonohaun Menetelmät Bioinformatiikassa

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 130/2013

Tutkimusala Ohjelmistotekniikka

Käsikirjoituksen pvm 12.04.2013 **Väitöspäivä** 20.09.2013

Julkaisuluvan myöntämispäivä 20.06.2013 **Kieli** Englanti

☐ **Monografia** ☒ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Biologiselta kannalta merkityksellisen datan tuottamisen kustannukset laskevat ennätyksellistä tahtia sekvensointiteknologian kehityksen myötä. Näiden kustannusten laskun nopeus ohittaa jopa laskentakustannusten laskun nopeuden. Tästä aiheutuu kasvava kysyntä, joka kohdistuu uusiin, tehokkaampiin laskennallisiin menetelmiin, joilla pystyttäisiin vastaamaan kasvavien datamäärien asettamiin haasteisiin. Tyypillisesti tällaisen datan analysointiin kuuluvat tekstihaut, muodossa tai toisessa. Tämä väitöskirja pureutuu sellaisten laskennallisten menetelmien tehokkuuden parantamiseen, joita tarvitaan, kun tällaisia tekstihakuja halutaan suorittaa. Tarkemmin, keskitymme kolmeen bioinformatiikan tekstihakujen osaongelmaan.

Ensimmäisenä tarkastelemme pitkien sekvenssien indeksoitua, likimääräistä hakua. Esitämme menetelmän, joka käyttää indeksirakenteita, jossa kaksi konseptia: q-sampling ja block addressing yhdistetään. Indeksirakenteen avulla löydetyt lupaavat alueet tarkistetaan usealle q-grammille suunnitellulla algoritmilla. Kokeelliset tuloksemme osoittavat, että tämä menetelmä vaatii vain murto-osan aikaisempien menetelmien vaatimasta muistista, mutta se on kuitenkin merkittävästi aikaisempia menetelmiä nopeampi.

Toiseksi, tarkastelemme usean hahmon indeksoitua hakua. Tässä osaongelmassa usean hahmon joukko esikäsitellään, tarkoituksena nopeuttaa tämän joukon myöhempää indeksoitua hakua. Tässä väitöskirjassa esitämme ensimmäiset tähän osaongelmaan liittyvät kokeelliset tulokset. Esitämme myös uusia teoreettisia huomioita tähän asetelmaan liittyen. Kokeelliset tuloksemme antavat viitteitä siitä, että esitetyt esikäsittelymenetelmät nopeuttavat hahmojoukkojen indeksoitua hakua huomattavasti. Keskitymme kahteen indeksirakenteeseen: tiivistettyyn loppuosataulukkoon ja kaksisuuntaiseen FM-indeksiin.

Viimeisenä osaongelmana keskitymme motifien etsimiseen proteiinisekvensseistä. Esittelemme graafiteoriaan pohjautuvan lähestymistavan, jossa käytämme de Bruijn -graafeja. Näytämme myös, kuinka tätä lähestymistapaa voidaan edelleen nopeuttaa samankaltaisuus-indeksointia apuna käyttäen. Kokeelliset tuloksemme osoittavat, että kehitetyt menetelmät ovat tarkkuudeltaan samaa tasoa, mutta merkittävästi nopeampia kuin aikaisemmat menetelmät.

Avainsanat sekvenssien rinnastus, usean hahmon indeksoitu haku, motifien tunnistus

ISBN (painettu) 978-952-60-5298-4 **ISBN (pdf)** 978-952-60-5299-1

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Helsinki **Painopaikka** Espoo **Vuosi** 2013

Sivumäärä 132 **urn** http://urn.fi/URN:ISBN:978-952-60-5299-1

Dedicated to the loving memory of my mother, Sinikka Karhu.

Preface

First, I would like to thank my supervisor, Jorma Tarhio, for always having time and good guidance for his students. I also want to thank Heikki Saikkonen, head of our department, for the wonderful facilities I have had at the Department of Computer Science and Engineering in Aalto University. I have had the privilege to work and co-author with a number of people, including Juho Mäkinen, Jussi Rautio, Hugh Salamon, Simon Gog, Juha Kärkkäinen, Veli Mäkinen, Niko Välimäki, Elena Czeizler, Tommi Hirvola, Gonzalo Navarro, Travis Gagie, Simon Puglisi, Jouni Sirén, Leena Salmela, Sami Khuri and Hannu Peltola. I wish to thank you all. I wish to express my gratitude towards my pre-examiners Marie-France Sagot and Thierry Lecroq for their feedback and helpful comments. I would also like to thank anonymous referees for their helpful notes along the way. I want to thank Tommi Suvitaival, Juuso Parkkinen, Seppo Virtanen, Eemeli Leppäaho and Jussi Gillberg for increasing my innovation potential nearly daily through our coffee break tournaments. I thank the Academy of Finland and the Helsinki Doctoral Programme in Computer Science for their funding and travel stipends. I am most thankful to my parents for their support throughout my life, making me pursue what I enjoy. Finally, I want to thank my wife Hanna for still bearing with me, and our kids Okko and Inna for making me smile every day.

Espoo, August 26, 2013,

Kalle Karhu

Contents

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Kalle Karhu, Juho Mäkinen, Jussi Rautio, Hugh Salamon and Jorma Tarhio. GAST, a genomic alignment search tool. In *BIOINFORMATICS 2011 - Proceedings of the International Conference on Bioinformatics Models, Methods and Algorithms*, 82–90, 2011.

II Kalle Karhu. Improving exact search of multiple patterns from a compressed suffix array. In *Proceedings of the Prague Stringology Conference*, 226–231, 2011.

III Simon Gog, Kalle Karhu, Juha Kärkkäinen, Veli Mäkinen and Niko Välimäki. Multi-Pattern Matching with Bidirectional Indexes. Accepted for publication in *Journal of Discrete Algorithms*, 2013.

IV Elena Czeizler, Tommi Hirvola and Kalle Karhu. A graph-theoretical approach for motif discovery in protein sequences. Submitted to *BMC Bioinformatics*, 2013.

Author's Contribution

Publication I: "GAST, a genomic alignment search tool"

The author had a notable role in designing and implementing the approaches presented in the paper. The major part of the writing was done by the author. The experiments were performed by the author.

Publication II: "Improving exact search of multiple patterns from a compressed suffix array"

This paper represents independent research conducted by the author.

Publication III: "Multi-Pattern Matching with Bidirectional Indexes"

This paper resulted from discussions following a presentation of [PII]. The author had a notable role in taking part in designing the presented methods. The author designed and implemented the practical variation presented in the paper. The author performed the experiments presented in the paper. The author wrote the major parts of the paper.

Publication IV: "A graph-theoretical approach for motif discovery in protein sequences"

A substantial part of the writing was done by the author, including the results, conclusions and parts of the methods section. A major part of the implementation tasks, parts of the initial testing and all of the final experiments were performed by the author. The author had a notable role

in designing the details of the approach, building on the core idea first presented by Dr Czeizler.

List of Abbreviations

AC-probe	12-mer starting with nucleotides adenine and cytosine
BG algorithm	BNDM with q-Grams algorithm
BLAST	Basic local alignment search tool
BLAT	BLAST-like alignment tool
BWT	Burrows-Wheeler transform
cDNA	Complementary deoxyribonucleic acid
CSA	Compressed suffix array
CST	Compressed suffix tree
DB	De Bruijn approach
DB-SS	De Bruijn approach using SS-tree-like similarity indexing
DNA	Deoxyribonucleic acid
GAST	Genomic alignment search tool
LZ77	Lempel-Ziv '77 algorithm
NCBI	National center for biotechnology information
NP-hard	Non-deterministic polynomial-time hard
RAM	Random access memory
RNA	Ribonucleic acid
SA	Suffix array
SDSL	Succinct data structure library
SS-tree	Similarity search tree
ST	Suffix tree

1. Introduction

1.1 Motivation

Since the beginning of 2008, the decreasing costs of DNA sequencing have far exceeded Moore's law, while the speed of producing biological data is still drastically increasing [78]. This means that, since the cost of obtaining biologically relevant data is declining notably faster than the cost of computing power, algorithmic means for analyzing the data efficiently are of increasing importance, possibly more than ever before.

Once one has obtained a biologically meaningful sequence of some sort, it is common to find out what kind of sequence one is dealing with. This may involve mapping the sequence onto other known sequences, trying to find other shorter sequences in this new sequence or trying to find repetitive patterns within the sequence or possibly within a set of such sequences. These tasks are the kinds of search problems focused on in this thesis.

Answering such questions will eventually help in gathering information about the biological sample that was initially sequenced. It may also answer questions about other sequences, samples and genomes that have been sequenced previously. The end results may provide numerous improvements in genetics and medical sciences. However, the focus of this thesis is not practical in that sense. The focus is on improving the computational approaches that one may wish to use when conducting searches on biologically meaningful data. We will review some theoretical improvements that were introduced in the publications associated with the thesis. Moreover, we describe ways to implement the proposed approaches and provide experimental results while comparing the presented approaches to typical, top-of-the-line approaches one would use

when conducting these kinds of searches. While these computational approaches are proposed for analyzing biological sequences, our approach has a strong flavor of theoretical computer science to it. In theoretical computer science, sequences of letters or symbols are commonly referred to as *strings*, while the term *sequences* is used more often in bioinformatics.

In order to have new, practical and efficient searching tools emerging, basic research on approaches of new kind is first needed. Let us consider the practical example of a relatively recent improvement in relevant search tools in bioinformatics, the `bowtie` [40], which was released in 2009. In terms of citation count, the bowtie can be considered the alignment tool that has had the greatest impact on bioinformatics within the last five years. It is a fast mapping tool that allows for some variation between the pattern sequence being searched and the portion of another text sequence that the pattern is to be mapped onto. In bioinformatics, such variation is typically of critical importance for finding the closest relevant matches for the pattern sequences being searched. This search tool is based on the concept of FM-index [11], which was first introduced in 2000. It is noteworthy that the FM-index itself does not allow for such variation, i.e. it only allows for exact matching. Without the FM-index, though, the bowtie would most likely have never been developed. The FM-index in turn is based on the concept of the Burrows-Wheeler transform [8] introduced in 1994. Thus, there was a gap of 9–15 years, going from the initial data structures, applicable only for mapping a pattern sequence exactly onto an existing text sequence, to an efficient, practical tool that was capable of performing more biologically relevant searches. This example demonstrates the importance of basic research and even of exact matching for bioinformatics.

1.2 Objectives and Scope

An enormous amount of effort is currently being focused on the field of bioinformatics, meaning that the precise focus of individual studies is essential to any realistic possibility for making meaningful improvements in the field. This goes for search problems in bioinformatics as well. Thus, this thesis does not focus on the entire field of searching for biologically meaningful patterns within a biologically meaningful text. Instead, three research questions are formulated, which in turn focus on three specific

problems.

First, we focus on biologically relevant searches for long patterns. In this setting, we focus on approximate matching, while allowing for some variation between the pattern sequence and the text sequence. More precisely, the focus is on sequence alignment and mapping. Alignment and mapping tools are commonly used when analyzing newly sequenced data. However, in the case of sequences longer than 200 nucleotides, we argue that the common methodologies are computationally excessive. Thus, we formulate our first research question (RQ1) as follows: *can we improve the efficiency of alignment and mapping methods for long patterns?*

Second, we focus on relevant searches for short patterns. In this setting, we have identified a gap in the previous research in terms of searching for a preprocessed set of multiple patterns simultaneously in a preprocessed text. The focus is mostly on sequences of up to 200 nucleotides (or characters) in length. With these shorter sequences, focusing on exact matching is arguably more relevant than with longer patterns. As we briefly discussed in Section 1.1, basic research in exact matching can be considered the foundation for moving towards more bioinformatically relevant approximate matching. This will be discussed more thoroughly in Chapter 7. Thus, we formulate our second research question (RQ2) as follows: *can we speed up exact indexed multi-pattern matching?*

Lastly, we consider a case involving multiple sequences, some of which share a common biological feature and for which it would be desirable to know more about what is possibly causing this particular biological feature. In this setting, the challenge is essentially to find biologically relevant shorter sequences in a set of longer sequences, without knowing exactly what these short sequences are. Instead of asking 'where are these sequences?', as we did in RQ1 and RQ2, here we ask 'what makes these sequences special?'. This is the general setting for *motif discovery*, where one searches for the re-occurring patterns that are associated with the specific biological functions, or *motifs*, of a set of sequences. Motif discovery is a widely studied topic, and thus, pinpointing the focus is of great importance. Our focus is on improving the performance of protein motif discovery approaches, while aiming to avoid sacrificing the quality of predictions for putative motifs. Thus, we formulate our third and last research question (RQ3) as follows: *can we speed up protein motif discovery without sacrificing its predictive quality?*

Table 1.1. Connections between the research questions 1–3 (RQ1–RQ3), and the publications addressing them (marked with an **x**). We will refer to the publications as [PX], where X is the roman number of the publication.

Research question	[PI]	[PII]	[PIII]	[PIV]
RQ1	**x**			
RQ2		**x**	**x**	
RQ3				**x**

1.3 Outline

The rest of the thesis is organized as follows. In Chapter 2 we present common definitions used throughout the thesis and review the background that is necessary for introducing the approaches and results presented in the later chapters.

In Chapter 3 we focus on answering the first research question (RQ1) by reviewing [PI], wherein we introduce a genomic alignment search tool (GAST) for improving the speed of alignment and mapping tasks with long pattern sequences over previous approaches.

In Chapter 4 we review initial work on multi-pattern matching with indexed text, presented in [PII]. This work provides the first experimental results on indexed multi-pattern matching, setting the background for the next chapter and answering research question 2 (RQ2).

In Chapter 5 we focus on answering the second research question (RQ2) by reviewing [PIII], while presenting new theoretical insights into indexed multi-pattern matching and experimental results with data sets consisting of short sequences of up to 200 nucleotides.

In Chapter 6 we address the third research question (RQ3) by reviewing [PIV], while introducing a graph-theoretical approach to protein motif discovery.

Finally, in Chapter 7 we discuss the contribution of this thesis and the associated publications, and give recommendations for further research on string searching methods in bioinformatics.

See Table 1.1, which summarizes the connections between the research questions and the publications addressing them.

2. Background

2.1 Common Definitions

A *string* or *sequence* $S = S[1..n] = S[1]S[2]\cdots S[n]$ of length n is a chain of *symbols* (a.k.a. characters or letters). Each symbol is an element of an ordered, finite *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. The *alphabetical order*, also known as *lexicographical order* "$<$" among strings is defined in the obvious way. A *substring* of S is defined as $S[i..j] = S[i]S[i+1]\cdots S[j]$, where $i, j \in [1, n]$. A substring of length k or q can be commonly referred to as k-mer, q-gram or q-sample. A *prefix* of S is a substring of the form $S[1..j]$, and a *suffix* is a substring of the form $S[i..n]$. If i is greater than j, then the substring $S[i..j]$ is the empty string ε of length 0. A *subsequence* of S is defined as a string that can be derived from S by deleting some symbols of S, without changing the order of the remaining symbols, e.g. $S[i]S[j]S[k]S[l]$, where $1 \leq i < j < k < l \leq n$. A *text* string $T[1..n]$ is a string terminated by the special symbol $T[n] = \$ \notin \Sigma$, smaller than any other symbol in Σ.

The operation $\texttt{rank}_c[i]$ on a string S returns the number of occurrences of character c in the prefix $S[1..i]$. The operation $\texttt{select}_c[i]$ on a string S returns the position of the i^{th} occurrence of character c in the string S, counting occurrences from left to right. When confusion is possible, we will use $S.\texttt{rank}_c[i]$ and $S.\texttt{select}_c[i]$ to explicitly denote the string S the `rank` and `select` operations are performed on.

An *exact match* or *occurrence* of a pattern P of length m in a text T occurs at position i if $P[1..m] = T[i..i+m-1]$. The operation of locating all the exact occurrences of a pattern P in a text T and reporting their positions in the text is referred to as *locating*. Another operation, where only the total number of such occurrences is reported, is referred to as

counting.

2.2 Index Structures

2.2.1 Inverted Indexes

An inverted index [80] consists of a mapping from a list of words to their occurrences in a text T. These words can be any collection of substrings of the text, e.g. all words occurring in a natural language text. When the text is not naturally split into distinct words, a possible approach is to create the list of words by collecting all substrings of fixed length k. This approach is referred to as k-mer indexing or k-gram indexing. It is crucial to note that the choice of words limits the queries that can be made against these indexes. For a text of length n, a naive implementation of inverted index containing pointers to f occurrences of words takes $O(f \log n)$ bits of space [80].

Block addressing indexes. In block addressing [49], the text T is initially divided into p documents or blocks of size b. Now, the list of pointers for each word point to individual blocks instead of exact positions in the text T. Note that in this setting a single block may contain multiple occurrences of a single word. Due to this property, the total number of pointers $g \leq f$. Additionally, the number of blocks p is notably less than n, depending on the chosen block size b. Thus, a naive implementation of block addressing index occupies only $O(g \log p)$ bits [80], which is typically significantly less than $O(f \log n)$ in practice. As both g and p depend on block size b, the size of the resulting index structure can be easily adjusted by changing the block size. By adjusting the block size, it is possible to support sublinear time queries in sublinear space using block addressing [3].

2.2.2 Suffix Trees

The suffix tree [77] $\mathcal{S}$ of a string $T[1..n]$ represents all substrings of T in a rooted, directed tree. In this tree, each internal node has at least two children and at most one outgoing edge for each $c \in \Sigma$, where c is the first character of the label of the edge. Edge labels are encoded as a reference to T, e.g. a pair of starting and ending text positions. For the node v, $\mathcal{S}.path(v)$ is the concatenation of edge labels from the root to the node v and the *string depth* $\mathcal{S}.sdepth(v)$ is the length of this concatenation. The

suffix tree has $O(n)$ nodes and, if T is terminated with a special symbol $\$ \notin \Sigma$, the resulting suffix tree has exactly n leaf nodes, one for each suffix of T.

A suffix tree of a string T of length n can be built in time $O(n)$ using negligible working space [26, 75]. The resulting suffix tree requires $O(n \log n)$ bits of space. Assuming constant size alphabet, we can find the subtree containing the occurrences of a pattern P of length m in time $O(m)$ by starting from the root and following the edge labels of the suffix tree.

2.2.3 Suffix Arrays

Suffix array SA is an array of length n with $\texttt{SA}[i]$ corresponding to the starting position of the lexicographically i-th smallest suffix in the text T of length n [24, 48]. Suffix array allows one to find the suffix array interval containing the occurrences of a pattern P of length m in time $O(m \log n)$, using binary search. There is an important connection between the suffix tree and the suffix array: the leaves of suffix tree have a one-to-one connection to the values of suffix array.

2.3 Compressed Index Structures

The most relevant compressed index structures in the scope of this thesis belong to the groups of *compressed suffix arrays* and the *compressed suffix trees*. In this section, we will review these compressed index structures. Additionally, we will review two data structures, which are used and needed by these compressed index structures, namely *Burrows-Wheeler transform* and *wavelet tree*.

2.3.1 Burrows-Wheeler Transform

The *Burrows-Wheeler transform* $T_{BWT}[1\,..\,n]$ [8] of text T is a string of length n with $T_{BWT}[i] = T[\texttt{SA}[i] - 1]$ if $\texttt{SA}[i] > 1$ and $BWT[i] = T[n] = \$$ otherwise. In the frame of indexing, one of the key features of a T_{BWT} is the ***LF-mapping***, allowing access from $T_{BWT}[i]$, corresponding to $T[k]$, to $T_{BWT}[i']$, corresponding to $T[k-1]$. For this mapping, we need an array $C[0\,..\,\sigma+1]$, where $C[c]$ is the number of characters that are lexicographically smaller than c in the text T. Note that $C[0] = 0$ and $C[\sigma+1] = n$.

Now, the LF-mapping can be defined as $LF(i) = C[T_{BWT}[i]] +$

$T_{BWT}.\texttt{rank}_{T_{BWT}[i]}[i]$ [12]. By following LF-mapping, one can get from any character $T_{BWT}[i]$ in the BWT to the character $T_{BWT}[i'] = LF(i)$ that precedes it in the text T. This comes with the exception that when looking at the first character of string T, LF-mapping will point to the last character \$ of T, which is lexicographically smaller than any character in Σ. That is, if $\texttt{SA}[i] = 1$, then $LF(i) = 1$.

Using LF-mapping, we can find the interval $[s\,..\,e]$ of T_{BWT} corresponding to the occurrences of a pattern P of length m. We initially set $s = C[P[m]]$ and $e = C[P[m] + 1]$. Then, for each remaining character $P[i]$ in P from right to left, we update s and p by setting $s = C[P[i]] + T_{BWT}.\texttt{rank}_{P[i]}[s - 1] + 1$ and $e = C[P[i]] + T_{BWT}.\texttt{rank}_{P[i]}[e]$ [12]. Thus, finding the interval requires $O(m)$ rank operations. This is commonly referred to as *backward search*.

2.3.2 Wavelet Tree

The wavelet tree $\mathcal{W}$ of a text T, with alphabet Σ of size σ, presents T as a tree of binary strings [25]. This allows one to implement $\texttt{rank}_c$ and $\texttt{select}_c$, where $c \in \Sigma$, using multiple constant time, binary rank and select operations [9], which work on binary strings.

Consider each character c of text T as a sequence of $\lceil \log \sigma \rceil$ bits. Now, the top level of wavelet tree $\mathcal{W}$ consists of the leftmost bits of these bit-sequences of characters in T. At the second level, the left branch corresponds to all the characters for which the bit at the above level was 0, while the right branch corresponds to all characters for which the bit at the above level was 1. Now, the second level consists of the second bits (counting from left) of the bit-sequences of these characters. Within each branch, the bits representing the characters are ordered by the order of the corresponding characters in the text T.

All the remaining levels are constructed identically, splitting each branch from the level above into two, until the wavelet tree has all $\lceil \log \sigma \rceil$ levels. The resulting wavelet tree can be navigated using constant time, binary rank and select operations, in order to answer operations $\texttt{rank}_c$ and $\texttt{select}_c$ in time $O(\log \sigma)$ [25]. The space requirement of a wavelet tree is $n \log \sigma + o(n \log \sigma)$ bits.

Now, by combining the properties of a wavelet tree and Burrows-Wheeler transform, we note that it is possible to find the interval $[s\,..\,e]$ of T_{BWT} corresponding to the occurrences of a pattern P of length m in time $O(m \log \sigma)$ [12].

2.3.3 Compressed Suffix Arrays

Compressed suffix arrays (CSAs) simulate suffix arrays, aiming for reduced space [11, 12, 25, 61, 65]. Typically, compressed suffix arrays are *self-indexes*, meaning that the text T for which the CSA has been constructed does not need to be saved separately, as it can be recreated from the index. We will go through one of the CSAs in more detail, namely a wavelet tree based *FM-index* [55].

FM-index. The *FM-index* [11] combines samples of suffix array values with the backward steps made possible by LF-mapping of BWT. As the sampling, we save the position in the T_{BWT} for every $(\log^{1+\epsilon} n)$th character in the text T. We save these positions as ones in a binary string. We also save the locations for each of these characters in the text, requiring $O(n/(log^{1+\epsilon} n) \log n) = o(n)$ bits of space.

Now, for a position i in BWT, we can get the corresponding position in the text T by taking steps back with LF-mapping until we hit a sampled position. If the sampled position points to position k in T and we took s steps back, we know that the character $T_{BWT}[i]$ is at position $k + s$ in the text T. This takes at most $O(\log^{1+\epsilon} n \log \sigma)$ time, as each LF-mapping operation takes $O(\log \sigma)$ time with wavelet tree of the BWT [12, 55].

Resulting, for a pattern P of length m, the FM-index can retrieve the interval $[s\,..\,e]$ of T_{BWT} corresponding to the occurrences of P in T, i.e. do *counting* or *backward search*, in time $O(m \log \sigma)$. Moreover, it can report all the positions of occ occurrences of P in T, i.e. do *locating*, in time $O(m + occ \log^{1+\epsilon} n \log \sigma)$. The FM-index described above requires $n \log \sigma + o(n \log \sigma)$ space. This can be further brought down to $nH_k + o(n \log \sigma)$ space without compromising the time bounds [13, 55]. Here H_k is the k^{th} order empirical entropy of T_{BWT}, which is always less than $\log \sigma$. Essentially, this means that the FM-index can obtain similar compression to BWT-based compression tools.

2.3.4 Compressed Suffix Trees

Compressed suffix trees (CSTs) simulate suffix trees, aiming for reduced space [15, 64, 68]. CSTs typically consist of a presentation of a CSA and a number of other data structures allowing typical operations of a suffix tree. A CST may also contain the inverse suffix array $\texttt{SA}^{-1}$ in a compressed form. For a CST $\mathcal{T}$ we denote the operation of accessing these two arrays as $\mathcal{T}.\texttt{SA}[i]$ and $\mathcal{T}.\texttt{SA}^{-1}[i]$, where the latter returns the lexicographi-

cal rank of $T[i]$.

For navigational operations, a *balanced parentheses* presentation of the structure of the tree $\mathcal{T}$ is saved. This enables operations such as accessing the children, parent or siblings of a node in $O(1)$ time. Moreover, the lowest common ancestor of two nodes and the subtree-size of a node can also be retrieved in $O(1)$ time [28]. Essentially, CSTs can support any operations that suffix trees can support, with up to $polylog(n)$ slowdown [68]. The balanced parentheses presentation occupies $2n + o(n)$ bits of space [21, 33, 52]. In total, compressed suffix tree requires $O(n \log \sigma)$ bits of space [68].

2.3.5 Compressed Bidirectional Indexes

Bidirectional FM-index. A *bidirectional FM-index* I of string T consists of a *forward* and *reverse* index. The forward index supports backward search in T, and the reverse index in T^R, where T^R denotes the reversed string of T. Lam et al. [37] and Schnattinger et al. [71] showed how to synchronize the forward and reverse index to support bidirectional search. Let P denote a pattern, and let $[sf\,..\,ef]$ denote the SA interval of the suffixes of T whose prefixes match P, and $[sr\,..\,er]$ denote the suffixes of T^R matching P^R. Now a *bidirectional search step* allows us to find out the new interval corresponding to either cP or Pc for any symbol $c \in \Sigma$. The new interval is empty if the pattern is not found.

We require the following operations. The *direction* of the operation is given by the parameter $d \in \{\mathsf{left}, \mathsf{right}\}$:

- `pushChar`$(d, c, [sf\,..\,ef], [sr\,..\,er])$: Assume that $[sf\,..\,ef]$ and $[sr\,..\,er]$ correspond to the pattern P. The operation returns new intervals corresponding to the concatenated pattern cP if $d = \mathsf{left}$, or Pc if $d = \mathsf{right}$. The operation returns an empty interval if the concatenated pattern does not occur in T. Both [37] and [71] show how to support this operation. The latter uses a wavelet tree for the task and supports the operation in $O(\log \sigma)$ time.

- `getBranches`$(d, [sf\,..\,ef], [sr\,..\,er])$: Returns a subset of symbols, that is, all symbols $c \in \Sigma$ having a non-empty `pushChar`$(d, c, [sf\,..\,ef], [sr\,..\,er])$ interval. Let T_{BWT} and T^R_{BWT} be the Burrows-Wheeler transforms of the text and its reverse, respectively. If $d = \mathsf{left}$, it returns the set of

distinct symbols occurring in $T_{BWT}[sf\,..\,ef]$, and if $d =$ right, it returns the distinct symbols occurring in $T^R_{BWT}[sr\,..\,er]$. This can be done in $O(\log \sigma)$ time per distinct symbol with a wavelet tree [20].

The space usage for the bidirectional FM-index is twice that of an FM-index based on wavelet tree, i.e. $2n \log \sigma + o(n \log \sigma)$ bits [55] for a text of length n.

Bidirectional indexing with a CST. A bidirectional search step, from P to Pc or cP, can be simulated in a CST $\mathcal{T}$. Let a node k be the node with the smallest string depth among the nodes for which a prefix of $\mathcal{T}.path(k)$ matches with the pattern P. Let a node u be the parent node of the node k. Now the *state(P)* is defined by the node u and the suffix $P[\mathcal{T}.sdepth(u) + 1\,..\,|P|]$. Using this notation, a *right step* in $\mathcal{T}$ adds a character c to the right side of P and updates the state from state(P) to state(Pc). A right step can be taken by simply following the edges of the CST, if possible with given c. A *left step* in $\mathcal{T}$ adds a character c to the left side of P and updates the state from state(P) to state(cP). A left step can be taken by following the Weiner link [7, 77] wl(c, k) from the node k with the character c. This Weiner link will point to the node k' in $\mathcal{T}$ with the smallest string depth among the nodes for which a prefix of $\mathcal{T}.path(k')$ matches with cP. Let u' be the parent node of k' and let $P' = cP$. Now $state(cP)$ will be correctly defined by the node u' and the suffix $P'[\mathcal{T}.sdepth(u') + 1\,..\,|P'|]$. If no such node k' exists, a left step cannot be taken with this character c.

2.4 Pattern Set Preprocessing for Indexed Text

Preprocessing of a pattern set to be searched in a text index is mainly affordable in a scenario, where the pattern set is to be matched to several text indexes. Scenarios of this kind arise, for example, in *read alignment metagenomics*, where the pattern set represents the DNA of several species and the goal is to find out which species are represented in the sample and in which quantity. Chapter 5 describes and focuses on this specific metagenomic setting in more detail.

This problem frame of indexed multi-pattern matching is a relatively unexplored one. Recently, Gagie et al. [19] gave the first theoretical improvement for indexed multi-pattern matching over the approach of searching each pattern separately. It is shown in [19] that

a given FM-index for the text of length n and the LZ77 parse of the concatenation of p patterns of total length M and maximum individual length m, one can count the occurrences of each pattern in a total of $O((z+p)\log M \log m \log^{1+\epsilon} n)$ time, where z is the number of phrases in the parse. First experimental results on indexed multi-pattern matching were given by [PII] and [PIII], latter of which also presents new theoretical insights into this problem field. We will review these results in Chapters 4 and 5.

2.5 Sequence Alignment

Levenshtein [42] or *edit distance* between two strings S_1 and S_2 is defined as the number of *edit operations* required to convert S_1 to S_2. The allowed operations are insertion, deletion and substitution of a character. An *approximate match* with up to k differences between a pattern $P[1\,..\,m]$ and a text T occurs at the position i if the edit Levenshtein distance D_L between $P[1\,..\,m]$ and $T[i\,..\,i+m-1]$ is $\leq k$. A set of edit operations converting S_1 to S_2 can be presented as an *alignment* between S_1 and S_2

In *sequence alignment* a typical task is to find such an alignment corresponding to minimal number of edit operations between a pattern and a text string. In the frame of sequence alignment, the terms query sequence, query, and pattern will be used interchangeably to stand for the sequence the user wishes to align or map to a database text sequence or sequences. More precisely, typically a *weighted local alignment* is preferred.

In weighted alignment, each edit operation has its own positive or negative weight, resulting in a scoring corresponding to the alignment. Typically, insertions or deletions will have negative impact on the score, while matching characters between S_1 and S_2 in the alignment will have a positive impact on the score. A substitution may have a positive or negative impact, depending on the similarity between the original character and the substituting character. E.g. in protein sequence alignment a substitution of an amino-acid with another similar amino-acid will have a positive impact, while substituting with a very dissimilar amino-acid will have a negative impact. Now, in weighted local alignment, the goal is to find substrings S_1 and S_2 of P and T, respectively, resulting in an alignment with maximal score. Moreover, in *affine gap alignment* the first insertion or deletion will be penalized more heavily than successive insertions or

deletions of neighboring characters.

Optimal weighted local alignment can be solved with Smith-Waterman algorithm [72], using dynamic programming [10]. Solving optimal affine gap alignment requires slight modifications of the Smith-Waterman algorithm [1], e.g. using three arrays for dynamic programming instead of a single one. However, these algorithms are slow in practice, resulting in need for faster, more practical solutions when dealing with e.g. genome scale data.

BLAST [2] and its successors are addressing this need for an efficient sequence alignment approach. It is noteworthy, that sequence alignment is a very common task in modern bioinformatics. NCBI BLAST [53] alone receives over 100,000 alignment queries a day. The computational requirements of these searches amount to a very notable use of resources.

2.6 Motif Discovery

Whereas the sections 2.1 – 2.5 have been more related to finding the occurrences of known patterns in a text, *motif discovery* is more about finding meaningful, yet initially unknown patterns in a specific group of sequences. Typically, this kind of group of DNA or amino-acid sequences have specific properties in common and the task is to find re-occurring patterns that are associated with specific biological functions, a.k.a. *motifs*. The goal is to essentially encapsulate the meaningful properties of given DNA or protein sequences and pinpoint the subsequences that are most likely to be responsible for these properties.

The biological definition of a motif is not unambiguous or straightforward, though. This problem has been addressed by dividing the motifs into various categories. On top of the natural division into DNA (or RNA) and protein motifs, e.g. Frith et al. [18] have divided motifs into three classes. The first class contains short motifs occurring at functional sites of biopolymers, e.g. binding or cleavage sites. The second class contains longer protein motifs associated with globular structural domains, usually occurring through divergent evolution, while the third class contains recurring motifs that can appear through evolutionary recent duplications. Due to the complexity of each of these classes and the variety of biological motifs in general, it seems improbable that they could all be tackled by a single motif discovery approach.

Many algorithms have already been developed for motif searching.

While some of these algorithms are specially designed for the discovery of DNA motifs, e.g. Weeder [59] and AlignACE [63], other ones can be applied to search for both protein and DNA motifs, e.g. MEME [4], Gibbs [56] and GLAM2 [18]. At the same time, there are also many databases specially designed to include DNA regulatory motifs, e.g. TRANSFAC [50], JASPAR [70], and protein motifs, e.g. PROSITE [32], ELM [60]. In Chapter 6, we review an approach for protein motif discovery presented in [PIV].

3. Indexed Approximate Alignment of Long Sequences

While BLAST-like alignment tools are able to align any kind of sequences, for a notable amount of cases, using these approaches is computationally excessive. This is especially true in the case of long sequences, of length 200 nucleotides and more. For this setting, we have presented a drastically faster approach in [PI], which will be reviewed in this chapter. Initially, our approach was designed for finding approximate matches for sequences of over 1000 nucleotides in length. Later, we noticed that the developed approach was able to handle shorter sequences as well, as long as the sequences were at least 200 nucleotides long.

3.1 Previous Alignment Approaches

At the time of writing of [PI], concerning the speed, two very popular alignment methods, Mega BLAST [51, 84] and BLAT [35], stood out. Mega BLAST's performance is increased by using a "greedy algorithm", which starts three different lines of further processing whenever an error is encountered. These three lines correspond to (i) handling a mismatch, (ii) an insertion in the query, and (iii) a deletion in the query. When a difference between the query and the database occurs, one of the lines is likely to continue running as the other two will terminate immediately. With high similarity between the query and the database, this method is computationally very effective. Mega BLAST also uses an index collecting the occurrences of, by default, every fifth 12-mer in the text. The exact occurrences of such 12-mers between the pattern and the text are extended using the greedy alignment algorithm.

BLAT uses indexing of all non-overlapping 11-mers in the database. The index is used in a search phase to connect these k-mers to the k-mers of the query sequence. Over-occurring 11-mers are not taken into

account when this mapping is being done. Lastly, an alignment is done by extending the sites found in the search phase.

3.2 Methods of the Genomic Alignment Search Tool

Our method can be divided into three different phases: the creation of a block addressing q-sample index, the initial search phase, and lastly the alignment phase, where the results of the initial search phase are processed in a greater detail. The index phase is a preprocessing step, which has to be done only once for each genome or other collection of database sequences. Initial search phase uses the index created to find potential sites having high probability of leading to a good alignment. The alignment phase performs a more precise alignment between these sites and the patterns provided. In this section, we will review the essentials of these three phases. The full workflow of our tool in these three phases is illustrated in Figure 1 in [PI], and the full details on the workings of this approach can be found in Section 2 in [PI].

3.2.1 Block addressing Q-sample Index

Our tool uses an index file to gain speed-up in the initial, approximate search. Essentially, this index structure combines q-sample filtration [74] with block addressing [49].

The index structure is formed as follows. Given database files containing the database sequences are initially divided into *blocks* of given size b. Then, the database sequences are scanned for occurrences of a certain dinucleotide, *AC*. These dinucleotide occurrences are expanded to what we call *AC-probes*. This expansion is done by taking the ten nucleotides following the dinucleotides AC, resulting in 12-mers. The blockwise locations of these probes are collected and overly occurring AC-probes are discarded.

As the result, we have an index consisting of lists of block ID numbers for the collection of remaining, non-discarded AC-probes. Using this index structure, our tool can rapidly retrieve blocks with occurrences of a given AC-probe, or a collection of multiple AC-probes.

Even though we consider our choice of dinucleotide AC good for most data, based on low mean and low variance of incidence in bacteria, archae and eukaryotes alike [34, 83, 82], the indexing could as well be based on

another dinucleotide or longer k-mer.

Additionally, as another preprocessing step, we k-mer encode the text and save it in a binary format.

3.2.2 Initial Search

The initial search phase compares the AC-probe profiles of database blocks, which were retrieved in the indexing phase, to the AC-probe profiles of patterns. If a block has many AC-probes in common with the pattern, the block in question is considered having a high probability of containing an approximate occurrence of the pattern. Such blocks will be further refined in the alignment phase, or if specified by the user, the search can be stopped here and the blocks together with the amount of matching AC-probes in them will be reported.

3.2.3 Alignment

The last refining phase in our tool is the alignment phase. In this phase, we use BG algorithm [69] together with the k-mer encoded text to find short (11 nt) exact matches between the pattern and the blocks of text that were passed on by the initial search phase. The promising clusters of such exact matches are extended into alignments between parts of the pattern and parts of the text block. See Section 2.3 and 2.4 in [PI] for details on thresholds defining promising clusters, and the reasoning for choices behind these and other parameters.

As the output, our approach, the Genomic Alignment Search Tool (GAST), reports the start and end sites of aforementioned alignments in both the pattern and the database. The number of mismatches and gaps is also reported. Additionally, there is an option to output an approximated alignment.

3.3 Experiments

The GAST algorithm reviewed in Section 3.2 and more elaborately described in Section 2 in [PI] was implemented in C++. The query time and error tolerance of GAST on a set of typical alignment tasks was compared with those of general alignment tools Mega BLAST [84] and BLAT [35]. On a separate set of experiments the query time on an exon mapping task and the mapping quality of these three approaches were compared with

those of a mapping tool GMAP [81]. In this section, these experiments are reviewed. See Section 3 in [PI] for full details.

All the runs were performed on a machine with 1GB DDRII RAM (667MHz) and an Intel Core 2 Duo T5500 (1.66 GHz) processor, running Ubuntu 7.04. All the run times in this section are times used by the program itself and any library subroutines it calls. The tests were later repeated on another machine with 6 GB of RAM in order to eliminate possible paging effects. No bias of this sort was detected.

Data. When comparing the performance of GAST and Mega BLAST on general alignment, searches were made against a database consisting of the whole human genome received from the Ensembl genome database [30]. The release in question was based on the NCBI 36 assembly of the human genome. In the case of BLAT, the system used for the runs lacked the memory to perform searches against the whole human genome. Therefore, another set of searches with BLAT, Mega BLAST, and GAST were performed against the chromosome 1 of the same genome. The patterns used in the alignment tasks will be described separately below.

For the exon mapping tasks, we had a collection of 6721 cDNA sequences, corresponding to various transcripts originating from human chromosome 1. The sequences were retrieved from the BioMart database [57] and were 2000 nucleotides long on average. The starting and ending positions of exons in the sequences were also retrieved.

Results. Necessary preprocessing for GAST, Mega BLAST and BLAT was made for the full genome and the chromosome 1 separately. On the full genome, preprocessing for GAST took 639.3 s, while for Mega BLAST it took 157.6 s. The index required to be read in memory totaled 79.5 MB in the case of GAST and 734.8 MB in the case of Mega BLAST. For full details, see Section 3 and Table 1 in [PI].

Query times for aligning the sequences of length 1000 and 5000 nucleotides on data sets described above were tracked for the three tools. These sequences were randomly sampled from the text. The results can be seen in Figures 3 and 4 in [PI]. As a summary of these results, we have listed average query time per pattern for these experiments in Table 3.1.

Comparing these average query times for the sequences of length 1000, GAST was 50.0 times faster than Mega BLAST on the full genome, but only 18.3 times faster on the chromosome 1. For query sequence length 5000, the respective numbers were 72.1 and 19.1. With chromosome 1 and query lengths 1000 and 5000, GAST was 4.8 and 10.2 times faster

Table 3.1. Average query time per pattern in milliseconds for GAST, BLAT and Mega BLAST on chromosome 1 and full genome, with query lengths 1000 and 5000 nucleotides.

Database	Chromosome 1		Full Genome	
Query length	1000	5000	1000	5000
GAST	84.09	374.51	106.22	380.3
BLAT	399.69	3824.64	N/A	N/A
Mega BLAST	1536.09	7151.8	5308.82	27445.76

Table 3.2. The run times for the mapping of 6721 cDNA sequences on human chromosome 1, allowing introns.

BLAT	Mega BLAST	GMAP	GAST
286m 40.3s	45m 19.2s	14m 52.4s	1m 13.7s

than BLAT, respectively.

We also compared the error tolerance of GAST, Mega BLAST and BLAT, by adding increasing quantities of random point mutations to the query sequences of length 5000. We noted that Mega BLAST and BLAT reliably aligned sequences to roughly correct locations up to an error rate of 0.12, while GAST reported correct alignments up to an error rate of 0.17. By adjusting the parameters of GAST, we were able to align sequences reliably up to an error rate of 0.25.

Lastly, GAST, Mega BLAST, BLAT and a mapping tool GMAP were tested on a set of exon mapping tasks. The 6721 cDNA sequences were mapped against the human chromosome 1. The query times for this task are listed in Table 3.2. Figure 3.4 in [PI] depicts the exon mapping quality of each tool. To encapsulate the mapping qualities, we note that all tools were able to reliably map exons of length 30 nucleotides and up. As exon length increased above this, Mega BLAST and GAST achieved roughly an exon mapping quality of 0.95, while BLAT and GMAP achieved quality of 0.97 and up.

The run times given in Table 3.2 show remarkable differences between the four tools, GAST being the fastest. The notable relative increase of run time for BLAT is mainly due to disabling fast DNA/DNA remapping, which needs to be disabled to allow introns.

Analysis of results. We have shown that GAST is capable of both error tolerant alignment and high quality exon mapping. This suggests that

the presented approach is very suitable for typical approximate matching problems. Regarding query times, GAST was able to outperform all the other tools in the comparison by a very notable margin. As the length of the pattern or the length of the text increased, the relative gap between GAST and other approaches grew further.

The main restriction of GAST is that the patterns have to be long enough to contain enough AC-probes to work with in the initial search phase. Due to this limitation, we do not encourage using the presented approach for patterns below 200 nucleotides in length. However, we would like to note that before proceeding in searching a pattern, it is possible to draw conclusions on whether or not we can reliably produce an extensive list of best hits for the pattern, based on the number of AC-probes the pattern contains. Note that due to block addressing, individual exons can still be as short as 30 nucleotides.

We have demonstrated that by combining block addressing and q-sampling, it is possible to develop a mapping and alignment approach that is fast and has a relatively small and adjustable memory footprint.

4. Multi-pattern Matching with Compressed Suffix Arrays

In the problem frames of bioinformatics, it is not uncommon to search for multiple sequences successively in the same text database. However, the possible improvements related to searching multiple patterns at once were not studied very broadly at the time of writing [PII], when considering the cases of searching the patterns in an index structure.

The focus of this chapter is to seek possible improvements in one case of searching multiple patterns in an index structure. The index structure that is being considered is a self-index, the compressed suffix array (CSA) [12, 25, 61, 65]. More specifically, this chapter focuses on the cases where one or more preprocessed sets of patterns are being searched in multiple preprocessed text databases. In such a problem frame, the preprocessing of a set of patterns needs to be done only once per set, but as the single set will be searched multiple times, the cost of the preprocessing is spread over multiple searches. Because of this, it is not reasonable to take the preprocessing times directly into account when looking at the run-time of a single search.

Moreover, the focus of this work is on exact matching which can be seen as a starting point for more practical implementations, including approximate matching. Even in bioinformatics, where exact matching is rarely sought after, it is noteworthy that a large number of successful tools use exact matching as part of a seed-and-extend methodology.

Lastly, this chapter focuses on a setting where we have the CSA of the text and a separate copy of the text itself, allowing swift extraction of substrings of the text.

Previous work. The idea of exploiting common substrings in alignment was first proposed in [38]. Landau and Ziv-Ukelson showed how to compute the part of the edit distance matrix corresponding to a common substring only once, such that one could extend the alignment directly at all

occurrences of the common substring. Although more general than ours in supporting approximate search, the authors do not consider a setting with indexed text.

4.1 Methods for Multi-pattern Matching with Compressed Suffix Arrays

In this section, we review an approach initially presented in [PII]. The workings of the proposed method are divided into three work phases: preprocessing of the text, preprocessing of the set of patterns, and searching for the set of patterns in the text. The two preprocessing steps need to be done only once for each set of patterns and each text. The search phase uses both of these preprocessing steps to improve speed in the search.

4.1.1 Preprocessing of Text

The text is preprocessed by making a compressed suffix array [66] of it. The implementation provided in the Pizza&Chili corpus [14] is used to produce this index.

The most important functionality for the searches that are the focus of interest of this work is the *locate* function. Locate function allows for location of the occ occurrences of a query of length m in a text of length n in $O(m \times \log(n) + occ \times \log^{\epsilon}(n))$ time. Here ϵ belongs to $(0, 1)$, depending on the chosen time-space tradeoff.

4.1.2 Preprocessing of Patterns

The set of patterns is preprocessed in order to find a certain set of substrings of the patterns. The goal is to find a collection of substrings, where each substring would occur in a large number of patterns, while still occurring comparatively rarely in the text.

As our initial pool of substrings, we use all phrases produced by the Re-Pair compression tool [41] when ran on the set of patterns. To select substrings with few occurrences in the text, we apply a minimum substring length threshold to this set of substrings. Remaining substrings are then sorted in descending order by the number of patterns in which the substring occurs and this sorted list is saved. The patterns in which each substring occurs and the respective offsets from the start of the pattern are saved, as this information is needed in the search phase.

4.1.3 Searching a Set of Patterns in Text

In the search phase, the preprocessed set of patterns is searched in the preprocessed text. The substrings obtained during the preprocessing are searched in the text in descending order by the number of patterns in which they occur, using the locate functionality of the CSA. For each occurrence of a substring, the possible occurrences of the patterns that include the substring are checked by character comparison. First the pattern is compared, character by character, with the text, starting from the beginning of the pattern, continuing up to the occurrence of the substring in the pattern. This is followed by comparing the characters of the pattern and the text, starting from the end of the pattern, moving towards the occurrence of the substring. If any mismatch is found during the exact matching or if the whole pattern matches the text, the search continues with processing the next pattern where the substring occurs.

When all occurrences of a substring have been checked with all of the patterns corresponding to the substring, all of these patterns are marked as *treated*. As all occurrences of a pattern are found when checking all occurrences of a substring of the pattern, the patterns that are marked as treated need not to be checked when handling later substrings.

After all of the substrings obtained from the preprocessing have been handled, the remaining patterns that are not yet marked as treated are searched using the locate functionality of the compressed suffix array for the full pattern. Alternatively, the search using the substrings can be terminated after a pre-selected amount of patterns have been marked as treated, finishing the remaining patterns with the locate functionality.

4.2 Experiments

The approach reviewed in Section 4.1, and described in Section 2.2 in [PII] in more detail, was implemented in C++. All the experiments were carried out using a single Intel®Core™i7 CPU 860 @ 2.80 GHz (8192 kB cache), with 16 GB RAM, running Ubuntu 10.04.

Data. The text used was a DNA text of 50 MB in size, obtained from the Pizza&Chili corpus [14]. The set of patterns consisted of 1000 substring of length 1000, sampled uniformly at random from the text. It was noticed that each of these patterns occurred exactly once in the text.

Results. The pattern set described above was preprocessed as described in Section 4.1.2. Minimum substring length used was varied between 25 and 35 nucleotides, resulting in total preprocessing time varying between 0.836 and 0.800 seconds, respectively. After this preprocessing of the pattern set, text was preprocessed by creating a compressed suffix array of it, using default parameter values. The creation of the index took 22.69 seconds and the total size of the resulting index was 36.8 MB.

The preprocessing steps were followed by searching the set of patterns from the text. Searches were done separately for each of the minimum substring lengths: 25, 28, 30 33 and 25. Additionally, the number of patterns allowed to be searched with the proposed method varied from 100 to 500. However, the actual number of patterns that had common substrings of required length within them was in some cases less than this, resulting in a smaller number of patterns being handled with the proposed method. The run-times of the proposed method were compared with searching all of the patterns with the locate functionality of the compressed suffix array implementation.

Looking at the full runs of 1000 patterns, the best results were retrieved when using a minimum substring length of 30, resulting in 14.0% saving in run-times, when 238 patterns were found by using the proposed method. When considering the average time for a single pattern to be found by searching the substring and then checking the exact match, the best results were retrieved when using a minimum substring length of 35, resulting in 71.6% saving in run-times per pattern, when 155 patterns were found by using the proposed method.

When a pattern was handled using the proposed approach, locating subpatterns took roughly 88% of the query time, leaving 12% for the exact matching, averaging over all different minimum substring lengths. For more elaborate details, see Section 3 in [PII].

Analysis of results. Locating all occurrences of certain substring of a pattern using CSA, and then verifying them using naive exact matching proved to be a reasonable way to improve query times in indexed exact matching of multiple patterns. In this rather direct approach, there is a definite tradeoff between how much performance increase per pattern can be gained and for how many patterns can this be applied to, when choosing a suitable minimum substring length. See Chapter 7 for a broader discussion on how this result could be more generally used.

5. Multi-pattern Matching with Bidirectional Indexes

In *metagenomics* a mixture of genomic material is sequenced from an environmental sample [27]. Typically, millions of short DNA reads are produced from the sample with the length of each read varying between 30 and 400 nucleotides depending on sequencing technology, and subsequent sequence analysis tries to identify the species present in the sample. Sequence analysis can be either *fragment assembly* -based, as in e.g. [36], or *read alignment* -based, as in e.g. [45]. In the former approach, the reads are first assembled into *contigs* (longer fragments glued together based on read overlaps) and then compared against reference genomes to locate statistically significant local alignments. In the latter approach, the reads are directly aligned to reference genomes. In the work reviewed in this chapter, and initially presented in [PIII] and [23], we will focus on this latter approach.

Such alignment can be efficiently done, e.g. using software packages building on the concepts of BWT [8] and FM-index [12], reviewed in Sections 2.3.1 and 2.3.3. Extensions of these data structures provide very efficient methods for doing read alignment with few mismatches, see `bowtie` [40], `bwa` [43], `SOAP2` [44], `readaligner` [47].

In this chapter, we review an approach for multi-pattern matching that takes the special characteristics of metagenomics read alignment into account. The methods above align each read separately without exploiting the fact that read sets typically cover the same genomic position many times. Additionally, repetitive areas cause similar reads to be produced. We are interested in a specific read alignment scenario with a database of metagenomics read data sets and reference genomes. In this scenario, one can afford preprocessing of both kinds of data to speed up the subsequent alignment of new read data set to all known reference genomes as well as alignment of all existing read data sets to a new reference genome.

This scenario is indexed multi-pattern matching, as described in Section 2.4, in the frame of metagenomics. The reviewed approach is currently limited to exact searching; see Chapter 7 for discussion on extensions to approximate search.

The work reviewed in this chapter was originally motivated by the work introduced in [PII] and reviewed in Chapter 4. In this section the focus is on using bidirectional indexes instead of CSAs in the setting of indexed multi-pattern matching.

5.1 Theoretical Results

5.1.1 Preliminaries

In Section 2.3.5 we explained the concept of the bidirectional FM-index, operations `pushChar`, `getBranches`, concept of bidirectional search step and methods of taking these steps with a compressed suffix tree. Let us define a few more notations that will be used in the following sections.

A sequence of left and right bidirectional search steps or left and right steps in a compressed suffix tree is referred to as a *search path* or just path, when no confusion with $path(v)$ should be possible. Let us consider a set of patterns $\mathbb{P} = P_1, P_2, \ldots, P_p$. A search path is said to be a *complete path*, when the path reads all characters in a pattern P_i, thus *handling* the pattern P_i. A *complete path forest* is a collection of trees, containing complete paths, handling the set $\mathbb{P}$. Moreover, a subpattern B *covers* a set of patterns $\mathbb{H}$ it occurs in, $\mathbb{H} \subset \mathbb{P}$.

5.1.2 Bidirectional Search

We construct a bidirectional index for both the text T and the set of p patterns $P_1, P_2, \ldots, P_p$. More precisely, the pattern index is constructed for the concatenated string $S = \#P_1\#P_2\#\cdots\#P_p\#\$$, where $\#$ is a special separator symbol that does not occur in any of the patterns. Let N and M denote the total length of the text and the concatenated string of patterns, respectively. The pattern index stores suffix array (SA) samples only at separator symbol positions. This requires $p \log M$ bits of space, which might be too much for patterns shorter than $\log M$, but allows $O(1)$ time locate for `SA` ranges $[i\,..\,j]$ that are prefixed by $\#$.

We assume that the subpattern P is given as input, and the task is to

locate the occurrences of patterns $P_1, P_2, \ldots, P_p$, that contain subpattern P, in the text T. In other words, for every P_i that has an occurrence of P, we must output all the occurrences of P_i in T. We proceed with the search as follows.

Initially, the subpattern P is searched from the text and pattern indexes, using pushChar operation. This is followed by extending P recursively to both directions, over all combinations of symbols on the left and right side of subpatterns occurrences in $P_1, P_2, \ldots, P_p$. The extension is done alternating between the directions $\{\mathsf{left}, \mathsf{right}\}$ — interleaving left and right symbols during the search. Let us refer to this part of the algorithm as `extend()`. Details of this full algorithm performing operation `search(P)` are described in Figure 1 in [PIII].

Let us now consider the number of steps required when searching a set of patterns using this approach.

Definition 1 *Let $steps(I, P)$ denote the number of steps taken by `extend()` on bidirectional FM-indexes I on calls from `search(P)`. Let $x = lsize(I, P)$ and $y = rsize(I, P)$ denote the search space size, in the worst case scenario of text containing occurrences of all the patterns, using bidirectional indexes I when extending P* only *to the left and* only *to the right, respectively.*

The upper and lower bounds for $steps(I, P)$ are analyzed in Section 3.1 in [PIII]. Theorem 2 below summarizes the resulting bounds.

Theorem 2 *Given text a T of length n, a set of patterns $P_1, P_2, \ldots, P_p$ of total length M, and a query pattern P, one can solve the* indexed subpattern search problem *of locating patterns $P_{i_1}, P_{i_2}, \ldots$ containing P as a subpattern and having an occurrence in T, in time $O((|P| + steps(I, P)) \times \log \sigma)$ after building bidirectional indexes I for the text and for the pattern set. One can bound $lsize(I, P) + rsize(I, P) \leq steps(I, P) \leq lsize(I, P) \times rsize(I, P)$ in the worst case instance of text containing all patterns, using the notions of Definition 1. The bidirectional indexes I required for the query take $2n \log \sigma(1 + o(1)) + 2M \log \sigma(1 + o(1)) + p \log M$ bits.*

5.1.3 Hardness of Subpattern Selection

In the subpattern selection problem, we would like to find a set $\mathbb{S}'$ of subpatterns covering, or handling, all the patterns in the pattern set $\mathbb{P}$ and minimizing the total cost of searching and extending this set of subpat-

terns. In Section 3.2 in [PIII] we analyze the hardness of this problem. The main result of this analysis is an observation of a connection between this problem and the set cover problem.

In Section 3.2 in [PIII] we note that, as the set cover problem, the subpattern selection problem is also NP-hard. However, a positive connection to set cover also exists; an algorithm analogous to the well-known greedy approximation algorithm for weighted set cover [76] can be used to compute a *greedy subpattern cover*: Choose first a pattern P which minimizes

$$\frac{|P| + steps(I, P)}{m(\mathbb{P}, P)}, \tag{5.1}$$

where $m(\mathbb{P}, P)$ denotes the number of patterns in $\mathbb{P}$ which contain P as a subpattern. Set $\mathbb{P} = \mathbb{P} \setminus \mathbb{P}'$, where $\mathbb{P}'$ denotes the set of patterns covered by P. Iterate the process until $\mathbb{P}$ is empty. The set cover analysis [76] can be used verbatim to see that the process results in a set of subpatterns with cost at most $\log p$ times the optimal, where p is the size of $\mathbb{P}$. Notice that here we do not know value $steps(I, P)$ exactly for any pattern, so we will only obtain approximation with respect to our estimate on $steps(I, P)$; the estimation error can be arbitrarily more than the $\log p$ factor from the set cover approximation. Let us later refer to this approach as the *greedy subpattern cover algorithm*.

5.1.4 Subpattern Selection Using Affix Trees

The greedy subpattern cover can be computed using *affix trees* [46, 73]. Here we assume that $steps(I, P)$ is estimated as a function of $lsize(I, P)$ and $rsize(I, P)$, without fixing the exact formula.

The affix tree of a string T incorporates the suffix tree of both T and its reversed string T^R. An internal node in the affix tree can have both suffix and prefix descendants: the outgoing *suffix edges* (resp. *prefix edges*) point to the descendants of the corresponding node in the suffix tree of T (resp. T^R). For each node v in the suffix tree of T (resp. T^R), there exists a corresponding node in the affix tree having the upward suffix edge (resp. prefix edge) labels equal to $\texttt{path}(v)$. The total number of nodes and edges is $O(n)$. Affix trees can be constructed in linear time and space [46].

The greedy subpattern cover algorithm requires us to compute $lsize(I, P)$, $rsize(I, P)$, and $m(\mathbb{P}, P)$ values. The latter values can be computed with the *color set size* algorithm [31]. It stores, for all nodes v in the suffix tree of $P_1\$P_2\$\cdots P_p\$$, the number of patterns in $\mathbb{P}$ which have $\texttt{path}(v)$ as a subpattern. The algorithm requires linear time and space —

we omit the technical details.

To compute $lsize(I, P)$ and $rsize(I, P)$ values, we first build an affix tree for the concatenated string $S = \#P_1\#P_2\#\cdots\#P_p\#\$$, where $\#$ is a special separator symbol, $\# \notin \Sigma$. Using this affix tree, we can take bidirectional search steps to either left or right from any node of the tree. Due to this property, we can find the $lsize(I, path(v))$ and $rsize(I, path(v))$ for each node of the tree. Section 3.3 in [PIII] explains this in full detail.

Using $O(M \log M)$ bits of space and $O(M)$ time, we can save these values and find the pattern minimizing the function shown in Equation 5.1. With this approach, we arrive at the following theorem.

Theorem 3 *Given a set of patterns $P_1, P_2, \ldots, P_p$ of total length M, the greedy subpattern cover algorithm of Section 5.1.3 can be implemented to work in $O(Mp^*)$ time using $O(M \log M)$ bits of space, where $p^* \leq p$ is the number of selected subpatterns.*

5.1.5 Subpattern Selection Using Bidirectional Compressed Suffix Trees

Next, we aim for a solution of $O(M \log \sigma)$ bits. To achieve this, we use compressed suffix trees, one for $S = \#P_1\#P_2\#\cdots\#P_p\#\$$ and one for S^R (i.e. latter being prefix tree). Let us denote these two compressed suffix trees $\mathcal{S}$ and $\mathcal{P}$ (standing for suffix and prefix). In Section 3.4 in [PIII] we describe in detail how to keep the suffix array intervals of these two trees updated and corresponding to each other. Essentially, if we have a node v and its suffix array range $[l\,..\,r]$ in $\mathcal{S}$, we can compute the corresponding suffix array range $[l'\,..\,r']$ in $\mathcal{P}$ by

$$\begin{aligned} l' &\leftarrow \mathcal{P}.\mathtt{SA}^{-1}[n - (\mathcal{S}.\mathtt{SA}[rminq(A, l, r)] + \mathcal{S}.sdepth(v))] \text{ and} \\ r' &\leftarrow \mathcal{P}.\mathtt{SA}^{-1}[n - (\mathcal{S}.\mathtt{SA}[rmaxq(A, l, r)] + \mathcal{S}.sdepth(v))]. \end{aligned}$$

Here $A[i] = \mathcal{P}.\mathtt{SA}^{-1}[n - \mathcal{S}.\mathtt{SA}[i]]$ for $1 \leq i \leq n$, while $rminq(A, l, r)$ and $rmaxq(A, l, r)$ return pointers to the minimum and maximum values in range $[l\,..\,r]$ in the vector A, respectively. With this conversion, it is possible to take left and right bidirectional search steps in this pair of compressed suffix trees and compute $lsize$ and $rsize$ values for all nodes the same way as in Section 5.1.4.

Finally, the space bottleneck in the computation is the storage of $rsize$ values in $\mathcal{S}$ and $lsize$ values in $\mathcal{P}$. The $rsize$ values in $\mathcal{S}$ can be computed during depth-first traversal and need not be stored, but one may still need

to maintain $O(n)$ values in a stack each occupying $O(\log n)$ bits; this can be improved to $O(n)$ bits by maintaining dynamic partial sums data structures both for the stack and for the values following almost verbatim the algorithm in [16].

Storage of $lsize$ values in $\mathcal{P}$ in $O(n)$ bits can be achieved by sampling. There are $O(n/(\log n))$ nodes for which computing $lsize$ takes $\Omega(\log n)$ time. For these nodes, we save the $lsize$ values, which takes a total of $O(n)$ bits. Now, the running time for computing the linking and finding the node minimizing the function shown in Equation 5.1 depends on the chosen compressed suffix tree, but $O(n \log n)$ time can be achieved e.g. using the compressed suffix tree presented in [68]. Thus, we arrive at the following theorem.

Theorem 4 *Given a set of patterns $P_1, P_2, \ldots, P_p$ of total length M, the greedy subpattern cover algorithm of Section 5.1.3 can be implemented to work in $O(M \log M p^*)$ time using $O(M \log \sigma)$ bits of space, where $p^* \leq p$ is the number of selected subpatterns.*

Notice that one can get different time-space tradeoffs and more accurate bounds by choosing an appropriate compressed suffix tree variant.

5.2 Practical Multi-pattern Matching

5.2.1 Practical Preprocessing

In this section, we review the practical approach for pattern set preprocessing, initially introduced in Section 4.1 in [PIII]. This preprocessing, as the approaches reviewed in Sections 5.1.4 and 5.1.5, aims to minimize the query time of finding the occurrences of a pattern set $\mathbb{P} = P_1, P_2, \ldots, P_p$ in a bidirectional index I. The practical approach uses a CST $\mathcal{T}$ of the concatenation $S = \#P_1\#P_2\#\cdots\#P_p\#\$$. Let us denote, for a node v and a subpattern $B = \mathcal{T}.path(v)$

$$\begin{aligned} lsize_v &= lsize(I, \mathcal{T}.path(v)) \\ rsize_v &= rsize(I, \mathcal{T}.path(v)) \\ steps_v &= steps(I, \mathcal{T}.path(v)) \\ m_v &= m(\mathbb{P}, \mathcal{T}.path(v)). \end{aligned}$$

Now, preprocessing searches the tree $\mathcal{T}$ for the node v minimizing the

function

$$\frac{\mathcal{T}.sdepth(v) + steps_v}{m_v}. \tag{5.2}$$

The full details of finding this minimizing node v are given in Section 4.1 and Figure 3 in [PIII], which shows a pseudocode of the preprocessing. First, we prepare $\mathcal{T}$ for calculating m_v for any node in $O(1)$ time, using methods introduced in [67]. This preprocessing takes $O(M \times t_{lca})$ time, where $M = |S|$ and t_{lca} is the time taken by lowest common ancestor operation in the CST. The resulting data structure uses $2M + o(M)$ bits of space on top of the space required by the CST.

After this preparation, we collect and save $lsize_v$ for each v in $\mathcal{T}$. This is done by following the Weiner links, allowing left bidirectional steps to be taken in a CST $\mathcal{T}$, as described in Section 2.3.5. This takes $O(M^2 + \sigma \times M \times t_{wl})$ time, where t_{wl} is the time taken by following a Weiner link, but as we quickly skip previously visited nodes in line 1 of the pseudocode shown in Figure 3 in [PIII], the $\sigma \times M \times t_{wl}$ -term dominates in practice. We save the $lsize_v$ for each node $v \in \mathcal{T}$, requiring $O(M \log M)$ bits of space.

Once $lsize_v$ has been calculated and saved for all nodes, we calculate the $rsize_v$ for each $v \in \mathcal{T}$. This can be done by following the edges of the CST, as described in Section 2.3.5. As we calculate the $rsize_v$ for a node v, we also calculate the m_v and the value of minimizing function 5.2, keeping track of the smallest minimizing function value and the corresponding node. This takes $O(M \times s_{SA} \times t_\phi)$ time, where $s_{SA} \times t_\phi$ is the time it takes to access an element of the compressed suffix array of the compressed suffix tree.

When a minimizing node v for a tree $\mathcal{T}$ has been found, the set of patterns $\mathbb{P}$ is updated by removing the patterns $\mathbb{H}$ that are covered by $B = \mathcal{T}.path(v)$.

A bidirectional index is formed from the #-separated concatenation of patterns in $\mathbb{H}$. Bidirectional search steps are taken in accordance with the subpattern $B = \mathcal{T}.path(v)$. This is followed by taking possible search steps to left and right by turns, keeping track of added characters. Pseudocode describing this in detail is shown in lines 1–20 of Figure 3 in [PIII]. Execution of this algorithm mimics running the `search` algorithm in Figure 1 in [PIII] with a text containing at least one occurrence for each of the patterns in $\mathbb{H}$. The added characters and the corresponding directions of the steps are saved into a tree structure $\mathcal{A}$, which is serialized and saved to a file. This takes $O((\mathcal{T}.sdepth(v) + steps_v) \times \log \sigma)$ time for each optimal node v. Construction of this structure is a practical improvement over the ap-

proaches suggested in Section 5.1.2, moving large portion of the work that was previously done in the searching phase to the preprocessing phase.

Once the tree $\mathcal{A}$ is saved, the preprocessing starts over again with the updated $\mathbb{P}$. This procedure is repeated, until $\mathbb{P}$ is empty, resulting in a file containing a complete path forest $\mathbb{F}$ for the original pattern set. The size of this forest is at most $\log p$ times the optimal, with respect to our estimate on $steps_v$.

As the sets $\mathbb{H}$ sum up to original set $\mathbb{P}$, construction of all the bidirectional indexes can be done in $O(M \log \sigma)$ time, the largest index requiring $2M \log \sigma + o(M \log \sigma)$ bits of space in the worst case.

5.2.2 Practical Searching

The search phase reads the complete path forest $\mathbb{F}$ created by the preprocessing. The pseudocode of the search is shown in lines 21–33 of Figure 4 in [PIII]. The search works with a bidirectional index I of the text T, updating the intervals $[sf\,..\,ef]$ and $[sr\,..\,er]$ of forward and reverse text index, respectively. The intervals are updated by calling the `pushChar` function in accordance with the left and right sequences saved in the edges of the trees in the forest $\mathbb{F}$. As long as $[sf\,..\,ef]$ is not empty, the child nodes of the node currently being processed are processed in the same way, branching the search. Whenever a leaf of a path tree $\mathcal{A}$ is read, a pattern is handled. If the resulting interval $[sf\,..\,ef]$ is not empty, it corresponds to the occurrences of the pattern in the text T.

Let a path tree $\mathcal{A}$ be created from the optimal node v in the tree $\mathcal{T}$. Processing all steps in the tree $\mathcal{A}$ takes a total of $O((\mathcal{T}.sdepth(v) + steps_v) \times \log \sigma)$ time. However, due to getting rid of the requirement of using `getBranches` operations for each step in the search phase, this is notably faster than executing the `search` algorithm in Figure 1 in [PIII] in practice.

5.2.3 Adjusting Minimizing Function

During the construction of the complete path forest $\mathbb{F}$ for $\mathbb{P}$, in the function `add_tree_lr` of Figure 4 in [PIII], it is possible to calculate the actual $steps_v$ for each optimal node v. Whenever a character c is added to $lseq$ or $rseq$ of a node in lines 9, 10 or 14 of Figure 4 in [PIII], $steps_v$ is incremented by 1. Thus using sum of these additions, we can approximate $steps_v$ as a function of $lsize_v$ and $rsize_v$ for the optimal nodes resulting from prepro-

cessing $\mathbb{P}$. Taking advantage of this observation in the following allows improvement of the search times in practice.

As the left and right pushChar operations of a bidirectional index are symmetrical, the most reasonable distinction between the two sizes is by their value. Thus, we assign

$$s \leftarrow \texttt{min(lsize, rsize)} \quad \text{and} \quad l \leftarrow \texttt{max(lsize, rsize)}$$

and approximate the $steps_v$ as a weighted sum

$$w_1 l + w_2 s + w_3 ls + w_4 l^2 + w_5 s^2,$$

resulting in using

$$value \leftarrow (d + w_1 l + w_2 s + w_3 ls + w_4 l^2 + w_5 s^2)/\texttt{get\_df(node)}$$

in place of line 27 of Figure 3 in [PIII].

The weights are initially set to $w = [1, 1, 0, 0, 0]$ to correspond to the original minimizing function. The weights are updated iteratively, doing a least absolute deviations (LAD) fitting of the weights using the data consisting of $[s, l, steps]$ values for all minimal nodes for the original set $\mathbb{P}$. After i iterations of preprocessing with updated w, one can choose the weights resulting in the least total steps for the complete path forest, saving the corresponding forest as the result of the preprocessing.

5.3 Experiments

The algorithms described in Section 5.2 were implemented in C++, compiling with gcc-4.4.5 `-O3 -funroll-loops`. All experiments were run on a single core of Intel®i7 860 @ 2.8 GHz, 16 GB RAM, running Ubuntu 10.10. Functionalities for the construction and the basic operations of compressed suffix trees use the `cst_sct3` class from the Succinct Data Structure Library (SDSL) version 0.9.8 [22]. For the bidirectional index, we use implementation by Schnattinger et al. [71]. See Section 5 in [PIII] for full details on these experiments.

Data. We experimented the preprocessing and the searching using a 50 MB DNA text from the Pizza & Chili Corpus [14]. Sampling from this text, we created pattern sets defined by three parameters. First parameter was the pattern length ($pLen$). The second parameter, *length of origin* (O) defined the length of the text area where the set of patterns would be

obtained from. Third parameter, *coverage* (C) affects the resulting number of patterns ($nPats$) so that $nPats = \lfloor C \times O/pLen \rfloor$ patterns were randomly sampled from this text area. The following ranges of these parameters were studied: $C \in [1\,..\,16]$, $pLen \in [40\,..\,200]$ and $O \in [500\,..\,8000]$, resulting in a total of 480 sets of patterns.

Results. To evaluate the run time of preprocessing, ten repeats of preprocessing with the aforementioned 480 pattern sets were done. Minimizing function weights used for these preprocessing experiments were fixed to $w = [1, 1, 0, 0, 0]$. Average time per symbol rate of the preprocessing, as the function of pattern set size ($pLen \times nPats$), is shown in Figure 5.1, top.

Time taken per symbol increases linearly when the size of the pattern set increases, while $pLen$ and C are fixed. As the coverage C increases, other parameters being fixed, the time per symbol decreases, as can be seen from the decrease of the slope in Figure 5.1, top. As the pattern length $pLen$ increases, other parameters being fixed, the number of patterns in the set will decrease, causing the time per symbol to decrease. Preprocessing times were dominated by the `find_minimizing_node` algorithm of Figure 3 in [PIII], accounting for an average of 94.8% of the time.

The peak memory consumption of preprocessing, calculated as the sum of peak heap size and peak stack size, was tracked for the same collection of 480 pattern sets. The `memusage` tool available from the Pizza & Chili Corpus [14] was used for this task.

The pattern length and coverage have little effect on the peak memory consumption of preprocessing when the pattern set size is fixed. With pattern set sizes starting from 5×10^4 characters, the peak memory consumption increases linearly as the pattern set size increases. Rate of this increasing peak memory consumption was 115 to 125 bytes per symbol, decreasing slightly with shorter patterns and larger pattern sets. The peak memory consumption was between 3.9 and 14.5 MB for all pattern sets.

Preprocessing with the same collection of pattern sets was repeated, this time doing 10 iterations of re-assigning the minimizing function weights, using $w = [1, 1, 0, 0, 0]$ as the starting point, as described in Section 5.2.3. Complete path forests with least total steps were saved for each pattern set.

To evaluate the search times, the `search_forest` algorithm of Figure 4 in [PIII] was run for the 480 pattern sets. Let us denote the execution of this algorithm for the preprocessing done with minimizing func-

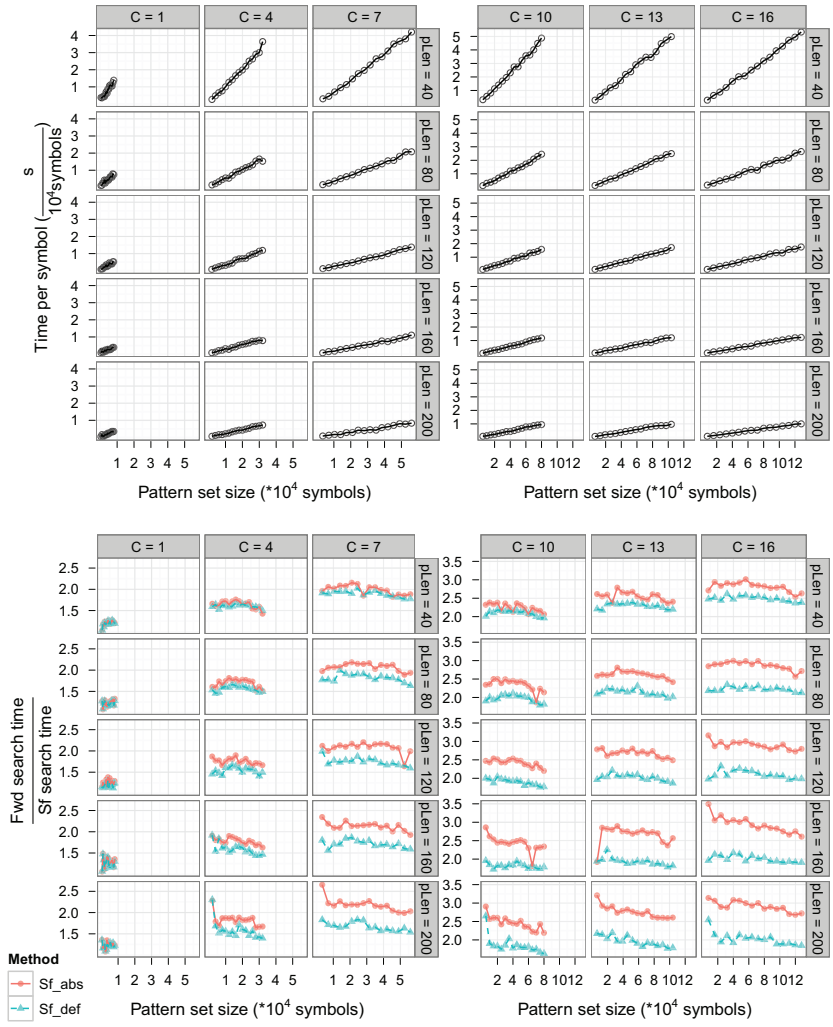


Figure 5.1. Top: Average time per symbol rate ($s/(10^4 symbols)$), over ten repeats of preprocessing, as a function of pattern set size. **Bottom:** Ratio t_{Fwd}/t_{Sf} for both approaches Sf_{abs} and Sf_{def} as a function of pattern set size. In both figures, the coverage varies horizontally from 1 to 16, the pattern length varies vertically from 40 to 200. Note that the axes in the left and right halves of the figures differ. This Figure is a reproduction of Figure 5 in [PIII].

tion weights $w = [1, 1, 0, 0, 0]$ as Sf_{def} and the search done with weights optimized as described above as Sf_{abs}. The performance of these two approaches was compared with searching the same pattern set with forward search of a bidirectional index, calling pushChar operation for each character of a pattern with $d \leftarrow$ right. This is the baseline approach one would use to search the pattern set using a bidirectional index. Let us denote this last approach as Fwd. Let us denote the time taken by the *Fwd* and *Sf* methods for searching a set of patterns as t_{Fwd} and t_{Sf}, respectively. Now, the ratio t_{Fwd}/t_{Sf} as a function of pattern set size, for both approaches Sf_{abs} and Sf_{def} is shown in Figure 5.1, bottom.

In all cases, the Sf methods were faster than Fwd. For both Sf methods, the ratio increases as coverage increases. With Sf_{abs} the increase is more drastic than with the default weights used in preprocessing for Sf_{def}. As the lengths of the patterns increase, Sf_{def} loses its edge over Fwd slightly. However, Sf_{abs} does not suffer from this phenomenon.

Finally, we used memusage to evaluate the peak memory consumption of the three approaches. The memory consumption of the search is dominated by the bidirectional index of the text, occupying 64.14 MB of RAM. For the Fwd method, additional peak memory consumption on top of this was 900–1400 bytes for all pattern sets. For the Sf methods, the respective peak memory consumption range was 12200–20600 bytes. Overall, the added memory consumption of the Sf methods is very small in comparison with the memory requirement of keeping the text index in memory.

Analysis of results. The sizes of pattern sets and text database used in our experiments are arguably somewhat smaller than the data that would be most likely used in a typical metagenomic read alignment setting. Nevertheless, the datasets used were sufficient to show the improvement one can obtain by using the presented techniques. We were able to remarkably improve the speed of searching a pattern set in this setting, due to shown preprocessing. With the coverage value of 16, we were able to commonly obtain 3-fold improvement over the baseline approach. In a metagenomic setting, the read coverage is typically in this range or larger, which should result in a more significant improvement.

One should also note that the reduced search times follow from the reduced number of search steps one has to do to process each pattern set. Thus the improvement was not merely a result of technical and practical tuning of the implementation details.

6. A Graph-theoretical Approach for Motif Discovery in Protein Sequences

Various approaches have been taken for discovering over-represented motifs within a set of protein sequences, including expectation-maximization [4], Gibbs sampling [18, 56] and graph-based [17, 54, 58, 62] approaches. However, most of these approaches have been developed to search motifs of fixed length specifiable by the user or motifs that do not allow for any gaps. While identifying gapped motifs is time consuming, many of the motifs included in databases such as PROSITE [32] and ELM [60] contain gaps of various lengths.

In this chapter, we review a graph-based motif discovery approach initially presented in [PIV]. Our approach is able to search for variable-length motifs and allows for gaps within putative motifs. Another less common advantage of our approach is incrementality, i.e. we can add more sequences to our analysis without rebuilding the graphs from scratch.

In recent years, there have been a few graph-based methods developed for motif discovery in DNA or protein sequences, e.g. [17, 54, 58, 62]. In particular, Pathward et al. [58] also use de Bruijn graphs to search for motifs within a set of protein sequences. However, there are a few essential differences between our approach and theirs. First of all, in a study by Patwardhan, Tang, Kim and Dalkilic [58] the authors construct only one de Bruijn graph for the set of all initial sequences, which can lead to the creation of artificial motifs formed by the concatenation of various segments from different initial sequences. To avoid this problem, we construct one graph for each input sequence, ensuring in this way that the obtained motifs actually appear as subsequences in the input set. Another important difference between the two methods appears in the handling of gaps. In the study by Patwardhan, Tang, Kim and Dalklic [58], the authors modify the initial de Bruijn graph such that each node is replaced by a set of nodes illustrating all possible combinations of gap occurrences

(with the number of gaps being at most half the size of the sequence stored in that node). In our approach, we add a new character every time we count the possible number of amino acids that can occur in a particular position when we traverse the graphs searching for a new motif, and, depending on this number, we decide whether there is a gap for a particular amino acid. Also, after generating a set of potential motifs, we use a combination of four scoring functions to obtain a sorted list of the results.

In the following sections, we review the operational principles of our approach and present experimental results comparing our method with MEME [4], which is one of the most widely used methods in the field, and GLAM2 (Gapped Local Alignment of Motifs) [18], which is a generalized version of the gapless Gibbs sampling algorithm [56].

6.1 Methods

Our graph-theoretical approach uses de Bruijn graphs to search for motifs within a set of protein sequences. When tackling this task, one receives a set of possibly related sequences and aims to identify the substrings that appear significantly more often than other sequences and have some given properties. The focus of this work is on finding arbitrarily long, extensible-length, flexible gap motifs. That is, we search for motifs in the form of regular expressions:

$$A_1 - x(p_1, q_1) - A_2 - x(p_2, q_2) - \cdots - A_r, \tag{6.1}$$

where A_i are continuous sequences of amino acids and $-x(p_i, q_i)-$ represents a gap with a length at least p_i and at most q_i. Moreover, at a given position a block, A_i, may also contain some ambiguous characters, i.e. there might be several choices for the characters appearing on that particular slot.

Our approach consists of three phases: graph construction, graph traversal and the scoring of putative motifs.

6.1.1 Graph Construction

In this section, the alphabet, Σ, consists of 1-letter codes for the 20 amino acids that make up any protein sequence. Let $S_1, \ldots, S_n \in \Sigma^*$ be a set of n protein sequences of lengths $l_1, \ldots, l_n$, respectively, and let k be a fixed parameter. For each input sequence, S_i, we construct a de Bruijn graph, $G_i = (V_i, E_i)$, where V_i and E_i are two disjunct sets, i.e. the set of nodes

and the set of edges. In particular, V_i contains an individual node for all distinct substrings of length k; throughout this section, we will use the term *node* to refer both to a node in this graph and to the corresponding k-mer. If $a_1a_2 \dots a_ka_{k+1}$ is a subword of length $k+1$ in sequence S_i, then we put a directed edge labeled (a_1, a_{k+1}) from the node corresponding to the prefix, $a_1 \dots a_k$, in the direction of the node corresponding to the suffix, $a_2 \dots a_{k+1}$:

$$a_1 \dots a_k \xrightarrow{(a_1,a_{k+1})} a_2 \dots a_{k+1}.$$

Additionally, our approach needs to find the similar node pairs in and between the graphs. To achieve this, we need to define a similarity measure. We measure the similarity between two nodes, v and v', as $S_k(v, v') = \sum_{i=1}^{k} S(v_i, v'_i)$, where $S(v_i, v'_i)$ is the similarity of the characters v_i and v'_i according to the Blosum62 similarity matrix [29].

With a naive direct approach, calculating the similarity between all pairs of N nodes, each representing a string of length k, takes $O(N^2 \times k)$ time. We have devised a variation of the *SS-tree* [79] to optionally improve this in practice. In order to use such similarity indexing, we first need to convert the similarities, $S(x, y)$, between two amino acids, given by the Blosum62 matrix, into distances, $D(x, y)$. Moreover, our similarity indexing approach requires that the triangle inequality of $D(a, b) + D(b, c) \geq D(a, c)$ holds true for all $a, b, c \in \Sigma$. To achieve this, we use a method from [6] to transform the Blosum62 matrix into a metric distance between pairs of amino acids. In particular, we create the distance matrix, D, by setting each cell as follows:

$$D(x, y) = \frac{S(x, x) + S(y, y) - S(x, y) - S(y, x)}{2}.$$

Then, the distance between two k-mers, $u, v \in \Sigma^k$, of length k is $D_k(u, v) = \sum_{i=1}^{k} D(u_i, v_i)$. Since the distance matrix converted from Blosum62 is static, it is easy to check that the triangle inequality holds true for all amino-acid triplets.

In order to decide whether two k-mers are similar to each other, we use a similarity threshold and its transformation into a distance threshold. Essentially, we calculate the average distances between roughly matching and mismatching amino-acid pairs, denoted as mat and mis, respectively. See Section *"De Bruijn graphs construction"* in [PIV] for full details on calculating mat and mis. Now, we define the distance threshold $T_D = k \times (0.8 \times mat + 0.2 \times mis)$. The similarity threshold, T_S, is calculated in nearly identical fashion, using similarity values, $S(x, y)$, instead of the

distances values, $D(x, y)$, when calculating mat and mis.

Now that we have a distance measure and threshold, we can use SS-tree-like similarity indexing when searching for similar node pairs. Our approach constructs a simple tree, initially containing just a blank root node. A new node, v, corresponding to a k-mer can be added to the tree by using the `AddNode(`v`)` function described in lines 1-12 of Figure 1 in [PIV]. All nodes similar to the v node can be found in the tree by using the `FindPairs(`v`)` function described in lines 13-17 of Figure 1 in [PIV]. Once the similar node pairs have been found, they can be linked and the exact similarity, $S_k(i, j)$, of all such n_S pairs can be reported in $O(n_S \times k)$ time. With the linked pairs, we can calculate a property denoted as *weight*, $w(v)$, for each node as follows:

$$w(v) = \sum_{v' \in G_j, j \in [1..n], j \neq i, D_k(v,v') \leq T_D} S_k(v, v'). \tag{6.2}$$

Simultaneously, for each node, $v \in G_i$, and all $1 \leq j \leq n$, $j \neq i$, we identify the node, $v_j \in G_j$, such that

$$S_k(v, v_j) = max_{v' \in G_j} S_k(v, v').$$

Moreover, if $D_k(v, v_j) \leq T_D$, then we add a directed edge from v to v_j. If we have the maximum similarity value for multiple nodes, then we include all directed edges from v to each of these nodes. From now on, we will refer to these edges as *inter-component edges*. If the SS-tree-like optimization described above is not used, the distance threshold condition is replaced with a similarity threshold condition, $S_k(v, v') \leq T_S$, instead in both Equation 6.2, and when adding inter-component edges.

Lastly, we compute the *generalized multiplicity*, $gm(v)$, for each node, $v \in G_i$, which represents the number of graphs accessible through inter-component edges from node v.

6.1.2 Graph Traversal

In the graph traversal phase, our aim is to effectively reduce the search space for motif discovery and produce a list of promising candidate motifs. We start this by first constructing a set:

$$S_{Max} = \{v \in G_1 \cup \cdots \cup G_n \mid gm(v) \geq \tau \times n\},$$

where τ is a parameter indicating the minimal proportion of input sequences required to contain occurrences of the k-mers stored in these

initial nodes. The nodes in the set, S_{Max}, are then sorted in decreasing order based on their gm-values. Furthermore, the nodes with the same gm-value are sorted in decreasing order based on their weights.

We continue by choosing the top node, v, on this list and by following the inter-component edges originating from this node. Let $\mathbb{V}$ now consist of the node v and all the nodes that are connected to it via a single inter-component edge. Using this set of nodes and the graphs containing them, we construct the first consensus word: w_C. Initially we set $w_C = v$ and start adding letters to the left and right of it.

Let us suppose that we want to add letters to the right of it; the other case is symmetric. We define 20 counters, C_X, with $X \in \Sigma$; these counters will be used to decide which letters should be added at each step.

By knowing the set of nodes, $\mathbb{V}$, we can use the counters, C_X, to store the number of nodes that have an outgoing edge with the letter $X \in \Sigma$. More formally, for each $v_i \in \mathbb{V}$ and for each edge, $v_i \xrightarrow{(a_1,a_2)} v_i'$, labeled by (a_1, a_2) with $a_1, a_2 \in \Sigma$, we increase the counter, C_{a_2}. (When we want to add letters to the left of the consensus word, we still need to look at the edges, $v_i' \xrightarrow{(a_1,a_2)} v_i$, labeled by (a_1, a_2) and increase the counter, C_{a_1}.)

Next, we use a combination of two statistical tests, the Q-test and the F-test, to identify the set of amino acids with the highest counter values, C_X, that form a separate set. See the Section *"A novel graph-based motif discovery method"* in [PIV] for full details on this.

As a result, we obtain a sorted list, $\mathfrak{C}$, corresponding to a set of amino acids, $\mathfrak{A}$, which will be output in square brackets in our regular expression representation to indicate that each of its elements is allowed to appear at the given position in the motif. Moreover, if the size of this set is larger than a given threshold, then we consider that position to be a gap in our motif, i.e. any amino acid is a successful match; we take 9 as the value for this threshold.

After identifying the letters added to the consensus word, in each graph, we follow the edges of the nodes, $v_i \in \mathbb{V}$, which are labeled by the letter corresponding to the highest counter value. If there is no such edge, then we will move alongside the edge labeled by the letter corresponding to the next value in the ordered list, $\mathfrak{C}$. If in some graph there is no possible alternative to continue the traversal, then we simply remove the graph and the corresponding initial sequence from our search procedure. To simulate a breakpoint in the motif, we use a parameter, $p_b \geq 0$, bounding the number of consecutive gaps allowed.

We also ensure that the number of times any node is visited during traversal is never greater than the number of times that the corresponding k-mer occurs in the corresponding initial sequence.

Once a putative motif, w_C, has been constructed, we save its gm, weight and the number of sequences that correspond to the set $\mathbb{V}$ at the end of the traversal, which we denote as the support, or $Sup(w_C)$. Following this, we mark the initial node, v, and decrease the weights of all visited nodes to make it less probable that an iteration of traversal will be initialized from them. Finally, we re-sort the S_{Max} list and take the new non-marked top node, v', as the next starting point for the traversal. In this way, we generate a set of motif candidates from the entries in S_{Max} that are over-represented in the set of input sequences.

6.1.3 Scoring of Putative Motifs

To measure the significance of a candidate motif, M, we use the summation of four different functions: (i) the *generalized multiplicity* and (ii) *weight* of the initial node from which a particular motif was found, (iii) the *LogOdd* measure providing the degree of surprise for M and (iv) a newly introduced measure called *credibility*. All of these scores are normalized to fit in the range of $[0, 1]$.

Generalized multiplicity and weight were explained in Section 6.1.1. These measures indicate two levels of repetitiveness for the initial node, v, from which a particular motif, M, was found.

The $LogOdd$ measure compares the observed frequency of occurrence of a given motif with the expected probability of occurrence, which can be computed using a given background distribution. For a given candidate motif, M, we compute its $LogOdd$ value using the following formula:

$$LogOdd(M) = log(\frac{(1/n) \times Sup(M)}{P(M)}), \tag{6.3}$$

where n is the number of input sequences we want to analyze, $Sup(M) \leq gm(M)$ is the support of M, i.e. the number of graphs that remained at the end of our search for M, and $P(M)$ is the expected probability of M.

To compute $P(M)$, we can use the following formula:

$$P(M) = \prod_{j=1}^{l} \sum_{r=1}^{|B_j|} P(b_{j,r}), \tag{6.4}$$

where $P(b_{j,r})$ is the frequency of the character $b_{j,r}$ occurring on the j-th position in M, which is computed using the background distribution, and

$r \in [1, |B_j|]$ runs through the characters at each position of our regular expression presentation of the putative motif. For instance, if $M = AV[GC]$, then its probability is $P(M) = P(A) \times P(V) \times (P(G) + P(C))$. The background distribution used was chosen based on the frequencies of amino acids in the Swiss-Prot database [5].

The fourth function used in our scoring schema is the *credibility* measure, which is defined as the average (over the length of the motif) of the counter values of the amino acids appearing at each position $1 \leq j \leq l$ of the motif and normalized based on the sum of the counter values at each position. When we have several possibilities for amino acids at a certain position, j, i.e. $B_j = [b_{j,1}; \ldots; b_{j,k}]$ with $b_{j,1}, \ldots, b_{j,k} \in \Sigma$, we then compute the summation of all the corresponding counters, $\sum_{r=1}^{|B_j|} C_{b_{j,r}}$. Let $C_{j,\cdot}$ be the sum of all counters at position j. Now, the credibility measure is computed as follows:

$$Cred(M) = \frac{1}{l} \sum_{j=1}^{l} \sum_{r=1}^{|B_j|} \frac{C_{b_{j,r}}}{C_{j,\cdot}}. \tag{6.5}$$

All candidate motifs produced during the search step are scored and ranked using the sum of these four scores. This ranked list, or a prefix of it, is provided to the user. In addition to the regular expression form, we can output the sequence logos based on the distribution of counter values at each position of the candidate motifs.

6.2 Experiments

We implemented the motif discovery approach, which was reviewed in Section 6.1 and more elaborately described in the Section *"Methods"* in [PIV], with and without the SS-tree-like optimization. The implementations were done with C++ and compiled with gcc-4.6.3 `-O3`. We compared the performance of these approaches with the performance of the MEME [4] and GLAM2 [18] tools. All of the experiments were run on an Intel i7 860 2.8 GHz (8192 kB cache) with 16 GB RAM, while running Ubuntu 12.04. In this section, we review these experiments - please see the Section *"Results and discussion"* in [PIV] for full details on these experiments.

Data. We initially chose a selection of 80 sequence collections from the PROSITE [32] database as our data. For practical reasons, we chose the smallest 30 sequence collections as our training data, which were used

Table 6.1. Fraction of motifs found within the top five results reported by each of the four approaches: MEME, GLAM2, DB and DB-SS.

	MEME	GLAM2	DB	DB-SS
fract.	0.74	0.50	0.70	0.58

while developing the method. This training set was then used when making our parameter choices. The remaining 50 sequence collections were left untouched as our test data in order to compare the predictive quality of the four approaches. Together, these 80 sequence collections totalled 5.3 MB in size and contained 26938 sequences. The data also contained a known motif in regular expression form for each of the sequence collections.

Results. To compare the predictive quality of these four approaches, we ran all of the tools with similar parameters on the test set and analyzed the number of known motifs that were correctly found. Parameters were chosen that would list up to 5 suggested motifs at a length of up to 50 amino acids in order to compare the results in a fair manner. Based on tests conducted on our training set using the de Bruijn approach, we chose to allow up to three consecutive gaps and set the parameter as $\tau = 0.625$. The fraction of known motifs listed within the top 5 results by each tool are shown in Table 6.1. We denote the de Bruijn approach using the abbreviation DB and the approach using SS-tree-like similarity indexing using the abbreviation DB-SS.

As can be seen from Table 6.1, the quality of the results provided by the DB approach is on par with those provided by the MEME tool. The quality of the results given by DB-SS is slightly above that of GLAM2.

Moreover, we recorded the query times of all four approaches for all 80 sequence collections using the same parameters as above. The query times as a function of the sequence collection sizes in bytes are shown in Figure 2 in [PIV]. With respect to the query times, DB-SS notably outperforms GLAM2 on smaller sequence collections and has quite similar run time on larger collections. The DB approach outperforms MEME by a notable margin on all of the sequence collections.

To encapsulate this, we calculated the geometric means of ratios t_{MEME}/t_{DB} and t_{GLAM2}/t_{DB-SS} over all sequence collections, where t_X is the time taken for method X to process a sequence collection. The DB-SS approach, producing results similar in quality with GLAM2 approach

was notably faster than it, demonstrated by the geometric mean of the ratios t_{GLAM2}/t_{DB-SS}, which was 5.69. In similar fashion, the DB approach, which produced results of a comparable quality with MEME approach was notably faster than it, as shown by the geometric mean of the ratios t_{MEME}/t_{DB}, which was 9.02. Our main result is this shown superior run time versus quality tradeoff achieved using our approaches. The differences in the arithmetic means of the ratios over all of the sequence collections were even more dramatic: 18.26 and 10.72, respectively. To encapsulate the effect of similarity indexing, we note that the geometric mean of the ratio t_{DB}/t_{DB-SS} over all sequence collections was 3.96.

Analysis of results. We have shown that by using a graph-theoretical approach, it is possible to achieve faster query times than with previous methods, while retaining a similar quality in terms of the results. Our results suggest, that by reducing the search space explored during the scoring phase, we are able to notably speed up motif discovery and throw away putative motifs that would get a high score but would not correspond to real motifs. Furthermore, in our setting it is possible to use similarity indexing to further reduce the query times.

7. Discussion

7.1 Approximate Alignment of Long Patterns

In Chapter 3, we reviewed an approach combining block addressing with q-sampling, which provides an efficient method for the indexed approximate alignment of long patterns. This approach is faster than previous methods and has a smaller memory footprint, which can be further adjusted. Both q-sampling and block addressing effectively reduce the size of the resulting index structure when comparing it with a full text index. As long as the size of the text blocks in this setting remains relatively small, doing a more precise alignment between the pattern and a text block is a relatively swift process.

As long as the patterns are long enough, q-sampling is a very suitable strategy for this type of problem setting. With long patterns having relatively few occurrences, the time taken retrieving individual text blocks is more or less negligible, making block addressing a fitting strategy for this setting as well. In this setting, as demonstrated in Section 3.3, BLAST-like approaches were shown to be computationally excessive.

Block addressing and q-sampling may also prove useful in a setting where full text indexes or even compressed indexes are simply too big to fit in memory. We would like to note that in our experiments with a full human genome, the index structure of GAST required 0.23 bits per character, while, for example, the index structure used by `bowtie` in this setting would require 6.56 bits per character [39].

Regarding future work, the combination of block addressing and q-sampling in a multi-pattern setting should be a concept worth exploring. Since multiple patterns might contain identical AC-probes and be found in the same text blocks, it is clear that it is possible to save computational

resources in this setting.

7.2 Indexed Matching of Multiple Patterns

In Chapters 4 and 5, we reviewed two practical approaches for improving indexed, exact multi-pattern matching. The results presented here show that the reviewed methods for preprocessing a set of patterns will notably improve the speed of searching for such a set in an indexed text. The basic principle of both of these approaches was similar in the sense that both start by searching for initial subpatterns or substrings common to multiple patterns and continue the search from there. Abandoning locating as an intermediate step resulted in a major and remarkable difference, one which yielded promising results on a realistic data set with properties common in a metagenomic setting.

However, in terms of the approach reviewed in Chapter 4, we would like to underline an observation that may not be so obvious. To some extent, the speedup seen in Section 4.2 is caused by what we denote as the *elbow-point effect*, which results from the above-mentioned intermediate locating step. Let us consider an index where extending or taking additional steps in the search process would computationally be more expensive than doing a simple character-by-character comparison between a pattern sequence and a text sequence. Now, a number of searches will have an *elbow-point*, where the number of occurrences of the subpattern corresponding to the current search step is so small that it would computationally be more expensive to extend the search to find the occurrences of the full pattern than to do character-by-character comparisons at each occurrence of the current subpattern to see if this is an actual occurrence of the full pattern or not. As long as access to the text and the character-by-character comparison is faster than extending the search, this is a viable strategy. This can be done with a separate uncompressed or swiftly uncompressable copy of the text.

This is a line of research we would like to further pursue, as the resulting gains may be widely applicable. This strategy could prove useful even beyond the scope of multi-pattern matching, improving the search of individual patterns as well.

The biggest issue in terms of the practicality of the approaches reviewed in Chapter 5 is that supporting an exact search will be sufficient only when cutting the reads into smaller pieces. However, one can sup-

port an approximate search using a general backtracking mechanism inside the bidirectional search, but to do this efficiently the existing pruning mechanisms (like in [40, 43, 44, 47]) need to be modified or new ones need to be introduced that will work within our search scheme. Also, the subpattern cover needs to be refined in order to guarantee that all of the approximate occurrences will be found.

A number of approaches for improving the preprocessing reviewed in Section 5.2.1 are mentioned in Section 6 in [PIII]. Most importantly, one should be able to improve the speed of preprocessing drastically by selecting more than one subpattern for each constructed compressed suffix tree. While this would affect the estimation error of $\log p$ in terms of finding the optimal subpatterns, a faster practical solution resulting in a roughly similar preprocessing quality could most probably be found. It is very likely that this kind of approach will be needed to preprocess significantly larger pattern sets. Currently, the speed of preprocessing poses the greatest challenge for working with such sets of patterns.

7.3 Motif Discovery

In Chapter, 6 we reviewed a graph-theoretical protein motif discovery approach and an optional performance improvement for it, based on similarity indexing. We have shown that these two approaches, the DB and the DB-SS approaches were able to perform as well as or better than the MEME and GLAM2 approaches. With respect to query times, the DB and the DB-SS approaches outperform the MEME and GLAM2 approaches in nearly all cases. With respect to the quality of the results, the DB approach is comparable to the MEME approach and the DB-SS approach is comparable to the GLAM2 approach. Incrementality is another advantage that the described graph-theoretical approaches have over the traditional approaches. It is possible to add additional sequences to our analysis without rebuilding the graphs from scratch.

Regarding future work, the most interesting direction would be to pursue a smaller drop in quality when using similarity indexing, as the performance improvement is relatively impressive. We have also considered the possibility of using our approach for data with a smaller alphabet, e.g. DNA. In this setting, we expect results of a similar quality by using a strategy built on top of an idea of using 2-mers from this smaller alphabet as representatives of a single character. This will require slight alter-

ations, e.g. considering two reading frames, but this seems achievable by adjusting our current implementation process.

Bibliography

[1] Stephen F. Altschul and Bruce W. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Mathematical Biology*, 48:603–616, 1986.

[2] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[3] Ricardo A. Baeza-Yates and Gonzalo Navarro. Block addressing indices for approximate text retrieval. *Journal of the American Society for Information Science (JASIS)*, 51(1):69–82, 2000.

[4] Timothy L. Bailey and Charles Elkan. Fitting a mixture model by expectation maximization to discover motifs in biopolymers. In *Proceedings of the International Conference on Intelligent Systems for Molecular Biology (ISMB)*, volume 2, pages 28–36. Department of Computer Science and Engineering, University of California, San Diego, 1994.

[5] Amos Bairoch, Brigitte Boeckmann, Serenella Ferro, and Elisabeth Gasteiger. Swiss-Prot: juggling between evolution and stability. *Briefings in Bioinformatics*, http://www.expasy.org/sprot/ [cited April 9, 2013], 5(1), 2004.

[6] Julie Baussand and Alessandra Carbone. Inconsistent distances in substitution matrices can be avoided by properly handling hydrophobic residues. *Evolutionary Bioinformatics*, 4:255–261, 2008.

[7] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. In *Proceedings of the European Conference on Algorithms (ESA)*, volume 6942 of *LNCS*, pages 748–759. Springer, 2011.

[8] Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[9] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391. ACM/SIAM, 1996.

[10] Sean R. Eddy. What is dynamic programming? *Nature Biotechnology*, 22(7):909–910, 2004.

[11] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. IEEE Computer Society, 2000.

[12] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[13] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.

[14] Paolo Ferragina and Gonzalo Navarro. Pizza & chili corpus, compressed indexes and their testbeds. http://pizzachili.dcc.uchile.cl/ [cited May 12, 2011].

[15] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Compututer Science*, 410(51):5354–5364, 2009.

[16] Johannes Fischer, Veli Mäkinen, and Niko Välimäki. Space efficient string mining under frequency constraints. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 193–202. IEEE Computer Society, 2008.

[17] Eugene Fratkin, Brian T. Naughton, Douglas L. Brutlag, and Serafim Batzoglou. MotifCut: regulatory motifs finding with maximum density subgraphs. *Bioinformatics*, 22(14):156–157, 2006.

[18] Martin C. Frith, Neil F. W. Saunders, Bostjan Kobe, and Timothy L. Bailey. Discovering sequence motifs with arbitrary insertions and deletions. *PLoS Computational Biology*, 4(5), 2008.

[19] Travis Gagie, Kalle Karhu, Juha Kärkkäinen, Veli Mäkinen, Leena Salmela, and Jorma Tarhio. Indexed multi-pattern matching. In *Proceedings of the Latin American Symposium on Theoretical Informatics (LATIN)*, volume 7256 of *LNCS*, pages 399–407. Springer, 2012.

[20] Travis Gagie, Simon J. Puglisi, and Andrew Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 1–6. Springer, 2009.

[21] Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.

[22] Simon Gog. SDSL — succinct data structure library 0.9.5. http://www.uni-ulm.de/in/theo/research/sdsl.html [cited September 10, 2011].

[23] Simon Gog, Kalle Karhu, Juha Kärkkäinen, Veli Mäkinen, and Niko Välimäki. Multi-pattern matching with bidirectional indexes. In *Proceedings of the International Computing and Combinatorics Conference (COCOON)*, volume 7434 of *LNCS*, pages 384–395. Springer, 2012.

[24] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In *Information Retrieval: Data Structures and Algorithms*, pages 66–82. Prentice-Hall, 1992.

[25] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850. ACM/SIAM, 2003.

[26] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[27] Jo Handelsman, Michelle R. Rondon, Sean F. Brady, Jon Clardy, and Robert M. Goodman. Molecular biological access to the chemistry of unknown soil microbes: a new frontier for natural products. *Chemistry & Biology*, 5:245–249, 1998.

[28] Dov Harel and Robert E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[29] Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences (PNAS)*, 89(22):10915–10919, 1992.

[30] Tim J. P. Hubbard, Bronwen L. Aken, Kathryn Beal, Benoit Ballester, Mario Cáccamo, Yuan Chen, Laura Clarke, Guy Coates, Fiona Cunningham, Tim Cutts, Thomas Down, S. C. Dyer, Stephen Fitzgerald, Julio Fernandez-Banet, Stefan Gräf, Syed Haider, Martin Hammond, Javier Herrero, Richard C. G. Holland, Kevin L. Howe, Kerstin Howe, Nathan Johnson, Andreas Kähäri, Damian Keefe, Felix Kokocinski, Eugene Kulesha, Daniel Lawson, Ian Longden, Craig Melsopp, Karine Megy, Patrick Meidl, Bert Overduin, Anne Parker, Andreas Prlic, S. Rice, Daniel Rios, Michael Schuster, I. Sealy, Jessica Severin, Guy Slater, Damian Smedley, Giulietta Spudich, S. Trevanion, Albert J. Vilella, Jan Vogel, Simon White, M. Wood, Tony Cox, Val Curwen, Richard Durbin, Xosé M. Fernández-Suarez, Paul Flicek, Arek Kasprzyk, Glenn Proctor, Stephen M. J. Searle, James Smith, Abel Ureta-Vidal, and Ewan Birney. Ensembl 2007. *Nucleic Acids Research*, 35(Database-Issue):610–617, 2007.

[31] Lucas C. K. Hui. Color set size problem with application to string matching. In *Proceedings of Symposium on Combinatorial Pattern Matching (CPM)*, volume 644 of *LNCS*, pages 230–243. Springer, 1992.

[32] Nicolas Hulo, Amos Bairoch, Virginie Bulliard, Lorenzo Cerutti, Edouard De Castro, Petra S. Langendijk-Genevaux, Marco Pagni, and Christian J. A. Sigrist. The prosite database. *Nucleic Acids Research*, 34(Database-Issue):227–230, 2006.

[33] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Ultra-succinct representation of ordered trees. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584. ACM/SIAM, 2007.

[34] Samuel Karlin and Cristopher B. Burge. Dinucleotide relative abundance extremes: a genomic signature. *Trends in Genetics*, 11(7):283–290, 1995.

[35] W. James Kent. BLAT - The BLAST-like alignment tool. *Genome Research*, 12:656–664, 2002.

[36] Victor Kunin, Alex Copeland, Alla Lapidus, Konstantinos Mavromatis, and Philip Hugenholtz. A bioinformatician's guide to metagenomics. *Microbiology and Molecular Biology Reviews*, 72(4):557–578, 2008.

[37] Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon C. K. Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 31–36. IEEE Computer Society, 2009.

[38] Gad M. Landau and Michal Ziv-Ukelson. On the common substring alignment problem. *Journal of Algorithms*, 41(2):338–359, 2001.

[39] Ben Langmead and Cole Trapnell. Bowtie: An ultrafast, memory-efficient short read aligner. http://bowtie-bio.sourceforge.net/index.shtml [cited April 2, 2013].

[40] Ben Langmead, Cole Trapnell, Miihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[41] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proceedings of the Data Compression Conference (DCC)*, pages 296–305. IEEE Computer Society, 1999.

[42] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[43] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–60, 2009.

[44] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[45] Martin S. Lindner and Bernhard Y. Renard. Metagenomic abundance estimation and diagnostic testing on species level. *Nucleic Acids Research*, 2012.

[46] Moritz G. Maaß. Linear bidirectional on-line construction of affix trees. *Algorithmica*, 37(1):43–74, 2003.

[47] Veli Mäkinen, Niko Välimäki, Antti Laaksonen, and Riku Katainen. Unified view of backward backtracking in short read mapping. In *Algorithms and Applications*, pages 182–195. Springer, 2010.

[48] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22.5:935–948, 1993.

[49] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. *Proceedings of the USENIX Winter Conference*, pages 23–32, 1994.

[50] Volker Matys, Olga V. Kel-Margoulis, Ellen Fricke, Ines Liebich, Sigrid Land, A. Barre-Dirrie, Ingmar Reuter, D. Chekmenev, Mathias Krull, Klaus Hornischer, Nico Voss, Philip Stegmaier, Birgit Lewicki-Potapov, H. Saxel, Alexander E. Kel, and Edgar Wingender. Transfac® and its module transcompel®: transcriptional gene regulation in eukaryotes. *Nucleic Acids Research*, 34(Database-Issue):108–110, 2006.

[51] Aleksandr Morgulis, George Coulouris, Yan Raytselis, Thomas L. Madden, Richa Agarwala, and Alejandro A. Schäffer. Database indexing for production MegaBLAST searches. *Bioinformatics*, 24(16):1757–1764, 2008.

[52] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[53] National Center for Biotechnology Information. Blast: Basic local alignment search tool. http://www.ncbi.nlm.nih.gov/BLAST/ [cited Mar 24, 2009], 2009.

[54] Brian T. Naughton, Eugene Fratkin, Serafim Batzoglou, and Douglas L. Brutlag. A graph-based motif detection algorithm models complex nucleotide dependencies in transcription factor binding sites. *Nucleic Acids Research*, 34(20), 2006.

[55] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[56] Andrew F. Neuwald, Jun S. Liu, and Charles E. Lawrence. Gibbs motif sampling: detection of bacterial outer membrane protein repeats. *Protein Science*, 4:1618–1632, 1995.

[57] Ontario Institute for Cancer Research and European Bioinformatics Institute. Biomart project. http://www.biomart.org [cited May 3, 2010].

[58] Rupali Patwardhan, Haixu Tang, Sun Kim, and Mehmet M. Dalkilic. An approximate de Bruijn graph approach to multiple local alignment and motif discovery in protein sequences. In *Proceeding of: Data Mining and Bioinformatics, First International Workshop*. Springer, 2006.

[59] Giulio Pavesi, Paolo Mereghetti, Giancarlo Mauri, and Graziano Pesole. Weeder web: discovery of transcription factor binding sites in a set of sequences from co-regulated genes. *Nucleic Acids Research*, 32(Web-Server-Issue):199–203, 2004.

[60] Pål Puntervoll, Rune Linding, Christine Gemünd, Sophie Chabanis-Davidson, Morten Mattingsdal, Scott Cameron, David M. A. Martin, Gabriele Ausiello, Barbara Brannetti, Anna Costantini, Fabrizio Ferrè, Vincenza Maselli, Allegra Via, Gianni Cesareni, Francesca Diella, Giulio Superti-Furga, Lucjan Stanislaw Wyrwicz, Chenna Ramu, Caroline McGuigan, Rambabu Gudavalli, Ivica Letunic, Peer Bork, Leszek Rychlewski, Bernhard Küster, Manuela Helmer-Citterich, William N. Hunter, Rein Aasland, and Toby J. Gibson. ELM server: a new resource for investigating short functional sites in modular eukaryotic proteins. *Nucleic Acids Research*, 31(13):3625–3630, 2003.

[61] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the ACM-SIAM Symposium on Discrete algorithms (SODA)*, pages 233–242. ACM/SIAM, 2002.

[62] Timothy E. Reddy, Charles DeLisi, and Boris E. Shakhnovich. Binding site graphs: A new graph theoretical framework for prediction of transcription factor binding sites. *PLoS Computational Biology*, 3(5), 2007.

[63] Frederick P. Roth, Jason D. Hughes, Preston W. Estep, and George M. Church. Finding DNA regulatory motifs within unaligned noncoding sequences clustered by whole-genome mRNA quantitation. *Nature Biotechnology*, 16:939–945, 1998.

[64] Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7:53:1–53:34, September 2011.

[65] Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceedings of International Symposium on Algorithms and Computation (ISAAC)*, volume 1969 of *LNCS*, pages 410–421. Springer, 2000.

[66] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[67] Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5:12–22, 2006.

[68] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41:589–607, December 2007.

[69] Leena Salmela, Jorma Tarhio, and Jari Kytöjoki. Multi-pattern string matching with q-grams. *ACM Journal of Experimental Algorithms*, 11(1), 2006.

[70] Albin Sandelin, Wynand Alkema, Pär G. Engström, Wyeth W. Wasserman, and Boris Lenhard. Jaspar: an open-access database for eukaryotic transcription factor binding profiles. *Nucleic Acids Research*, 32(Database-Issue):91–94, 2004.

[71] Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees. In *Proceedings of the Conference on Combinatorial Pattern Matching (CPM)*, volume 6129 of *LNCS*, pages 40–50. Springer, 2010.

[72] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[73] Jens Stoye. Affix trees. Technical Report 2000-04, Faculty of Technology, Bielefeld University, 2000. http://www.techfak.uni-bielefeld.de/~stoye/rpublications/report00-04.pdf.

[74] Erkki Sutinen and Jorma Tarhio. Filtration with q-samples in approximate string matching. In *Proceedings of the 7th Symposium on Combinatorial Pattern Matching (CPM)*, volume 1075 of *LNCS*, pages 50–63. Springer, 1996.

[75] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[76] Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[77] Peter Weiner. Linear pattern matching algorithm. In *Proceedings of the IEEE Symposium on Switching and Automata Theory*, pages 1–11. IEEE Computer Society, 1973.

[78] Kris A. Wetterstrand. DNA sequencing costs: Data from the NHGRI genome sequencing program (GSP) [on-line]. http://www.genome.gov/sequencingcosts [cited Mar 27, 2013].

[79] David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proceedings of the International Conference on Data Engineering*, pages 516–523. IEEE Computer Society, 196.

[80] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, Second edition, 1999.

[81] Thomas D. Wu and Colin K. Watanabe. GMAP: a genomic mapping and alignment program for mRNA and EST sequences. *Bioinformatics*, 21(9):1859–1875, 2005.

[82] Shang-Hong Zhang and Ya-Zhi Huang. Characteristics of oligonucleotide frequencies across genomes: Conservation versus variation, strand symmetry, and evolutionary implications. *Nature Precedings*, hdl:10101/npre.2008.2146.1, 2008.

[83] Shang-Hong Zhang and Jian-Hua Yang. Conservation versus variation of dinucleotide frequencies across genomes: Evolutionary implications. *Genome Biology*, 6(11):1–21, 2005.

[84] Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7:203–214, 2000.

The cost of obtaining biologically relevant data via sequencing has been declining rapidly, far surpassing the decline in computing costs. This is highlighting a need for more efficient, and thus cheaper, ways to analyze all of this data. Analyzing such data commonly requires searching through the text representing it in one way or another. The focus of this thesis is on improving the efficiency of the computational approaches that one may wish to use when searching through such texts. More precisely, it addresses three subproblems related to text searches in bioinformatics.

The subproblems considered in this thesis are (i) approximate, indexed alignment of long sequences, (ii) indexed multi-pattern matching, and (iii) protein motif discovery. We present new theoretical insights, practical improvements and experimental results related to these subproblems.

Aalto-DD 130/2013

ISBN 978-952-60-5298-4
ISBN 978-952-60-5299-1 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS